# Construction and Verification of Software 2024/2025
# Model Midterm

Nova School of Science and Technology
Mário Pereira `mjp.pereira@fct.unl.pt`

Version of November 3, 2024

- This is model/self-assessment test that mimics the structure of CVS 2023/2024 midterm. You should take roughly between one hour and half and two hours to solve all the questions.
- The midterm will be open-book, so this *model midterm* is also designed with this premise in mind.
- This document has **8** (**eight**) pages in total. The test is divided in **4** (**four**) groups and **7** (**seven**) questions.
- Group 1 is composed of questions **1** (**one**) to **3** (**four**).
- Group 2 contains question **4** (**six**).
- Group 3 contains question **5** (**seven**).
- The test also presents and Appendix.

# 1 Functional Programming

**Question 1.** Consider the following Coq implementation of Binary Trees of natural values as keys:

```
Inductive tree : Type :=
| Leaf
| Node (l : tree) (v : nat) (r : tree).
```

Consider the property *"there exists a node of a* `tree` *that satisfies a given predicate* `P`*"*, as formally defined by the following rules:

$$(\textsc{ExistsTVal})\ \frac{\texttt{P v}}{\texttt{ExistsT P (Node l v r)}}$$

$$(\textsc{ExistsTLeft})\ \frac{\texttt{ExistsT P l}}{\texttt{ExistsT P (Node l v r)}} \qquad \frac{\texttt{ExistsT P r}}{\texttt{ExistsT P (Node l v r)}}\ (\textsc{ExistsTRight})$$

Complete the following Coq definition

```
Inductive ExistsT : (nat → Prop) → tree → Prop :=
```

according to rules (ExistsTVal), (ExistsTLeft), and (ExistsTRight). □

**Question 2.** Consider the property *"all nodes of a* `tree` *satisfy a given predicate* `P`*"*, as formally defined by the following rules:

$$(\textsc{ForallTLeaf})\ \frac{}{\texttt{ForallT P Leaf}} \qquad \frac{\texttt{ForallT P l} \qquad \texttt{ForallT P r} \qquad \texttt{Pv}}{\texttt{ForallT P (Node l v r)}}\ (\textsc{ForallTNode})$$

Complete the following Coq definition

```
Inductive ForallT : (nat → Prop) → tree → Prop :=
```

according to rules (ForallTLeaf) and (ForallTNode). □

**Question 3.** Consider the property *"an element of type* `tree` *is a Binary Search Tree"*, as formally defined by the following rules:

$$(\textsc{BST\_Leaf})\ \frac{}{\texttt{BST Leaf}}$$

$$\frac{\texttt{BST l} \qquad \texttt{BST r} \qquad \texttt{ForallT (fun y => y < v) l} \qquad \texttt{ForallT (fun y => y > v) r}}{\texttt{BST (Node l v r)}}\ (\textsc{BST\_Node})$$

Complete the following Coq definition

```
Inductive BST : tree → Prop :=
```

according to rules (BST_Leaf) and (BST_Node). □

**Question 4.** Consider the following Coq implementation of the `mem` operation, which checks if an element is bound in a tree:

```
Fixpoint mem (x: nat) (t: tree) : bool :=
  match t with
  | Leaf ⇒ false
  | Node l y r ⇒
     if x <? y then mem x l
     else if y <? x then mem x r
     else true
  end.
```

Prove the following Lemma about the `mem` operation:

```
Lemma mem_ExistsT : ∀ (v: nat) (t : tree),
  BST t → mem x t = true → ExistsT (fun e ⇒ e = x) t.
```

Proceed by induction on the structured of the tree `t`. **Hint:** when `t` is not the `Leaf` tree (inductive case), you should split your proof in sub-cases, one for each branch in the `if..then..else` expression. □

