Software Specification - Dafny Project

QS 2023/2024

Exercise 1: Incomplete Binary Trees (6 val = 1 + 1 + 1 + 3) Recall the serialisation and deserialisation algorithms for binary trees studied in the lectures. The goal of this exercise is to generalise those algorithms and prove their correctness for *incomplete binary trees* as defined below:

```
datatype Tree<V> =
    Leaf(V)
    | SingleNode(V, Tree<V>)
    | DoubleNode(V, Tree<V>, Tree<V>)
```

Informally, an incomplete binary tree can be: **(1)** a leaf node storing a value; **(2)** a single-branching internal node storing a value and one subtree; and **(3)** a double-branching internal node storing a value and two subtrees. In order to serialise incomplete binary trees, we make use of the following code type:

```
datatype Code<V> = CLf(V) | CSNd(V) | CDNd(V)
```

Where each code type represents a specific tree node type in a serialised sequence. More concretely, we use `CLf(V)` to represent a serialised leaf node, `CSNd(V)` to represent a serialised single-branching node, and `CDNd(V)` to represent a serialised double-branching node. The serialisation code is given below:

```
function serialise<V>(t : Tree<V>) : seq<Code<V〉
  decreases t
{
  match t {
    case Leaf(v) ⇒ [ CLf(v) ]
    case SingleNode(v, t) ⇒ serialise(t) + [ CSNd(v) ]
    case DoubleNode(v, t1, t2) ⇒ serialise(t2) + serialise(t1) + [ CDNd(v) ]
  }
}
```

Figure 1 illustrates three possible incomplete trees along with their corresponding serialised traces. For convenience, serialised traces are constructed in a backwards fashion.

1. Implement a recursive Dafny function `deserialise(s:seq<Code<V>>):Tree<V>` that given a sequence of tree codes outputs their corresponding original tree.
2. Write three concrete tests to check the behaviour of the function `serialise`.
3. Write three concrete tests to check the behaviour of the function `deserialise`.
4. Prove that `deserialise` is the inverse of `serialise`, that is, for every incomplete binary tree `t`, it must hold that: `deserialise(serialise(t)) == t`.

Note: In order to obtain the full score, the proofs of 1.4 must be written in *calculational style*.

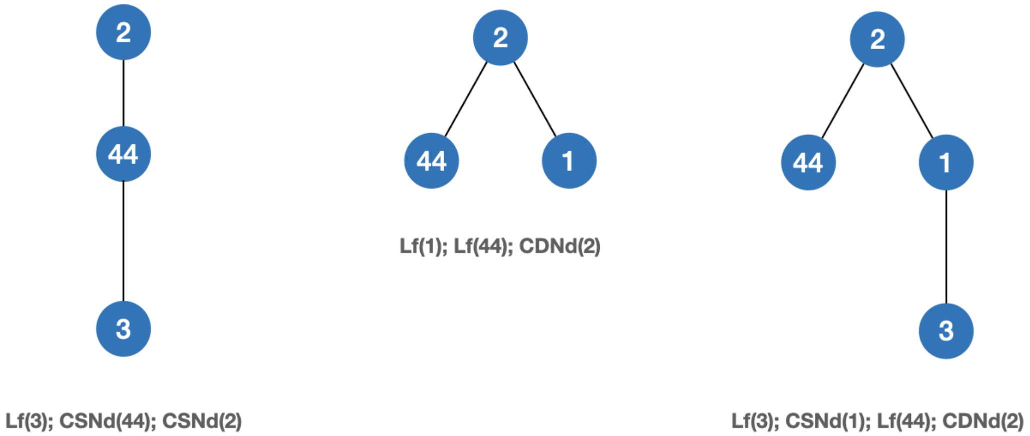


Figure 1: Three incomplete trees and corresponding serialised traces

Exercise 2: Merging Sorted Arrays (6 val = 1.5 + 3 + 1.5) The goal of this exercise is to create a verified Dafny implementation of the MergeSort algorithm. To this end, consider the predicate `sorted` defined below:

```
function sorted(s : seq<int>) : bool {
  ∀ k1, k2 • 0 ≤ k1 ≤ k2 < |s| ⟹ s[k1] ≤ s[k2]
}
```

1. Implement a method `copyArr` that takes an array `a`, an integer left limit `l`, and an integer right limit `r` as parameters and returns a new array containing the elements of `a` starting from index `l` up to, but not including, index `r`. Verify that the implemented method satisfies the specification below:

```
method copyArr(a : array<int>, l : int, r : int) returns (ret : array<int>)
  requires 0 ≤ l < r ≤ a.Length
  ensures ret[..] = a[l..r]
```

2. Implement a method `mergeArr` that takes an array `a`, and three integer indexes `l`, `m`, and `r`, limiting two sorted contiguous segments of `a`, respectively `a[l..m]` and `a[m..r]`. The method should rearrange the elements of `a[l..m]` and `a[m..r]` so that their concatenation, `a[l..r]`, becomes sorted. Verify that the implemented method satisfies the specification below:

```
method mergeArr(a : array<int>, l : int, m : int, r : int)
  requires 0 ≤ l < m < r ≤ a.Length
  requires sorted(a[l..m]) ∧ sorted(a[m..r])
  ensures sorted(a[l..r])
  ensures a[..l] = old(a[..l])
  ensures a[r..] = old(a[r..])
  modifies a
```

