CSSE3100 Crib Sheet

Collection Types

Arrays

Arrays are a mutable collection of elements stored on the heap.

For an array, a, to be modified by a method, it must include `modifies a` in the specification.

Type	`array<T>`
Creation	`var a := new T[length];`
Accessing	`var value := a[i];`
Assigning	`a[i] := value;`
Alias	`var b := a;`
Length	`var l := a.Length;`
Slicing	`var s: seq<T> := a[start...end];`

Multi-dimensional Arrays

Type	`array<array<...<array<T>>>>`
Creation	`var a := new T[l1, l2, ..., lN];`
Accessing	`var value := a[i1, i2, ..., iN];`
Asigning	`a[i1, i2, ..., iN] := value;`
Alias	`var b := a;`
Length	`(l1, l2, ..., lN) := (a.Length1, a.Length2, ... a.LengthN);`

Sequences

Sequences are used to represent an ordered list. They are immutable.

Type	`seq<T>`
Creation	`var s := [x1, x2, ..., xN];`
Accessing	`var value := s[i];`
Length	`var l := |s|;`
Slicing	`var t := a[start...end];`
Appending	`var u := s + t;`
Contains	`value in s;`
Excludes	`value !in s;`

Sets

Sets are used to represent an orderless collection of elements, without repetition. Sets are immutable.

Type	`set<T>`
Creation	`var s := {x1, x2, ..., xN};`
Equality	`{x1, x2} == {x2, x1} == {x1, x1, x2, x2};`
Subset	`s <= t`
Proper Subset	`s < t`
Union	`var u := s + t;`
Intersection	`var u := s * t;`
Difference	`var u := s - t;`
Contains	`elem in s;`
Excludes	`elem !in s;`

Multisets

Multisets are used to represent an orderless collection of elements, with repetition. Multisets are immutable.

Type	`multiset<T>`
Creation	`var s := multiset{x1, ..., xN};`
From seq	`var s := multiset([x1, ..., xN]);`
From set	`var s := multiset({x1, ..., xN});`
Equality	`multiset{x1, x2} == multiset{x2, x1} != multiset{x1, x1, x2, x2};`
Union	`var u := s + t;`
Difference	`var u := s - t;`
Contains	`elem in s;`
Excludes	`elem !in s;`
Disjoint	`elem !! s;`

Specification Keywords

Requires

A requires clause stipulates a condition `P` which must be true upon entry to the method.

Ensures

A requires clause stipulates a condition `R` which must be true when exiting the method.

Modifies

A modifies clause is required if a method changes a value on the heap (i.e. a value in an array is changed).

Reads

A reads clause is required if a method reads a value on the heap (i.e. a value in an array is read).

Invariant

An invariant clause stipulates a condition `I` which must be true at the beginning and end of a loop.

Decreases

A decreases clause indicates a value `D` which decreases after every iteration of a loop.

Forall

A forall clause is used to stipulate that a condition `Q` must hold forall values of a given variable. For example, `forall i :: P[i] ==> Q[i]` requires `Q[i]` to hold for all values of `i` where `P[i]` holds.

Exists

An exists clause is used to stipulate that a condition `Q` must hold for at least one value of a given variable. For example, `exists i :: P[i] ==> Q[i]` requires `Q[i]` to hold for at least one value of `i` where `P[i]` holds.

Fresh

Fresh is used to indicate that a value stored on the heap must be brand new with no modifications. For example, `fresh(x)` requires x to be a brand new value on the heap.

Old

Old is used to reference the value on the heap before the method began. For example, `old(a[i])` refers to the element at index `i` at the beginning of the method.

