CS:5810 Formal Methods in Software Engineering
Fall 2025

Mini Project 2

Due: Friday, November 14, by 11:59pm

This project has two parts, which can be done independently, although we recommend doing them in order. Download the accompanying files

`project2a.dfy` and `project2b.dfy`,

and complete it as specified below. Type your name(s) as indicated in the file, and submit it on ICON. *Only one team member should submit the file*, but make sure to write down the name of both team members in the source files.

This project will test your ability to write functions in the Dafny language, annotate them with specs, check their correctness, and prove and use lemmas about some of them. The project is based on the material presented in Chapters 3 through 6 of the Program Proofs textbook. *You are strongly advised to study those chapters before attempting to solve the problems below.*

A Functions over lists

File `project2a.dfy` contains a number of functions over lists, using the list datatype discussed in Chapter 6 of the textbook. Three ghost functions, elements, isEmpty and isIncreasing, are provided that can be used in the specification of other functions. For any list l, elements(l) returns the set of elements in l, while isEmpty(l) returns true iff l is the empty list. Predicate isIncreasing takes as input an integer list and returns true iff the elements of l are in strictly increasing order.

To talk about finite sets in specifications we use Dafny's builtin set type. See the tutorial at

`http://dafny.org/dafny/OnlineTutorial/Sets`

for the set operators provided in Dafny and their syntax.

A.1 Specifying and implementing list functions

You are to do the following on the remaining functions in the file.

1. Function append, for appending two arbitrary lists together, is defined as in the textbook.

   Provide the *strongest contract* for append that relates the set of elements in the output list to those in the input lists.

2. Function reverse takes as input a list and returns its reverse.

   Provide the strongest contract for reverse that relates the set of elements in the output list to those in the input list.

3. Function len takes as input a list l and returns its length.

   Provide the strongest contract for len that relates its output to the cardinality of the set of elements in l.

4. Function first takes as input a non-empty list l and returns its first element, that is, its head.

   Provide the strongest contract for first that relates its output to the set of elements in l.

5. Function rest takes as input a non-empty list l and returns the list resulting from removing the first element from l.

   Provide the strongest contract for rest that relates its output to the set of elements in l.

6. Function last takes as input a non-empty list l and returns its last element, that is, the deepest element in the list syntax tree.

   Provide an implementation of last as well as the strongest contract that relates its output to the set of elements in l.

7. Predicate member takes a value x of some type T and a list l of elements type T, and returns true iff x occurs in l.

   Provide the strongest contract for member that relates its output to x and the set of elements in l.

8. Function max takes as input a non-empty list of integers and returns its maximum element.

   With the provided implementation, Dafny is not able to prove that the function is terminating. Add a suitable decreases clause for that. The clause you provide may still not be enough for Dafny. In that case, add assert statements in the function's body as needed for Dafny to be able to prove termination.

   Provide also the strongest contract that relates the function's output to the set of elements in the input list.

9. Function min takes as input a non-empty list of integers and returns its minimum element.

   Provide an implementation of min as well as the strongest contract that relates its output to the set of elements in the input list.

   As with max, provide also a suitable decreases clause and insert assert statements in the implementation as needed for Dafny to be able to prove the function's termination.

10. Predicate memberInc takes an integer value x and an increasing list l of integers, and returns true iff x occurs in l.

    Implement memberInc taking advantage that the list is increasing in order to be more efficient than the generic predicate member.

    Provide the strongest contract for memberInc that relates its output to x and the set of elements in l. Add assert statements in the function's body as needed for Dafny to be able to prove the contract.

11. Function insert takes an integer value x and an increasing list l of integers, and returns l if x is already in l; otherwise, it returns the result of inserting x into l so that the new list is still increasing.

    Provide an implementation of insert as well as the strongest contract that expresses the specification above and also relates the set of elements in the output list to the set of elements in the input list.

    Write your code so that it goes over the input list only once — and so, for instance, it does not call member to check if the input integer is already in the input list.

12. Function remove takes an integer value x and an increasing list l of integers, and returns l if x is not in l; otherwise, it returns the result of removing x from l so that the new list is still increasing.

