Lists

```
datatype List<T> = Nil | Cons(head: T, tail: List<T>)
function Length<T>(xs: List<T>): nat {
  match xs
  case Nil => 0
  case Cons(_, tail) => 1 + Length(tail)
}
function Length'<T>(xs: List<T>): nat {
  if xs == Nil then 0 else 1 + Length'(xs.tail)
}
lemma  LengthLength'<T>(xs: List<T>)
  ensures Length(xs) == Length'(xs)
{}
```

Concatenate at the end

```
function Snoc<T>(xs: List<T>, y: T): List<T> {
  match xs
  case Nil => Cons(y, Nil)
  case Cons(x, tail) => Cons(x, Snoc(tail, y))
}

lemma LengthSnoc<T>(xs: List<T>, x: T)
  ensures Length(Snoc(xs, x)) == Length(xs) + 1
{
}
```

Append– intrinsic definition

```
function Append<T>(xs: List<T>, ys: List<T>): List<T>
ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)
{ match xs
    case Nil => ys
    case Cons(x, tail) => Cons(x, Append(tail, ys))
}
```

If the length is not included in the definition of the function (extrinct) one could proof the following lemma

```
lemma LengthAppend<T>(xs: List<T>, ys: List<T>)
  ensures Length(Append(xs, ys)) == Length(xs) + Length(ys)
{
}
```

```
lemma {:induction false} AppendNil<T>(xs: List<T>)
  ensures Append(xs, Nil) == xs
{
  match xs
  case Nil =>
  case Cons(x, tail) =>
    calc {
      Append(xs, Nil);
    ==  // def. Append
      Cons(x, Append(tail, Nil));
    ==  { AppendNil(tail); }
      Cons(x, tail);
    ==
      xs;
    }
}
lemma AppendAssociative<T>(xs: List<T>, ys: List<T>, zs: List<T>)
  ensures Append(Append(xs, ys), zs) == Append(xs, Append(ys, zs))
 {}


function At<T>(xs: List<T>, i: nat): T
requires i < Length(xs)
{
if i == 0 then xs.head else At(xs.tail, i - 1)
}
lemma AtAppend<T>(xs: List<T>, ys: List<T>, i: nat)
  requires i < Length(Append(xs, ys))
  ensures At(Append(xs, ys), i)
       == if i < Length(xs) then
            At(xs, i)
          else
            At(ys, i - Length(xs))
{
}
```

List Sorting

```
ghost predicate Ordered(xs: List<int>) {
  match xs
  case Nil => true
  case Cons(x, Nil) => true
  case Cons(x, Cons(y, _)) => x <= y && Ordered(xs.tail)
}
lemma AllOrdered(xs: List<int>, i: nat, j: nat)
```

```
  requires Ordered(xs) && i <= j < Length(xs)
  ensures At(xs, i) <= At(xs, j)
{
  if i != 0 {
    AllOrdered(xs.tail, i - 1, j - 1);
  } else if i == j {
  } else {
    AllOrdered(xs.tail, 0, j - 1);
  }
}
```

Insertion Sort

```
function InsertionSort(xs: List<int>): List<int> {
  match xs
  case Nil => Nil
  case Cons(x, tail) => Insert(x, InsertionSort(tail))
}

function Insert(y: int, xs: List<int>): List<int> {
  match xs
  case Nil => Cons(y, Nil)
  case Cons(x, tail) =>
    if y < x then Cons(y, xs) else Cons(x, Insert(y, tail))
}
```

```
lemma InsertionSortOrdered(xs: List<int>)
  ensures Ordered(InsertionSort(xs))
{
  match xs
  case Nil =>
  case Cons(x, tail) =>
    InsertOrdered(x, InsertionSort(tail));
}

lemma InsertOrdered(y: int, xs: List<int>)
  requires Ordered(xs)
  ensures Ordered(Insert(y, xs))
{
}
Missing: the elements are the same (Exercise)
```