Question 1

Predicate Logic

$A \wedge (A \vee B) \equiv A \equiv A \vee (A \wedge B)$ (A.6)
$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$ (A.7)
$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$ (A.8)
$\neg(A \wedge B) \equiv \neg A \vee \neg B$ (A.18)
$\neg(A \vee B) \equiv \neg A \wedge \neg B$ (A.19)
$A \vee (\neg A \wedge B) \equiv A \vee B$ (A.20)
$A \wedge (\neg A \vee B) \equiv A \wedge B$ (A.21)
$A \Rightarrow B \equiv \neg A \vee B$ (A.22)
$A \Rightarrow B \equiv \neg(A \wedge \neg B)$ (A.24)
$\neg(A \Rightarrow B) \equiv A \wedge \neg B$ (A.25)
$A \Rightarrow B \equiv \neg B \Rightarrow \neg A$ (A.26)
$C \Rightarrow (A \wedge B) \equiv (C \Rightarrow A) \wedge (C \Rightarrow B)$ (A.33)
$(A \vee B) \Rightarrow C \equiv (A \Rightarrow C) \wedge (B \Rightarrow C)$ (A.34)
$C \Rightarrow (A \vee B) \equiv (C \Rightarrow A) \vee (C \Rightarrow B)$ (A.35)
$(A \wedge B) \Rightarrow C \equiv (A \Rightarrow C) \vee (B \Rightarrow C)$ (A.36)
$A \Rightarrow (B \Rightarrow C) \equiv (A \wedge B) \Rightarrow C \equiv B \Rightarrow (A \Rightarrow C)$ (A.37)
$(A \Rightarrow B) \wedge (\neg A \Rightarrow C) \equiv (A \wedge B) \vee (\neg A \wedge C)$ (A.38)
$(\forall x \text{ s.t. } x = E \Rightarrow A) \equiv A[x \backslash E] \equiv (\exists x \text{ s.t. } x = E \wedge A)$ (A.56)
$\forall x :: A \wedge B = (\forall x :: A) \wedge (\forall x :: B)$ (A.65)
$\forall x :: A = A$ provided x not free in A (A.74)

Weakest Precondition

Assignment

`wp(x := E, Q) = Q[x\E]`

Simultaneous Assignment

```
wp(x1, x2, ..., xN := E1, E2, ..., EN, Q)
        = Q[x1, x2, ..., xN\E1, E2, ... EN]
```

Variable Introduction

`wp(var x, Q) = forall x :: Q`

Can be ignored when we have `var x := E;` or `var x; x := E;`

Condition

```
wp(if B { S } else { T }, Q) =
        (B ==> wp(S, Q)) && (!B ==> wp(T, Q))
```

Loops

```
{J}
while B
        invariant J
{
        {B && J}
        ...
        {J}
}
{J && !B}
```

E.g.

```
{y >= 4 && z >= x}
while z < 0
        invariant y >= 4 && z >= x
{
        {z < 0 && y >= 4 && z >= x}
        {y >= 4 && z + y >= x}
        z := z + y;
        {y >= 4 && z >= x}
}
{z >= 0 && y >= 4 && z >= x}
```

Loops Proving Decreases

```
{J}
while B
        invariant J
        decreases D
{
        {B && J}
        ghost var d := D;
        ...
        {J && d ≻ D}
}
{J && !B}
```

Methods

For a generic method M,

```
method M(x: int, a: array<int>) returns (y: int)
        requires P
        modifies a
        ensures R
```

The WP rule is:

```
wp(t := M(E, b), Q)  =
    P[x, a\E, b] &&
    forall y', b' :: b'.Length = b. Length &&
        R[x, y, a, old(a[i])\E, y', b', b[i]]
                ==> Q[t, b\y', b']
```

For example.

Given:

```
method Triple(x: int) returns (y: int)
requires x >= 0
ensures y == 3*x {}
```

```
{ u == 15}
{ u + 3 >= 0 &&
        3*(u + 3) == 54 } (A.56)
{ u + 3 >= 0 &&
        forall y' :: y' == 3*(u + 3)
                ==> y' == 54 }
t := Triple(u + 3);
{ t == 54 }
```

```
function SeqSum(s: seq<int>, lo: int, hi: int)
                                        : int
requires 0 <= lo <= hi <= |s|
decreases hi - lo
{
        if lo == hi then 0 else s[lo] +
            SeqSum(s, lo + 1, hi)
}
```

Lemmas

For a call to lemma `M`, the WP rule is:

`wp(M(E), Q)  =  P[x\E] && (R[x\E] ==> Q)`

where P is the requires class of the lemma and R is the ensures clause.

Old

In a WP proof, `old(E)` can be replaced with `E` if there is no modifications to E above the current position in the method.

Question 2

Loop Design Techniques

Look in the postcondition.