    Provide the strongest contract that expresses the specification above and also relates the set of elements in remove's output list to the set of elements in l. Insert suitable instances of lemma Increasing1 and lemma Increasing2 (see next section) and assert statements in the code as needed to help Dafny prove the contract's postconditions.

For the functions above, *do not change their implementation* if is already provided. The only allowed modifications to the given implementations are the insertion of specification clauses (assertions and lemma calls).

The function contracts and lemma statements you provide should abstract away the concrete implementation of lists. In other words, they may contain applications of the various functions described so far but *may not contain* the list constructors (Nil and Cons), discriminators (Nil? and Cons?), and destructors (head and tail).

A.2 Proving extrinsic properties of list functions

File `project3a.dfy` contains also a number of lemmas about some of the functions from the previous section. You are to provide for each of them a proof sketch that is enough for Dafny to prove the lemma. In all the provided lemmas, automated induction is turned off. For full credit, *your proof sketch should fully work with automated induction off.*

1. Lemma MaxLast states that the maximum value in an increasing list is the last element of the list.

2. Lemma MinFirst states that the minimum value in an increasing list is the first element of the list.

3. Lemma Increasing1 states that the first element of an increasing non-empty list is smaller than all the elements in the rest of the list.

4. Lemma Increasing2 states that no integer smaller than the first element of an increasing non-empty list is in the list.

5. Lemma AppendIncreasing states that appending two increasing lists results in an increasing list provided that one of the two is empty or the last element of the first list is smaller than the first element of the second list.

6. Lemma AppendReverse states that reversing the concatenation (via append) of two lists l1 and l2 is the same as concatenating the reverse of l2 with the reverse of l1.

   This lemma is the one with the most complex proof. It will require auxiliary lemmas about append. Part of the problem is for you to figure out which auxiliary lemmas you need.

   Provide and prove the auxiliary lemmas you need in this problem. For them (and only for them), you are allowed to leave automated induction on.

   Optional, Extra credit. Provide a proof of your auxiliary lemmas with automated induction off.

Note: Dafny may need less detailed sketches if the functions mentioned in a lemma have their contract in place already. So it is generally advisable to do the problems in the previous section first.

B Functions over trees

This part is similar to Part A but focuses a different data structure: binary search trees, implemented as an inductive datatype like the binary tree datatype discussed in class and in Chapter 4 of the textbook. For convenience, and with no loss of generality, we consider only binary trees that store integer values in their internal nodes.

B.1 Specifying and implementing tree functions

A binary *search* tree (BST) is a binary tree satisfying the data structure invariant that the value in any internal node of the tree is (strictly) greater than all the values in the left subtree and (strictly) smaller than all the values in the right subtree of the node. This invariant enables a worst-case log-time search for values in a tree instead of a linear time search for binary trees in general. The price to pay for that is that the invariant must be maintained when inserting or removing values, making the implementation of those operations more complex.

File `project2b.dfy` contains a number of functions over binary trees. Three ghost functions, elements, isEmpty and isSearchTree, are provided that can be used in the specification of other functions. For any binary tree t, elements(t) returns the set of integers in t while isEmpty(t) returns true iff t is the empty tree. Predicate isSearchTree takes as input a binary tree and returns true iff the tree satisfies the binary search invariant above.

The list datatype from Part A is reproduced in `project2b.dfy` but within the module List. The reason is that in this part of the project you will need some of the list functions and lemmas from Part A. Copy inside the List module those that you need. For functions, copy the function in full, including its implementation and contract. For lemmas you can omit their proof (i.e., their entire body). To use those functions and lemmas outside the module, prepend List. to their name, as in List.first(l) .

In the problems below that ask you to add a contract to a function, make sure that Dafny is able to verify the contract. In some cases where Dafny might not be able to do that on its (her?) own, help the system by inserting assert statements in the implementation as needed. Similarly, add a decreases clause as needed for Dafny to prove termination of the function. Of course, also make sure you add only contract or termination clauses that are indeed satisfied by the implementation. This means in particular that you should not have any assert statements in the implementation that Dafny cannot verify.