## 2 Loops and Loops Invariants

Consider the following Dafny program:

```
function sum (a: array<int>, i: int, j: int) : int
    requires //FILL HERE
    reads //FILL HERE
{
    if j ≤ i then 0 else a[j-1] + sum(a, i, j-1)
}

method querySum (a: array<int>, i: int, j: int) returns (s: int)
    requires //FILL HERE
    ensures //FILL HERE
{
    s := 0;
    var k := i;

    while (k < j)
        decreases //FILL HERE
        invariant //FILL HERE
        invariant //FILL HERE
    {
        s := s + a[k];
        k := k + 1;
    }
}
```

**Question 5.** Logical function `sum` takes an integer array `a` as argument and returns the sum of elements in the range [i; j[, *i.e.*, index `i` inclusive, index `j` exclusive. Method `querySum`, on the other hand, also computes the sum of the elements of `a` in range [i; j[, but via an imperative implementation.

Complete the specification for function `sum` and method `querySum` with the strongest post-conditions, weakest pre-conditions, the necessary loop invariants and decreasing measures so that Dafny verifies the code without errors.

Each specification element you must provide is marked with the comment *//FILL HERE*. If needed, you can use multiple lines for each specification clause. □

# 3 Abstract Data Types

In appendix A you can find a Dafny implementation of an ADT that represents a queue with a fixed capacity, implemented in a circular buffer. Field `first` stands for the index, in array `data`, of the first element in the queue, while field `size` stands for the number of elements the queue contains.

This class offers `push`, `pop`, and `clear` methods. The capacity of the queue is given as an argument to the constructor, which should always be greater than zero. The `push` and `pop` methods require the appropriate pre-conditions in order to be called. The specification relies on a sequence `logicalView` that contains the elements in the queue at any given point in time.

Recall that `data := new T[N](_ => default)` initializes the array `data` with length `N`, where every position of the array contains the value `default`. Also, recall that the notation `logicalView[1..]` denotes the sequence obtained from `logicalView` starting from the element in position 1 up to the length of the sequence (i.e. `logicalView[1..] = logicalView[1..|logicalView|]`).

**Question 6.** Complete the specification of the `Queue` ADT so that Dafny verifies the code without errors. You are asked to provide the appropriate representation invariants, typestates, strongest post-conditions, weakest pre-conditions possible, and loop invariants. Each specification element you are supposed to complete is marked with the comment `FILL HERE`. If needed, you can use multiple lines for each specification clause. **Note:** in order to hide the internal representation of `Stack`, you are expected to use the *dynamic frames* technique studied in class. □

# A Queue ADT

```
class Queue<T>
{
  var data: array<T>
  var first: int
  var size: int

  ghost var logicalView: seq<T>
  ghost var Repr: set<object>

  ghost predicate Valid ()
    reads this, Repr
  {
    /* FILL HERE */ &&
    ∀ i ::0 ≤ i < size ⟹
      if first + i < data.Length then logicalView[i] == data[/* FILL HERE */]
      else logicalView[i] == data[/* FILL HERE */]
}

  ghost predicate isEmpty ()
    reads this, Repr
  { /* FILL HERE */ }

  ghost predicate notFull ()
    reads this, Repr
  { /* FILL HERE */ }

  ghost predicate notEmpty ()
    reads this, Repr
  { /* FILL HERE */

  constructor (N: int, default: T)
    requires //FILL HERE
    ensures //FILL HERE
  {
    data := new T[N](_ ⇒ default);
    first, size := 0, 0;
    logicalView := [];
    Repr := {this, data};
  }

  method pop () returns (r: T)
    requires //FILL HERE
    modifies //FILL HERE
    ensures //FILL HERE
  {
    r := data[first];
    size := size - 1;
    first := (first + 1)%data.Length;

    logicalView := logicalView[1 ..];
  }

  method push (x: T)
```

```
    requires //FILL HERE
    modifies //FILL HERE
    ensures //FILL HERE
  {
    data[(first + size)%data.Length] := x;
    size := size + 1;

    logicalView := logicalView + [x];
  }

  method clear ()
    requires //FILL HERE
    modifies //FILL HERE
    ensures //FILL HERE
  {
    while (size > 0)
      invariant //FILL HERE
      invariant fresh(Repr-old(Repr))
      modifies Repr
    {
      var _ := pop();
    }
  }
}
```