For a postcondition A && B, choose the invariant to be A and the guard to be !B.

```
method SquareRoot(N: nat) returns (r: nat)
ensures r*r <= N && N < (r + 1)*(r + 1)
    { { 0 <= N }
    { 0*0 <= N}
    r := 0;
    { r*r <= N }
    while (r + 1)*(r + 1) <= N
    invariant r*r <= N
    {
        { (r + 1)*(r + 1) <= N
                && r*r <= N } (strengthen)
        { (r + 1)*(r + 1) <= N }
        r := r + 1;
        { r*r <= N }
    }
}
```

Programming by wishing

If a problem can be made simpler by having a precomputed quantity Q, then introduce a new variable q with the intention of establishing and maintaining the invariant q == Q

```
method SquareRoot(N: nat) returns (r: nat)
ensures r*r <= N < (r + 1)*(r + 1)
{
    r := 0;
    var s := 1;
    while s <= N
    invariant r*r <= N
    invariant s == (r + 1)*(r + 1)
    {
        s := s + 2*r + 3;
        r := r + 1;
    }
}
```

Replace a constant by a variable

For a loop to establish a condition P(C), where C is an expression that is held constant throughout the loop, use a variable k that the loop changes until it equals C, and make P(k) a loop invariant.
For example, Min method (Week 4) had postcondition

```
ensures forall i :: 0 <= i < a.Length
            ==> m <= a[i]
```

and invariant

```
    invariant forall i :: 0 <= i < n
            ==> m <= a[i]
```

What's yet to be done

. If you're trying to solve a problem of the form p == F(n), replacement of a constant by a variable results in a what-has-been-done invariant

```
    invariant p == F(i)
```

Alternatively, you may use a what's-yet-to-be-done invariant

```
    invariant p @ F(n - i) == F(n)
```

where @ is some kind of combination operation.

Use the postcondition

To establish a postcondition Q, make Q a loop invariant.
For the Min example, to ensure the postcondiVon

```
ensures exists i :: 0 <= i < a.Length &&
                                    m == a[i]
```

we used the invariant

```
invariant exists i :: 0 <= i < a.Length &&
                                    m == a[i]
```

Question 3

Termination Metrics

Any set of values which have a *well-founded* order can be used as a termination metric.
An order $\succ$ is well-founded when

- $\succ$ is irreflexive: a $\succ$ a never holds
- $\succ$ is transitive:
  a $\succ$ b && b $\succ$ c $\implies$ a $\succ$ c
- there is no infinite descending chain
  $\mathrm{a}_1 \succ \mathrm{a}_2 \succ \mathrm{a}_3 \succ \ldots$

We write X decreases to x as $X \succ x$.
For integers, $X \succ x$ when $X > x$ && $X >= 0$.
For booleans, $X \succ x$ when X && $!x$.
A termination metric for a recursive function is a metric that can be proven to decrease every iteration.
E.g. for the function;

```
    function F(x: int): int
    {
        if x < 10 then x else F(x { 1)
    }
```

the termination metric would be x since $x \succ x - 1$.

Lexicographic tuples

A lexicographic order is a component-wise comparison where earlier components are more significant.
$\{\mathrm{a}_0, \mathrm{a}_1, \mathrm{a}_2, \ldots, \mathrm{a}_n\} \succ \{\mathrm{b}_0, \mathrm{b}_2, \mathrm{b}_3, \ldots, \mathrm{b}_n\}$ if and only if
$\mathrm{a}_0 \succ \mathrm{b}_0 \mid\mid (\mathrm{a}_0 == \mathrm{b}_0 \;\&\&\; \mathrm{a}_1 \succ \mathrm{b}_1) \mid\mid$
$(\mathrm{a}_0 == \mathrm{b}_0 \;\&\&\; \mathrm{a}_1 == \mathrm{b}_1 \;\&\&\; \mathrm{a}_2 \succ \mathrm{b}_2) \mid\mid \ldots \mid\mid$
$(\mathrm{a}_0 == \mathrm{b}_0 \;\&\&\; \mathrm{a}_1 == \mathrm{b}_1 \;\&\&\; \ldots \;\&\&\; \mathrm{a}_{n-1} == \mathrm{b}_{n-1} \;\&\&\; \mathrm{a}_n \succ \mathrm{b}_n)$
A lexicographic ordering allows tuples to be used as termination metrics.

Mutually Recursive Functions

Tuples can be used to provide termination metrics for mutually recursive functions since you can provide multiple values that the functions may reduce on.
E.g. for the following methods;

```
    method F(i: nat) returns (r: nat) {
        if i <= 2 { r := 1; }
        else {
            var h := H(i - 2);
            r := 1 + h;
        }
    }

    method H(i: nat) returns (r: nat) {
        if i == 0 { r := 0; }
        else {
            var f := F(i);
            var h := H(i - 1);
            r := f + h;
        }
    }
```

the termination matrix would be {i, 1} for H and {i, 0} for F since the call F(i) in H will reduce on 1 $\succ$ 0.

