```
ADT Stack
Uses Element, boolean
operations
  create   : → Stack
  isEmpty: Stack → boolean
  push: Stack × Element → Stack
  pop: Stack ⇀ Stack
  top  : Stack ⇀ Element
preconditions
  pop(p) iff !isEmpty(p)
  top(p) iff !isEmpty(p)
axioms
  isEmpty(create) ≡ true // A1
  isEmpty(push(p,e)) ≡ false // A2
  top(push(p,e)) ≡ e // A3
  pop(push(p,e)) ≡ p // A4
```

Operations *pop* and *top*, being not defined everywhere, need pre-conditions.

A few notes about axioms:

- A1: we make shure that a newly created stack must be empty
- A2: as soon as we push an element on top of the stack, it is not empty any more
- A3: when we push an element on top of the stack, the (new) top is the element we just pushed
- A4: if we pop the stack just after pushing an element, we go back to the initial stack (before the push / pop couple)

3.2 The Queue ADT

A queue is an homogeneous collection of elements accessed by one of the two ends of the structure: insertion at the one end, removals at the other end. Intuitively, it mirrors the behaviour of a real queue, such as the one that can appear at the entrance of a cinema for instance. It can be called FIFO, for *First In, First Out*: the first element that entered the queue is the first to leave it.

Expected functionalities are as follows:

- Tell wether the queue is empty
- Add an element to the end of the queue
- Remove the first (head) element from the queue
- Tell which is the head element

From those functionalities we get the following ADT:

```
ADT queue
Uses boolean, Element
operations
  create : → queue
  isEmpty: queue → boolean
  add    : queue × Element → queue
  remove : queue ⇀ queue
  head   : queue ⇀ Element
pre-conditions
  remove(f,e)) iff !isEmpty(f)
  head(f)) iff !isEmpty(f)
axioms
  isEmpty(create) ≡ true // A1
  isEmpty(add(f,e) ≡ false // A2
  isEmpty(f) ⇒ head(add(f,e)) ≡ e // A3
  !isEmpty(f) ⇒ head(add(f,e)) ≡ head(f) // A4
  isEmpty(f) ⇒ remove(add(f,e)) ≡ create // A5
  !isEmpty(f) ⇒
    remove(add(f,e)) ≡ add(remove(f),e) // A6
```

As for the *Stack*, operations *remove* and *head* being partial, one needs pre-conditions.

A few notes about the axioms:

- A1: a newly created queue is empty
- A2: a queue to which we add an element is not empty any more
- A3: if we add an element to an empty queue, that element is the head of the queue

- A4: if we add an element to a non-empty queue, that does not change the head of the queue
- A5: if we add, then remove, an element to an empty queue, we go back to an empty queue
- A6: when we add, then remove, an element to a non-empty queue, we get the same queue as if we had first removed the head, then added the new element

3.3 The List ADT

Inside a list, each element is associated with its (unique) *position* (a strictly positive integer). Unlike the previous two ADT, any element of the list can be accessed, whatever its position is.

The minimum functionalities expected from a basic list are as follows:

- access the top element (first) of the list
- access the rest of the list (i.e. the list minus its first element)
- tell the length of the list
- access an element based on its position in the list
- add an element at a given position in the list
- remove an element at a given position in the list

As a first step, we just define the ADT with its constructors and access functions (*accessors*) to the first element and to the rest of the list (the other operations can then be expressed using those basic ones):

```
ADT List
Uses boolean, Element, Natural
operations
  create   : → list
  isEmpty: list → boolean
  rest   : list ⇀ list
  cons    : Element × list → list
  first: list ⇀ Element
```

```
pre-conditions
  first(l)) ⇔ !isEmpty(l)
  rest(l)) ⇔ !isEmpty(l)
axioms
  isEmpty(create) ≡ true // A1
  isEmpty(cons(e,l)) ≡ false // A2
  first(cons(e,l)) ≡ e // A3
  rest(cons(e,l)) ≡ l // A4
```

Partial operations *first* and *rest* need pre-conditions, as usual.

A few comments on these axioms:

- A1: A list just created is empty
- A2: as soon as we add an element to the list, it is not empty any more
- A3: We add an element to the head of the list, so after adding an element the (new) first element is the one we just added
- A4: after adding an element to a list, the rest of the new list is the old one (before the insertion)

3.4 Minimum List extension

Based on the functions described above, we can now define all the classical operations about lists. We get the following ADT:

```
extension ADT list
Uses Natural
operations
  length : list → Natural
  add  : list × Natural × Element ⇀ list
  remove  : list × Natural ⇀ list
  nth    : list × Natural ⇀ Element
pre-conditions
  remove(l,n)) iff 1 ≤ n ≤ length(l)
  nth(l,n)) iff 1 ≤ n ≤ length(l)
  add(l,n,e)) iff 1 ≤ n ≤ length(l) + 1
axioms
  length(create) ≡ 0 // A5
```

```
l!=create⇒length(l) ≡ length(rest(l)) + 1 // A6
nth(l,1) ≡ first(l) // A7
i>1⇒nth(l,i) ≡ nth(rest(l),i-1) // A8
add(l,1,e) ≡ cons(e,l) // A9
i>1⇒add(l,i,e) ≡
       cons(first(l),add(rest(l),i-1,e) // A10
remove(l,1) = rest(l) // A11
i>1⇒remove(l,i) ≡
       cons(first(l),remove(rest(l),i-1) // A12
```

Partial operations *remove, add* and *nth* are as usual completed with a pre-condition.

A few notes about axioms:

- A5: the length of a newly created list is 0 (it's empty).
- A6: When an element is added, the length is increased by 1
- A7: The element at position 1 is the first (!)
- A8: The element at position i in a list is the one at position $i-1$ in the rest of that list
- A9: Adding an element at position 1 is "consing" the element (head-addition)
- A10: Adding an element at position i in a list is adding it at position $i-1$ in the rest of that list
- A11: removing the first element of a list results in getting the rest of that list
- A12: Removing an element at position i in a list is removing it at position $i-1$ in the rest of that list

4 Binary trees

Binary trees are one of the simplest forms of tree structures. They are of great importance because they offer a rather simple way to obtain a good level of performance (in average) when searching, adding or removing information.