In order to obtain full marks, your implementation must be iterative and make use of the method `copyArr` defined in 2.1.

```
class Node {
    ghost var list : seq<int>;
    ghost var footprint : set<Node>;

    var data : int;
    var next : Node?;

    function Valid() : bool
      reads this, footprint
      decreases footprint;
    {
      (this in footprint) ∧
      ((next = null) ⟹ list = [ data ] ∧ footprint = { this }) ∧
      ((next ≠ null) ⟹
        (next in footprint) ∧
        footprint = next.footprint + { this } ∧
        (this ∉ next.footprint) ∧
        list = [ data ] + next.list ∧
        next.Valid())
    }
}
```

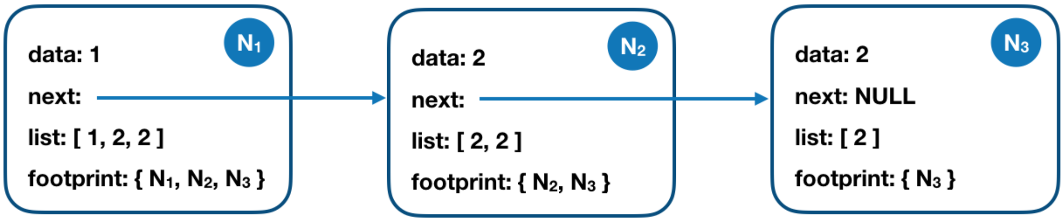
Figure 2: Partial implementation of *list node* class

3. Implement a method `mergeSort` that sorts the given a array of integers `a` and verify that it adheres to the following specification:

```
method mergeSort (a : array<int>)
  ensures sorted(a[..])
  modifies a
```

In order to obtain full marks, your implementation should make use of the method `mergeArr` defined in 2.2. You may find it handy to use a recursive auxiliary method in your implementation of `mergeSort`.

Exercise 3: Reversing Lists (3 val = 1.5 + 1.5) Consider the partial implementation of a *list node* class, `Node`, given in Figure 2. Each list node has two fields: **(1)** the field `data`, storing an integer value, and **(2)** the field `next`, storing the next list node, which may be `null`. Furthermore, for verification purposes, the ghost state of every list node includes the fields: **(i)** `list` that stores the sequence of integers contained in the list headed by the current node; and **(ii)** `footprint` that stores the set of all the list nodes reachable from the current node, including itself. Below, you can see a concrete example of a list composed of three list nodes:



1. Consider the following implementation of a method `reverse` designed to reverse the nodes in the linked list headed by the current node (i.e., the node bound to the `this` keyword):

```
method reverse(tail : Node?) returns (r : Node)
  {
    var old_next := this.next;
    this.next := tail;

    if (old_next = null) {
      r := this; return;
    } else {
      r := old_next.reverse(this);
      return;
    }
  }
```

   Write a specification of `reverse` and add to its body the ghost code required by Dafny to prove it. Your specification should be as precise as possible.

2. Implement and specify a method `extendList` that receives a possibly null node `nd` and an integer value `v` as inputs and returns a new node with data `v` and next element `nd`. The method should have the following signature:

```
method ExtendList(nd : Node?, v : int) returns (r : Node)
```

Exercise 4: Implementing a Queue with Two Lists (5 val = 2 + 1 + 2) Consider the well-known implementation of a *queue* data structure using two lists given in Figure 3. Recall that queues follow the *first-in-first-out* discipline, meaning that the `pop` method retrieves the first element pushed into the queue. Complete the code of the Queue implementation, including:

1. the class ghost fields and validity predicate;

2. the specification of the class constructor;

3. the specifications of the methods `push` and `pop`, along with their respective ghost code.

Instructions

Hand-in Instructions The project is due on the 9th of October, 2022. Be sure to follow the steps described below:

- Your solution must be comprised of four files: one separate file for each exercise (`Ex1.dfy`, `Ex2.dfy`, `Ex3.dfy`, and `Ex4.dfy`). Each Dafny file must be implemented within its own Dafny module.[1]

- Exercises 4 requires the methods defined and/or specified in Exercise 3. Use Dafny's `include` and `import` directives to avoid code duplication.

- Create a `zip` file containing the four answer files and upload it in **Fenix**. Submissions will be closed at 23h59 on the 9th of October, 2022. Do not wait until the last few minutes for submitting the project.

[1] See `https://dafny-lang.github.io/dafny/OnlineTutorial/Modules` for a quick tutorial on how to use Dafny modules.

```
class Queue {

  var lst1 : Node?
  var lst2 : Node?

  constructor ()
  {
    this.lst1 := null;
    this.lst2 := null;

  }

  method push(val : int)
  {
    lst1 := ExtendList(lst1, val);
  }


  method pop() returns (r : int)
  {
    if (lst2 = null) {
      lst2 := lst1.reverse(null);
      lst1 := null;
    }
    r := lst2.data; lst2 := lst2.next;
  }

}
```

Figure 3: Queue Implementation with Two Lists

Project Discussion After submission, you may be asked to present your work so as to streamline the assessment of the project as well as to detect potential fraud situations. During this discussion, you may be required to perform small changes to the submitted code.

Fraud Detection and Plagiarism The submission of the project assumes the commitment of honour that the project was solely executed by the members of the group that are referenced in the files/documents submitted for evaluation. Failure to stand up to this commitment, i.e., the appropriation of work done by other groups or someone else, either voluntarily or involuntarily, will have as consequence the immediate failure of this year's Software Specification course for all students involved (including those who facilitated the occurrence).