Question 4

Classes

Ghost variables can be used for specification and reasoning only.

```
    ghost var d: T
```

Simple Classes

A simple class consits of only simple object, (i.e. objects that are not stored on the heap).
The specification for a simple class consists of:

- ghost variables for abstract state
- have class invariant, **ghost predicate Valid()**
- Valid() and functions have **reads this**
- constructor has **ensures Valid()**
- methods have **requires Valid()**, **modifies this**, **ensures Valid()**

Concrete states that consist of only simple objects are created and are related to the abstract state in **Valid()**.
The constructor, methods, and functions must satisfy the class specification and will require both concrete and abstract state to be updated.

Complex Classes

Complex classes consist of any combination of simple and complex objects, (i.e. objects that are stored on the heap).
Complex classes require a representation set,

```
ghost var Repr: set<object>
```

Invariant

The invariant valid will consist of the following, where a, a0, a1 are non-composite objects or arrays and b, b0, b1 are composite objects.

```
ghost predicate Valid()
    reads this, Repr
    ensures Valid() ==> this in Repr
{
    this in Repr && ...
}
```

For an array **a**, include;

```
a in Repr
```

For two identically typed arrays **a0, a1**, include;

```
a0 != a1
```

For a non-composite object **b**, include;

```
b in Repr && b.Valid()
```

For two identically typed non-composite objects **b0, b1**, include;

```
b0 != b1
```

For a composite object **c**, include;

```
c in Repr && c.Repr <= Repr &&
this !in c.Repr && c.Valid()
```

For a composite objects **c0, c1** and non-composite objects and arrays **a0, a1, b0, b1**, include;

```
{a0, a1, b0, b1} !! c0.Repr !! c1.Repr
```

Constructor

For a non-composite array or object **a** and a composite object **b**.

```
constructor()
    ensures Valid() && fresh(Repr)
    ensures ... (initial abstract state)
{
    ... (initialise concrete and abstract state)
    new;
    Repr := {this, a, b} + b.Repr;
}
```

Functions

```
function F(x:X): Y()
    requires Valid() && ...
    reads Repr
    ensures F(x) == ... (abstract state)
```

Methods (Mutating)

```
method M(x:X) returns Y()
    requires Valid() && ...
    modifies Repr
    ensures Valid() && fresh(Repr - old(Repr))
    ensures ... (resultant abstract state)
```

Question 5

Lemmas

```
lemma L(x1 : T, x2 : T, . . . , xN : T)
        requires P
        ensures R
{ }
```

Lemmas can be called in a method to **prove** the lemmas property from that point onwards.

Calc

To prove a lemma by hand, you can add a **calc** section into the lemmas body, where γ is the default transitive operator between lines.

```
calc γ {
        5 * (x + 3);
        == 5 * x + 5 * 3;
        == 5x + 15;
}
```

You can use use any transitive operator between lines (e.g. ==>). If no default operator is specified, the default is ==.
The **calc** statements can also be added inline within a method instead of creating and calling a lemma.

Induction

Lemmas can also be used to prove using induction by recursively calling the lemma in the body. E.g.

```
lemma SumLemma(a: array<int>, i: int, j: int)
        requires P
        ensures R
{
        if i == j {
                // base case: Dafny can prove
        }
        else {
                // inductive case
                SumLemma(a, i+1, j);
        }
}
```

2023 Final Exam

Question 1

Provide weakest precondition proofs to determine whether or not the following methods satisfy their specifications.

(a)

```
method M(x: int) returns (r: int)
  requires x >= -2
  ensures r >= 1
{
    { x == -2 || x >= 0 }
    { x + 1 == -1 || x + 1 >= 1 }
    r := x + 1;
    { r == -1 || r >= 1}
    { (r < 0 && r >= -1) || (r >= 0 && r >= 1) }
    { (r < 0 ==> r >= -1) && (r >= 0 ==> r >= 1) }
    if r < 0 {
        { r >= -1 }
        { r + 2 >= 1 }
        r := r + 2;
        { r >= 1}
    }
    { r >= 1}
}

Not correct since !(x >= -2 ==> x == -2 || x >= 0)
        since x >= -2 allows x to be -1.
```