Immutable Queue

A queue is a list of elements with three operations

- Initialization: creates an empty queue
- Enqueue: appends an element to the queue (at the end)
- Dequeue: removes and returns the head element of the queue

```
module ImmutableQueue {
  type Queue<A>
  function Empty(): Queue
  function Enqueue<A>(q: Queue, a: A): Queue
  function Dequeue<A>(q: Queue): (A, Queue)
    requires q != Empty<A>()
    }
```

Modules

- a `module` allows to group a set of types and methods/functions
- allows to have `namespaces`
- can be used with `import` in another
- can `export` (some of its functions/methods)
  - `provides`: signatures (type, function names,specifications)
  - `reveals`: body (fields, expressions, statements inside curly brackets)

Abstraction

- we want to proof some properties of a Queue without bother how it is implemented
- What we know is that a queue is a list of elements
- We consider an *abstraction* of the implementation (abstraction function)

  ```
  ghost function Elements(q: Queue): LL.List
  ```

- we can express the queue operations in terms of list operations
- the specifications will be extrinsic to the functions

```
 module ListLibrary {
  datatype List<T> = Nil | Cons(head: T, tail: List<T>)

  function Snoc<T>(xs: List<T>, y: T): List<T> {
    match xs
    case Nil => Cons(y, Nil)
    case Cons(x, tail) => Cons(x, Snoc(tail, y))
  }
}

module ImmutableQueue {
import LL = ListLibrary
  type Queue<A>
  function Empty<A>(): Queue
  function Enqueue<A>(q: Queue, a: A): Queue
  function Dequeue<A>(q: Queue): (A, Queue)
    requires q != Empty<A>()
  ghost function Elements(q: Queue): LL.List
  lemma EmptyCorrect<A>()
    ensures Elements(Empty<A>()) == LL.Nil
  lemma EnqueueCorrect<A>(q: Queue, x: A)
    ensures Elements(Enqueue(q, x)) ==
                LL.Snoc(Elements(q), x)
  lemma DequeueCorrect(q: Queue)
    requires q != Empty()
    ensures var (a, q') := Dequeue(q);
      LL.Cons(a, Elements(q')) == Elements(q)
```

Exports

```
module ImmutableQueue {
  import LL = ListLibrary

  export
    provides Queue, Empty, Enqueue, Dequeue
    provides LL, Elements
    provides EmptyCorrect, EnqueueCorrect, DequeueCorrect

  type Queue<A> ...
  }
```

Client

```
module QueueClient {
  import IQ = ImmutableQueue
  method Client() {
```

```
    IQ.EmptyCorrect<int>();   var q := IQ.Empty();
    IQ.EnqueueCorrect(q, 20); q := IQ.Enqueue(q, 20);
    IQ.DequeueCorrect(q);     var (a, q') := IQ.Dequeue(q);
    assert a==20;
    assert q'== IQ.Empty();
  }
}
```

Last assert fails, as there may exist several values of type `Queue` that represent an empty list.

```
lemma EmptyUnique(q: Queue)
    ensures Elements(q) == LL.Nil ==> q == Empty()
    ...  Client ...
    IQ.EmptyUnique(q');
    assert q' == IQ.Empty();
```

Queue implementation

- Implementing with a list is not efficient (due to enqueue)
- Better is to use two lists: `front` and `rear`
- `front`: has the head of the queue
- `rear`: rest of the elements in reverse order
- Dequeue and Enqueue are constant time
- when `Dequeue` finds the front empty it copies and reverses `rear` to `front`

```
datatype Queue<A> = FQ(front: LL.List<A>, rear: LL.List<A>)

   ghost function Elements(q: Queue): LL.List {
    LL.Append(q.front, LL.Reverse(q.rear))
  }


function Empty(): Queue {
    FQ(LL.Nil, LL.Nil) }
  predicate IsEmpty(q: Queue)
    ensures IsEmpty(q) <==> q == Empty()
  { q == FQ(LL.Nil, LL.Nil)}
  function Enqueue<A>(q: Queue, x: A): Queue {
    FQ(q.front, LL.Cons(x, q.rear))
    }
  function Dequeue<A>(q: Queue): (A, Queue)
    requires  ! IsEmpty(q)
```

```
  {
    match q.front
    case Cons(x, front') =>
      (x, FQ(front', q.rear))
    case Nil =>
      var front := LL.Reverse(q.rear);
      (front.head, FQ(front.tail, LL.Nil))
  }
```