You are to do the following on the remaining BST functions in the file.

1. Function collect takes BST t and returns in a list the enumeration of t's elements obtained by performing an in-order traversal of the tree (visit left subtree, then root, then right subtree).

   Provide the strongest contract for collect that relates the set of elements in the output tree to those in the input tree.

   A consequence of traversing a BST in in-order fashion is that the elements of the returned list are in (strictly) increasing order. This is stated in the file as an extrinsic property of collect in a lemma. Provide a proof of the lemma with automated induction turned off. Use list functions and lemmas from Part A as needed.

2. Predicate member takes an integer x and a BST t, and returns true iff x occurs in t. Note that it is implemented to take advantage of the BST invariant in order to minimize the amount of search in the tree.

   Provide the strongest contract for member that specifies exactly when member(x, t) is true.

3. **Optional, extra credit** Provide an implementation for a function member2 with the same spec as member but using, as needed, only Boolean operators and predicates, and the binary tree constructors, destructors and discriminators (but no match or if). You can introduce local variables if you want. Make sure in this case too that Dafny is able to verify the function's contract.

4. Function insert takes as input an integer x and a BST t, and returns a BST resulting from inserting x in t so as to maintain the BST invariant.

   Provide the strongest contract for insert that relates the set of elements in the output tree to those in the input tree.

5. Function pred, used by remove, takes as input an integer x and a BST t, and and returns the (largest) predecessor of x in t, if any; otherwise it returns x itself. See the included test method for examples.

   Provide the strongest contract for pred that relates the returned value to the set of elements in the input tree, and also expresses the fact that such value is the predecessor of x in t when such predecessor exists, and is x otherwise.

6. Function remove takes as input an integer x and a BST t, and and returns t if x is not in t; otherwise, it returns the BST resulting from removing x from r.

   Provide the strongest contract for remove that relates the set of elements in the output tree to those in the input tree.[1]

[1]You have probably seen in previous courses implementations of insert and remove for BSTs written in an imperative programming language. Please observe how much cleaner and simpler a functional-style implementation with inductive datatypes is. In addition, it is a lot easier to verify the correctness of such implementations (both manually and automatically) with respect to implementations that apply in-place modifications to the input tree.

Also note that the potential downside of having to reconstruct part of the tree when not using in-place modifications is becoming increasingly less important thanks to advances in optimizing compilers for functional languages.

7. **Optional, extra credit** Provide specification and implementation for function `max` that takes a non-empty BST and returns the maximum integer stored in it. The specification should reflect the fact that the returned value is the maximum element in the tree. The implementation should use `match` to distinguish cases as needed.

Notes.

1. For those functions above that preserve the BST invariant, make sure to reflect that in their contract.
2. For those functions above that are provided with an implementation, *do not change it*. The only allowed modifications to the implementation are the insertion of specification clauses (assertions and lemma calls).

VS Code hints

We recommend you change the default settings of the Dafny extension in Visual Studio Code so that Automatic Verification is triggered only when the file is saved (`onsave`), and not as soon as it is modified (`onchange`).

Also, you may want to lower the Verification Time Limit to 20-30 seconds so you do not have to wait too long when Dafny is struggling to prove something.

Finally, if Dafny seems to be stuck, you can restart it by reloading the VS Code window in which it is running. You can do that by typing Shift-Control-P on a Windows or Linux machine or Shift-Command-P on a Mac. Then type `Reload Window` in the text box that appears at the top of the window.

Submission Instructions

You will be reviewed for:

- The clarity of your implementation and annotations. *Keep both short and readable.* Submissions with complicated, lengthy, redundant, or unused code or specs may be rejected.
- The correctness of your code with respect to original specification.
- The correctness of your annotations.

For each part, *your Dafny code should be free of syntax and typing errors.* You may get no credit for that part otherwise. Submission with verification errors or warnings will receive partial credit.

Recall that only one team member should submit your solution. However, both are required to submit a team evaluation, as specified on Piazza.