(b)

```
method B(x: int, y: int) returns (r: int)
requires x >= 0 && y >= 0
ensures r == x * y


method A(x: int, y: int) returns (r: int)
  requires y >= 4
  ensures r >= x + y
{
  {y >= 4}
  {y >= 4 && x == x}
  {y >= 4 && x >= x}
  var z := x;
  {y >= 4 && z >= x}
  while z < 0
    invariant y >= 4 && z >= x
  {
    {y >= 4 && z >= x && z < 0}
    {y >= 4 && z + y >= x && z < 0} (Strengthening)
    {y >= 4 && z + y >= x}
    {y >= 4 && z + y >= x}
    z := z + y;
    {y >= 4 && z >= x}
  }
  {z >= 0 && y >= 4 && z >= x} (Strengthening)
  {z >= 0 && y - 1 >= 0 && z * y - 1 >= x} (A.56)
  {z >= 0 && y - 1 >= 0 && forall y' :: y'
                  == z * y - 1 ==> y' >= x}
  r := B(z, y - 1);
  { r >= x}
  { r + y >= x + y}
  r := r + y;
  {r >= x + y}
}

Correct since y >= 4 ==> y >=
```

Question 2

(a)

Write a specification for a Dafny method to reverse an array. For example, given the array [1, 2, 3, 4, 5] the method will change it to [5, 4, 3, 2, 1]. Note that the method should modify an existing array, not create a new one.

```
method Reverse(a: array)
  modifies a
  ensures forall i :: 0 <= i < a.Length ==>
                a[i] == old(a[a.Length-1-i])
```

(b)

Based on your specification, provide a loop specification (guard and invariant) for the Reverse method, and code to initialise the loop variables.

```
var n := 0;
while n < a.Length/2
  invariant 0 <= n <= a.Length/2
  invariant forall i :: 0 <= i < n
            ==> a[i] == old(a[a.Length-1-i])
  invariant forall i :: a.Length-n <= i < a.Length
            ==> a[i] == old(a[a.Length-1-i])
  invariant forall i :: n <= i < a.Length-n
            ==> a[i] == old(a[i])
```

The second and third invariants are instances of the Replace a Constant by a Variable loop design technique. In the second invariant, the constant a.Length is replaced by n. In the third invariant, the constant 0 is replaced by a.Length-n. The final invariant states that nothing between indices n and a.Length-n have been changed by the loop. This is similar to the additional invariant we required for the IncrementArray example in Week 5.

(c)

Provide a termination metric for the loop.

```
decreases a.Length/2 - n
```

Question 3

Provide termination metrics for the following mutually recursive methods

```
method F(i: nat) returns (r: nat) {
  if i <= 2 {
    r := 1;
  } else {
    var h := H(i - 2);
    r := 1 + h;
  }
}
method H(i: nat) returns (r: nat) {
  if i == 0 {
    r := 0;
  } else {
    var f := F (i);
    var h : = H(i - 1);
    r := f + h;
  }
}
```

Justify your choice of termination metrics using the fact that an integer value X decreases to x when X ¿ x & X =¿ 0
Call H from F $i, 1 \succ i - 2, 1$
Call F from H $i, 1 \succ i, 0$
Call H from H $i, 1 \succ i - 1, 0$

F decreases $i, 0$
H decrease $i, 1$

Question 4

(a)

Provide variable declarations representing the abstract and concrete states of the class. Assume that the class has a generic parameter Event corresponding to the event type

```
// abstract
ghost var schedule: seq<Event>
ghost var additions: seq<Event>
ghost const n: nat
ghost var Repr: set<object>
// concrete
var events: array<Event>
var m: int
var n: int
```

(b)

Provide a class invariant, Valid, for the class.

```
ghost predicate Valid( )
    reads this, Repr
    ensures Valid( ) ==> this in Repr
              && |schedule| + |additions| <= N &&
    forall i, j :: 0 < = i < j
              < |schedule+additions| ==>
    (schedule + additions)[i]
              != (schedules + additions)[j]
{
    this in Repr && a in Repr &&
    0 <= m <= n <= a.Length && a.Length == N &&
    a[..m] == schedule && a[m..n] == additions &&
    forall i, j :: 0 <= i < j < n ==> a[i] != a[j]
}
```