Data structure invariants

- In general data structures are not defined just using datatypes but some other invariants must be considered
- E.g. a binary search tree needs that values are sorted
- We are going to consider a priority-queue: a queue that gives access only to a minimal element.
- Operations are: creating, checking for Empty and
  - insertion of an element
  - removal of the minimal element
- we are going to consider integer elements

```
module PriorityQueue {
  type PQueue

  function Empty(): PQueue
  predicate IsEmpty(pq: PQueue)
  function Insert(pq: PQueue, y: int): PQueue
  function RemoveMin(pq: PQueue): (int, PQueue)
    requires !IsEmpty(pq)
}
```

Abstraction of the elements

```
ghost function Elements(pq: PQueue): multiset<int>
  lemma EmptyCorrect()
    ensures Elements(Empty()) == multiset{}
  lemma IsEmptyCorrect(pq: PQueue)
    ensures IsEmpty(pq) <==> Elements(pq) == multiset{}
  lemma InsertCorrect(pq: PQueue, y: int)
    ensures Elements(Insert(pq, y))
```

```
        == Elements(pq) + multiset{y}
  lemma RemoveMinCorrect(pq: PQueue)
    requires !IsEmpty(pq)
    ensures var (y, pq') := RemoveMin(pq);
      IsMin(y, Elements(pq)) &&
      Elements(pq') + multiset{y} == Elements(pq)
  ghost predicate IsMin(y: int, s: multiset<int>) {
    y in s && forall x :: x in s ==> y <= x }
```

Data Structure Implementation

- use of a binary heap, i.e., a binary tree where the value of a node is minimal among all values stored in the subtree rooted at that node (heap property)
- `Insert` and `RemoveMin` operate in time $O(\log n)$ if the tree is balanced
- A tree is balanced if every left tree has roughly the same size as the right tree
- For that one uses a `Braun tree`
- left subtree can have one more element than the right tree
- heap property and balancedness are the *data structure invariants*
- A Braun tree ensures that an insertion is always made in the right tree (as that one is never larger than the left) but the resulting tree swaps left and right trees.

```
type PQueue = BraunTree
datatype BraunTree = Leaf
  | Node(x: int, left: BraunTree, right: BraunTree)
ghost predicate Valid(pq: PQueue)
lemma EmptyCorrect()
    ensures var pq := Empty(); Valid(pq) && Elements(pq) == multiset{}
lemma IsEmptyCorrect(pq: PQueue)
    requires Valid(pq)
    ensures IsEmpty(pq) <==> Elements(pq) == multiset{}
lemma InsertCorrect(pq: PQueue, y: int)
    requires Valid(pq)
    ensures var pq' := Insert(pq, y);
      Valid(pq') && Elements(pq') == Elements(pq) + multiset{y}
lemma RemoveMinCorrect(pq: PQueue)
    requires Valid(pq) && !IsEmpty(pq)
    ensures var (y, pq') := RemoveMin(pq);
      Valid(pq') &&  IsMin(y, Elements(pq)) &&
      Elements(pq') + multiset{y} == Elements(pq)
```

Valid()

```
ghost predicate Valid(pq: PQueue) {
    IsBinaryHeap(pq) && IsBalanced(pq) }
ghost predicate IsBinaryHeap(pq: PQueue) {
    match pq
    case Leaf => true
    case Node(x, left, right) =>
      IsBinaryHeap(left) && IsBinaryHeap(right) &&
      (left == Leaf || x <= left.x) &&
      (right == Leaf || x <= right.x) }
ghost predicate IsBalanced(pq: PQueue) {
    match pq
    case Leaf => true
    case Node(_, left, right) =>
      IsBalanced(left) && IsBalanced(right) &&
      var L, R := |Elements(left)|, |Elements(right)|;
      L == R || L == R + 1}
```

