TÖL212M Rökstudd Forritun - Hópverkefni 10

Andri Fannar Kristjánsson

25. mars 2025

Hópverkefni 10

1

Sækið skrána `H10-skeleton.java` í Canvas og vistið hana sem `H10.java`. Klárið að útfæra klasann í skránni.

1.1 Svar:

Hér fyrir neðan má sjá kóðann þar sem föllin hafa verið forrituð. Einnig er hægt að sjá skrána hér: https://tinyurl.com/4a3ymtff. Misvísandi skilaboð komu frá Dafny varðandi hvort þessi lausn virkar eða ekki, en mér sýndist það vera timeout vandamál frekar en vandamál með útfærsluna.

```
// Author of question:  Snorri Agnarsson, snorri@hi.is

// Author of solution:      Andri Fannar Kristjánsson, afk6@hi.is
// Permalink of solution:   https://tinyurl.com/4a3ymtff

// Finish programming the class QueueCycleChain
// by programming the bodies of the operations
// IsEmpty, Put and Get, as well as the
// constructor for the class.
// Everything needed is in this file and you
// do not need to call any lemmas or write
// any asserts. Note that it is unlikely that
// you can solve this problem on the tio.run
// web page.

class QueueCycleChain<T> extends Queue<T>
  {
  var last: Link?<T>
  ghost var cycle: seq<Link<T>>

  ghost predicate Valid()
    reads this, Repr
  {
    (last != null ==> last in Repr) &&
    (forall z | z in cycle :: z in Repr) &&
    (forall z | z in Repr :: z in cycle) &&
    (ghostseq == ValueSeq(cycle)) &&
    (last == null ==> cycle == []) &&
    (last != null ==> last.ValidCycle(cycle)) &&
    (|cycle| == |ghostseq|) &&
    (forall i | 0 <= i < |cycle| :: cycle[i].head == ghostseq[i])
  }

  constructor()
    ensures Valid()
```

```
    ensures Repr == {}
    ensures ghostseq == []
  {
    // Give the instance variables last, Rept, ghostseq and
    // cycle correct values considering the data invariant
    // and the postcondition.
    // To satisfy the postcondition and the data invariants, we
    // simply initialize last to null and Repr, ghostseq and cycle
    // to empty sets / seq.
    last := null;
    Repr := {};
    ghostseq := [];
    cycle := [];
    new;
    assert  // This assert is the same as the data invariant
      // and is unnecessary if the constructor is
      // correctly programmed, but if not it is
      // helpful in identifying errors.
      (last != null ==> last in Repr) &&
      (forall z | z in cycle :: z in Repr) &&
      (forall z | z in Repr :: z in cycle) &&
      (ghostseq == ValueSeq(cycle)) &&
      (last == null ==> cycle == []) &&
      (last != null ==> last.ValidCycle(cycle)) &&
      |cycle| == |ghostseq| &&
      (forall i | 0 <= i < |cycle| :: cycle[i].head == ghostseq[i]);
  }

  predicate IsEmpty()
    reads this, Repr
    requires Valid()
    ensures IsEmpty() <==> ghostseq==[]
  {
    // Here we simply check if the last is null,
    // if it is then the queue is empty.
    last == null
  }

  method Put( x: T )
    modifies this, Repr
    requires Valid()
    ensures Valid()
    ensures fresh(Repr-old(Repr))
    ensures ghostseq == old(ghostseq)+[x]
  {
    // We create a new link with the value x and add it to the end
    // of the cycle. This works both for the case when the queue is
    // empty and when it is not, as doing this on the empty queue
    // is equivalent to `new Link<T>(x, null, [], []);`
    var newlink := new Link<T>(x,last,cycle,ghostseq);
    last := newlink;
    ghostseq := ghostseq+[x];
    cycle := cycle+[newlink];
    Repr := Repr + {newlink};
  }

  method Get() returns ( x: T )
    modifies this, Repr
```

```
    requires Valid()
    requires ghostseq != []
    ensures Valid()
    ensures Repr < old(Repr)
    ensures ghostseq == old(ghostseq[1..])
    ensures x == old(ghostseq[0])
  {
    // We remove the first link from the cycle and update the last
    // link.
    var removedLink := last.tail;
    // We call the RemoveFirst method to remove the first link from
    // the cycle.
    var newlast, newcycle, newvals := RemoveFirst(last,cycle,ghostseq);
    // We update the last link and the cycle, as well as the ghost
    // variables.
    last := newlast;
    cycle := newcycle;
    ghostseq := newvals;
    Repr := Repr - {removedLink};
    x := removedLink.head;
  }
}

////////////////////////////////////////////////////////////////
// Here the mutable part of the file ends.
// Do not change the program text below this.
////////////////////////////////////////////////////////////////

// This is the fundamental definition of the behaviour of a queue.
trait Queue<T>
  {
  ghost var ghostseq: seq<T>
  ghost var Repr: set<object>

  ghost predicate Valid()
    reads this, Repr

  predicate IsEmpty()
    reads this, Repr
    requires Valid()
    ensures IsEmpty() <==> ghostseq==[]