(c)

```
constructor (N : int)
    ensures Valid( ) && fresh(Repr)
    ensures schedule == [ ] && additions == [ ]
            && this.N == N
method AddEvent(e: Event)
    requires Valid( ) && e !in schedule
            && e !in additions
            && |schedule + additions| < N
    modifies Repr
    ensures Valid( ) && fresh(Repr - old(Repr))
    ensures additions == old(additions) + [e]
            && schedule == old(schedule)
method Commit( )
    requires Valid( )
    modifies Repr
    ensures Valid( ) && fresh(Repr - old(Repr))
    ensures additions == [] && schedule ==
            old(schedule + additions)
method Abort( )
    requires Valid( )
    modifies Repr
    ensures Valid( ) && fresh(Repr - old(Repr))
    ensures additions == []
            && schedule == old(schedule)
```

Question 5

Recall the datatype definition of a list and function Length from the lectures.

```
datatype List<T> = Nil | Cons(head: T, tail: Lis
function Length<T>(xs: List<T>): nat {
        match xs
        case Nil => 0
        case Cons(_ , tail) => 1 + Length(tail)
}
```

(a)

Write a function Remove which takes a list and an index i of the list as arguments and returns a new list with the element at index i removed. For example, given the list [0, 1, 2, 3] and index 2, the function should return [0, 1, 3].

```
function Remove<T>(xs: List<T>, i: nat): List<T>
  requires i < Length(xs)
{
  match xs
  case Cons(x, tail) => if i == 0 then tail
        else Cons(x, Remove(tail, i-1))
}
```

(b)

The length of the list returned by Remove is one less than the length of the list provided as an argument. Show how this would be stated as an intrinsic property of Remove.
The following is added to the function above

```
  ensures Length(Remove(xs,i)) == Length(xs) - 1
```

(c)

State the property of part (b) as an extrinsic property of Remove.

```
lemma LengthRemove<T>(xs: List<T>, i: nat)
  requires i < Length(xs)
  ensures Length(Remove(xs,i)) == Length(xs) - 1
```

Tut 10.3

```
class Node<T> {
  ghost var s: seq<T>
  ghost var Repr: set<object>
  // concrete state
  var value: T
  var next: Node?<T>

  ghost predicate Valid()
    reads this, Repr
    ensures Valid() ==> this in Repr && |s| > 0
  {
    this in Repr &&
    (next == null ==> s == [value]) &&
    (next != null ==> next in Rep
    && next.Repr <= Repr && this !in next.Repr
    next.Valid() && s == [value] + next.s)
  }

  constructor (v: T)
    ensures Valid() && fresh(Repr)
    ensures s == [v]
  {
    value := v;
    next := null;
    s, Repr := [v], {this};
  }
```

```
method SetNext(n: Node<T>)
  requires Valid() && n.Valid()
    && this !in n.Repr && n.Repr !! Repr
  modifies Repr
  ensures Valid() && fresh(Repr - old(Repr) - n.Repr)
  ensures s == old([s[0]]) + n.s
{
  next := n;
  s, Repr := [value] + n.s, Repr + next.Repr;
}

method GetNext() returns (n: Node?<T>)
  requires Valid()
  ensures n == null ==> |s| == 1
  ensures n != null ==> n in Repr
    && n.Repr <= Repr
    && this !in n.Repr
    && n.Valid() && s == s[0] + n.s
{
  n := next;
}

method GetValue() returns (v: T)
  requires Valid()
  ensures v == s[0]
{
  v := value;
}
}

class Stack<T> {
  ghost var s: seq<T>
  ghost var Repr: set<object>
  // concrete state
  var top: Node?<T>
  ghost predicate Valid()
  reads this, Repr
  ensures Valid() ==> this in Repr
{
  this in Repr &&
  (top == null ==> s == []) &&
  (top != null ==> top in Repr
    && top.Repr <= Repr
    && this !in top.Repr &&
  top.Valid() && top.s == s)
}

constructor ()
  ensures Valid() && fresh(Repr)
  ensures s == []
{
  top := null;
  s, Repr := [], {this};
}

method Push(v: T)
  requires Valid()
  modifies Repr
  ensures Valid()
    && fresh(Repr - old(Repr))
  ensures s == [v] + old(s)
{
  var newNode := new Node(v);
  if top != null {
  newNode.SetNext(top);
  }
  top := newNode;
  s, Repr := [v] + s, {this}
    + newNode.Repr;
}

method Pop() returns (v: T)
  requires s != []
  requires Valid()
  modifies Repr
  ensures Valid()
    && fresh(Repr - old(Repr))
  ensures v == old(s[0])
    && s == old(s[1..])
{
  v := top.GetValue();
  top := top.GetNext();
  s := s[1..];
  // note that the removal of
  // old(top) from Repr is not required
}
}
```