Elements

```
ghost function Elements(pq: PQueue): multiset<int> {
    match pq
    case Leaf => multiset{}
    case Node(x, left, right) =>
      multiset{x} + Elements(left) + Elements(right)
  }
```

Empty

```
function Empty(): PQueue {
    Leaf
  }
predicate IsEmpty(pq: PQueue) {
    pq == Leaf }
  lemma EmptyCorrect()
    ensures var pq := Empty();
      Valid(pq) &&
      Elements(pq) == multiset{}
  {}

  lemma IsEmptyCorrect(pq: PQueue)
    requires Valid(pq)
    ensures IsEmpty(pq) <==> Elements(pq) == multiset{}
  {}
```

Insert

```
function Insert(pq: PQueue, y: int): PQueue {
    match pq
    case Leaf => Node(y, Leaf, Leaf)
    case Node(x, left, right) =>
      if y < x then
        Node(y, Insert(right, x), left)
      else
        Node(x, Insert(right, y), left)
  }
lemma InsertCorrect(pq: PQueue, y: int)
    requires Valid(pq)
    ensures var pq' := Insert(pq, y);
      Valid(pq') &&
      Elements(pq') == Elements(pq) + multiset{y}
  {}
```

Removal

- `RemoveMin` does not apply to a `Leaf`
- it is applied to a `Node`
- the element of that node is returned
- the new minimal must be the root of the new queue, returned by `DeleteMin`

```
function RemoveMin(pq: PQueue): (int, PQueue)
   requires !IsEmpty(pq)
 {   (pq.x, DeleteMin(pq)) }
```

- the correctness of this functions is

```
lemma DeleteMinCorrect(pq: PQueue)
requires Valid(pq) && pq != Leaf
ensures var pq' := DeleteMin(pq);
Valid(pq') && Elements(pq') + multiset{pq.x} == Elements(pq)
```

- and then

```
lemma RemoveMinCorrect(pq: PQueue)
   requires Valid(pq) && !IsEmpty(pq)
   ensures var (y, pq') := RemoveMin(pq);
     Valid(pq') &&  IsMin(y, Elements(pq)) &&
     Elements(pq') + multiset{y} == Elements(pq)
 {    DeleteMinCorrect(pq);  }
```

Delete Min

```
   function DeleteMin(pq: PQueue): PQueue
    requires !IsEmpty(pq)
  {
    if pq.left == Leaf || pq.right == Leaf then
      pq.left
    else if pq.left.x <= pq.right.x then
      Node(pq.left.x, pq.right, DeleteMin(pq.left))
    else
      Node(pq.right.x, ReplaceRoot(pq.right, pq.left.x),
           DeleteMin(pq.left))
  }
```

where `ReplaceRoot(pq,y)` removes the minimum element of `pq` and inserts `y`. This ensures the preservation of the invariants.

DeleteMinCorrect

```
 lemma DeleteMinCorrect(pq: PQueue)
    requires Valid(pq) && pq != Leaf
    ensures var pq' := DeleteMin(pq);
      Valid(pq') &&
      Elements(pq') + multiset{pq.x} == Elements(pq)
  {
    if pq.left == Leaf || pq.right == Leaf {
    } else if pq.left.x <= pq.right.x {
      DeleteMinCorrect(pq.left);
    } else {
      var left, right :=
        ReplaceRoot(pq.right, pq.left.x), DeleteMin(pq.left);
      var pq' := Node(pq.right.x, left, right);
      assert pq' == DeleteMin(pq);
      ReplaceRootCorrect(pq.right, pq.left.x);
      DeleteMinCorrect(pq.left);
```