  method Put( x: T )
    modifies this, Repr
    requires Valid()
    ensures Valid() && fresh(Repr-old(Repr))
    ensures ghostseq == old(ghostseq)+[x]

  method Get() returns ( x: T )
    modifies this, Repr
    requires Valid()
    requires ghostseq != []
    ensures Valid() && fresh(Repr-old(Repr))
    ensures ghostseq == old(ghostseq[1..])
    ensures x == old(ghostseq[0])
}

// Here is the definition of mutable links
```

```
// that are used in circular chains.
class Link<T>
  {
  var head: T
  var tail: Link<T>

  predicate ValidCycle( cycle: seq<Link<T>> )
    reads this, cycle
  {
    |cycle| > 0 &&
    this == cycle[|cycle|-1] &&
    tail == cycle[0] &&
    tail.ValidSequence(cycle) &&
    forall p,q | 0 <= p < q < |cycle| ::
      cycle[p] != cycle[q]
  }

  predicate ValidSequence( sequence: seq<Link<T>> )
    reads this, sequence
  {
    |sequence| > 0 &&
    this == sequence[0] &&
    forall i | 0 <= i < |sequence|-1 ::
      sequence[i].tail == sequence[i+1]
  }

  constructor( h: T, x: Link?<T>, ghost cycle: seq<Link<T>>, ghost values: seq<T> )
    modifies if cycle != [] then {cycle[|cycle|-1]} else {}
    requires (x == null && cycle == []) || (x != null && x.ValidCycle(cycle))
    requires values == ValueSeq(cycle)
    requires forall i | 0 <= i < |cycle| :: cycle[i].head == values[i]
    ensures head == h
    ensures tail == if x == null then this else cycle[0]
    ensures fresh(this)
    ensures forall i | 0 <= i < |cycle| :: cycle[i].head == values[i]
    ensures forall i | 0 <= i < |cycle| :: cycle[i].head == old(cycle[i].head)
    ensures forall z | z in cycle :: z.head == old(z.head)
    ensures ValueSeq(cycle) == values
    ensures ValidCycle(cycle+[this])
    ensures ValueSeq(cycle+[this]) == values+[h]
  {
    head := h;
    tail := this;
    new;
    if x != null
    {
      tail := x.tail;
      x.tail := this;
      HeadsEqual(cycle,values);
      AppendLink(cycle,this);
    }
  }
}

method RemoveFirst<T>   ( last: Link<T>
                        , ghost cycle: seq<Link<T>>
                        , ghost vals: seq<T>
)
```

```
  returns ( newlast: Link?<T>
          , ghost newcycle: seq<Link<T>>
          , ghost newvals: seq<T>
  )
  modifies last
  requires last.ValidCycle(cycle)
  requires ValueSeq(cycle) == vals
  requires |vals| == |cycle|
  requires |vals| > 0
  ensures last.head == old(last.head)
  ensures |vals| == 1 ==>
            newlast == null &&
            newcycle == [] &&
            newvals == []
  ensures |vals| > 1 ==>
            newlast == last &&
            newcycle == cycle[1..] &&
            newlast.ValidCycle(cycle[1..]) &&
            newvals == vals[1..] &&
            ValueSeq(newcycle) == newvals
{
  if last.tail == last
  {
    newlast := null;
    newcycle := [];
    newvals := [];
    return;
  }
  ValueSeqHeads(cycle,vals);
  newlast := last;
  newcycle := cycle[1..];
  newvals := vals[1..];
  newlast.tail := newlast.tail.tail;
  HeadsEqual(cycle[1..],vals[1..]);
}

function ValueSeq<T>( x: seq<Link<T>> ): seq<T>
  reads x
  ensures |x| == |ValueSeq(x)|
{
  if x == [] then
    []
  else
    [x[0].head]+ValueSeq(x[1..])
}

lemma ValueSeqHeads<T>( x: seq<Link<T>>, v: seq<T> )
  requires v == ValueSeq(x)
  ensures forall i | 0 <= i < |x| :: x[i].head == v[i]
{
  if |x| == 0 { return; }
  ValueSeqHeads(x[1..],v[1..]);
}

lemma AppendLink<T>( x: seq<Link<T>>, z: Link<T> )
  ensures ValueSeq(x+[z]) == ValueSeq(x)+[z.head]
{
  if x == [] { return; }
```

```
  AppendLink(x[1..],z);
  calc ==
  {
    x+[z];
    (x[..1]+x[1..])+[z];
    x[..1]+(x[1..]+[z]);
  }
}

lemma HeadsEqual<T>( y: seq<Link<T>>, v: seq<T> )
  requires |y| == |v|
  requires forall i | 0 <= i < |y| :: v[i] == y[i].head
  ensures v == ValueSeq(y)
{
  if |y| == 0 { return; }
  HeadsEqual(y[1..],v[1..]);
}

method Factory() returns ( q: Queue<int> )
  ensures fresh(q)
  ensures fresh(q.Repr)
  ensures q.Valid()
  ensures q.IsEmpty()
{
  q := new QueueCycleChain<int>();
}

method Main()
{
  var q1 := Factory();
  var q2 := Factory();
  var q3 := Factory();
  var q4 := Factory();
  q1.Put(1);
  q2.Put(1);
  q3.Put(1);
  q4.Put(1);
  q1.Put(2);
  assert q1.ghostseq == [1,2];
  q2.Put(2);
  q3.Put(2);
  q4.Put(2);
  var x;
  x := q1.Get(); print x; print " ";
  assert x == 1;
  x := q1.Get(); print x; print " ";
  assert x == 2;
  assert q1.ghostseq == [];
  x := q2.Get(); print x; print " ";
  x := q2.Get(); print x; print " ";
  x := q3.Get(); print x; print " ";
  x := q3.Get(); print x; print " ";
  x := q4.Get(); print x; print " ";
  x := q4.Get(); print x; print " ";
  assert q1.IsEmpty();
  assert q2.IsEmpty();
  assert q3.IsEmpty();
  assert q4.IsEmpty();
```

```
}
```