Solution
Sheet 3

Bernhard Westfechtel

Exercise 4

Algebraic specification of queues

```
module Queue;
   import Bool, true, false from Bool;
      Nat, zero from Nat;
   export all;
   sort Queue;
   operations
      newqueue : -> Queue;
      enqueue : Queue x Nat -> Queue;
      isempty : Queue -> Bool;
      dequeue : Queue -> Queue;
      head : Queue -> Nat;
      tail: Queue -> Nat;
   declare q : Queue; n,m : Nat;
   axioms
      isempty(newqueue) == true;
      isempty(enqueue(q,n)) == false;
      dequeue(newqueue) == newqueue;
      dequeue(enqueue(newqueue,n)) == newqueue;
      dequeue(enqueue(enqueue(q,m),n)) == enqueue(dequeue(enqueue(q,m)),n);
      tail(newqueue) == zero;
      tail(enqueue(q,n)) == n;
      head(newqueue) == zero;
      head(enqueue(newqueue,n)) == n;
      head(enqueue(enqueue(q,m),n)) == head(enqueue(q,m));
end module Queue;
```

Decomposition into constructors and other operations

```
module Queue;
   import Bool, true, false from Bool;
      Nat, zero from Nat;
   export all;
   sort Queue;
   constructors
      newqueue : -> Queue;
      enqueue : Queue x Nat -> Queue;
   operations
      isempty : Queue -> Bool;
      dequeue : Queue -> Queue;
      head : Queue -> Nat;
      tail: Queue -> Nat;
   declare q : Queue; n,m : Nat;
   operation axioms
      isempty(newqueue) == true;
      isempty(enqueue(q,n)) == false;
      dequeue(newqueue) == newqueue;
      dequeue(enqueue(newqueue,n)) == newqueue;
      dequeue(enqueue(enqueue(q,m),n)) == enqueue(dequeue(enqueue(q,m)),n);
      tail(newqueue) == zero;
      tail(enqueue(q,n)) == n;
      head(newqueue) == zero;
      head(enqueue(newqueue,n)) == n;
      head(enqueue(enqueue(q,m),n)) == head(enqueue(q,m));
end module Queue;
```

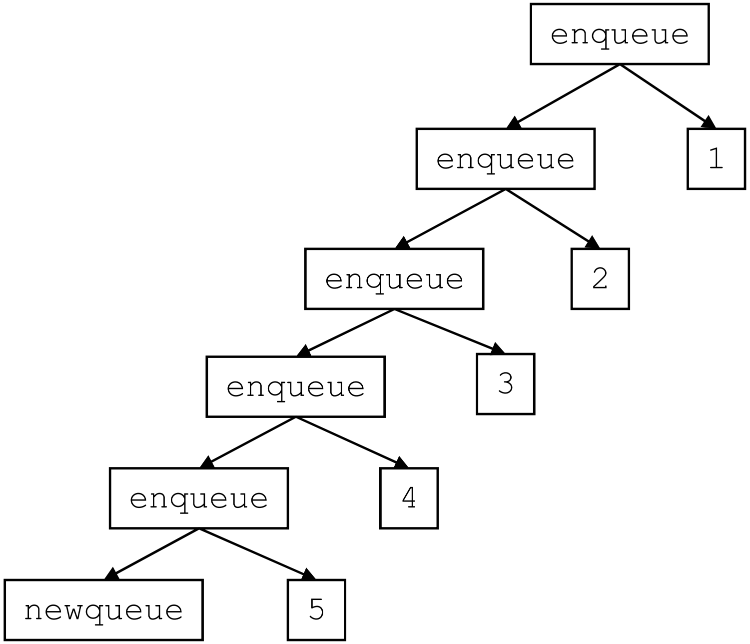
Constraints for constructive specifications