DeleteMinCorrect

```
calc {
        Elements(pq') + multiset{pq.x};
      ==  // def. Elements, since pq' is a Node
        multiset{pq.right.x} + Elements(left) +
        Elements(right) + multiset{pq.x};
      ==
```

```
      Elements(left) + multiset{pq.right.x} +
      Elements(right) + multiset{pq.x};
    ==  { assert Elements(left) + multiset{pq.right.x}
            == Elements(pq.right) + multiset{pq.left.x}; }
      Elements(pq.right) + multiset{pq.left.x} +
      Elements(right) + multiset{pq.x};
    ==
      Elements(right) + multiset{pq.left.x} +
      Elements(pq.right) + multiset{pq.x};
    ==  { assert Elements(right) + multiset{pq.left.x}
            == Elements(pq.left); }
      Elements(pq.left) + Elements(pq.right) +
      multiset{pq.x};
    ==
      multiset{pq.x} + Elements(pq.left) +
      Elements(pq.right);
    ==  // def. Elements, since pq is a Node
      Elements(pq);
    }
  }
}
```

Replace root

```
function ReplaceRoot(pq: PQueue, y: int): PQueue
    requires !IsEmpty(pq)
  {
    if pq.left == Leaf ||
       (y <= pq.left.x && (pq.right == Leaf || y <= pq.right.x))
    then
      Node(y, pq.left, pq.right)
    else if pq.right == Leaf then
      Node(pq.left.x, Node(y, Leaf, Leaf), Leaf)
    else if pq.left.x < pq.right.x then
      Node(pq.left.x, ReplaceRoot(pq.left, y), pq.right)
    else
      Node(pq.right.x, pq.left, ReplaceRoot(pq.right, y))
  }
```

Replace root Correct

```
lemma ReplaceRootCorrect(pq: PQueue, y: int)
    requires Valid(pq) && !IsEmpty(pq)
    ensures var pq' := ReplaceRoot(pq, y);
      Valid(pq') &&
```

```
    Elements(pq) + multiset{y} == Elements(pq') + multiset{pq.x} &&
    |Elements(pq')| == |Elements(pq)|
{
  if pq.left == Leaf ||
     (y <= pq.left.x && (pq.right == Leaf || y <= pq.right.x))
  {
  } else if pq.right == Leaf {
  } else if pq.left.x < pq.right.x {
    var left := ReplaceRoot(pq.left, y);
    var pq' := Node(pq.left.x, left, pq.right);
    assert pq' == ReplaceRoot(pq, y);
    ReplaceRootCorrect(pq.left, y);


calc { Elements(pq) + multiset{y};
    ==  // def. Elements, since pq is a Node
      multiset{pq.x} + Elements(pq.left) +
      Elements(pq.right) + multiset{y};
    ==
      Elements(pq.left) + multiset{y} +
      Elements(pq.right) + multiset{pq.x};
    ==  { assert Elements(pq.left) + multiset{y}
              == Elements(left) + multiset{pq.left.x}; } // I.H.
      multiset{pq.left.x} + Elements(left) +
      Elements(pq.right) + multiset{pq.x};
    ==  // def. Elements, since pq' is a Node
      Elements(pq') + multiset{pq.x};
    }  }


else { var right := ReplaceRoot(pq.right, y);
    var pq' := Node(pq.right.x, pq.left, right);
    assert pq' == ReplaceRoot(pq, y);
    ReplaceRootCorrect(pq.right, y);
calc {
      Elements(pq) + multiset{y};
    ==  // def. Elements, since pq is a Node
      multiset{pq.x} + Elements(pq.left) +
      Elements(pq.right) + multiset{y};
    ==
      Elements(pq.right) + multiset{y} +
      Elements(pq.left) + multiset{pq.x};
    ==  { assert Elements(pq.right) + multiset{y}
              == Elements(right) + multiset{pq.right.x}; }
      multiset{pq.right.x} + Elements(pq.left) +
      Elements(right) + multiset{pq.x};
```

```
      ==  // def. Elements, since pq' is a Node
        Elements(pq') + multiset{pq.x};
      }
}}
```