- The outermost operation of a left-hand side of an axiom is no constructor, all inner operations are constructors
- A variable occurs at most once on the left-hand side
- All variables of the right-hand side occur on the left-hand side
- The system of axioms is **unique** with respect to a (non-constructor) operation, i.e. for each tuple of argument terms there is at most one matching rule
- The system of axioms is **complete** with respect to a (non-constructor) operation, i.e. for each tuple of argument terms there is at least one matching rule
- The system of axioms is **terminating**, i.e. for variable-free terms there are only derivations of finite length

Representation of queues

`q = 1 2 3 4 5` Informal representation

`q = enqueue(enqueue(enqueue(enqueue(enqueue(newqueue,5),4),3),2),1)` Term



Tree

Execution of `head(q)`

```
head(q) ==
head(enqueue(enqueue(enqueue(enqueue(enqueue(newqueue,5),4),3),2),1)) ==
head(enqueue(enqueue(enqueue(enqueue(newqueue,5),4),3),2)) ==
head(enqueue(enqueue(enqueue(newqueue,5),4),3)) ==
head(enqueue(enqueue(newqueue,5),4)) ==
head(enqueue(newqueue,5)) ==
5
```

Execution of `head(q)` requires linear time and should require constant time!

Exercise 5

Specification of `ArrayNatNat`

```
module ArrayNatNat;
   (* Record composed of an array and two natural numbers. *)
   import Array from Array; Nat from Nat;
   export all;
   sort ArrayNatNat;
   constructor (_,_,_) : Array x Nat x Nat -> ArrayNatNat;
   operations
      arrayOf _ : ArrayNatNat -> Array;
      nat1Of _ : ArrayNatNat -> Nat;
      nat2Of _ : ArrayNatNat -> Nat;
      _[_/array] : ArrayNatNat x Array -> ArrayNatNat;
      _[_/nat1] : ArrayNatNat x Nat -> ArrayNatNat;
      _[_/nat2] : ArrayNatNat x Nat -> ArrayNatNat;
   declare a : Array; n1, n2 : Nat; ann : ArrayNatNat;
   operation axioms
      arrayOf((a,n1,n2)) == a;
      nat1Of((a,n1,n2)) == n1;
      nat2Of((a,n1,n2)) == n2;
      ann[a/array] == (a, nat1Of ann, nat2Of ann);
      ann[n1/nat1] == (arrayOf ann, n1, nat2Of ann);
      ann[n2/nat2] == (arrayOf ann, nat1Of ann, n2);
end module ArrayNat;
```

Implementation of queues

```
module ↓Queue;
   import ArrayNatNat, (_,_,_), arrayOf _, nat1Of _, nat2Of _ from ArrayNatNat;
      Array, empty, _[_/_], read from Array; Bool from Bool;
      Nat, zero, succ, pre, _ = _, _ < _ from Nat
         rename Nat as Item, zero as 0, zero as error, succ as _+1, pre as _-1;
   export all;
   operations
      ↓newqueue : -> ArrayNatNat;
      ↓enqueue : ArrayNatNat x Item -> ArrayNatNat;
      ↓isempty : ArrayNatNat -> Bool;
      ↓dequeue : ArrayNatNat -> ArrayNat;
      ↓head : ArrayNatNat -> Item;
      ↓tail : ArrayNatNat -> Item;
   declare ann : ArrayNat; it, it1, it2 : Item; a : Array;
   axioms
      ↓newqueue == (empty, 0, 0);
      ↓enqueue(ann,it) ==
         (arrayOf ann[it/nat2Of ann], nat1Of ann, nat2Of ann + 1);
      ↓isempty((a, n1, n2)) == n1 = n2;
      ↓dequeue((a, n1, n2) ==
         if n1=n2 then (a, n1, n2) else (a, n1+1, n2);
      ↓head((a,n1,n2)) == if n1 = n2 then error else read(a,n1);
      ↓tail((a,n1,n2)) == if n2 = n1 then error else read(a,n2-1);
end module ↓Queue;
```

Data representations

- Assuming a built-in array type, each queue would be represented by an array `a` with minimal index `n1` and maximal index `n2-1`
- All operations require constant time
- `head` accesses the element with index `n1`