Universidad ORT Uruguay
Facultad de Ingeniería

Lógica de Programación con Dafny

Entregado como requisito para la obtención del título de Licenciado en Ingeniería de Software

Matías Hernández - 169236

Tutor: Álvaro Tasistro

2021

Yo, Matías Hernández, declaro que el trabajo que se presenta en esta obra es de mi propia mano. Puedo asegurar que:

- La obra fue producida en su totalidad mientras se realizaba el Trabajo integrador de Licenciatura en Ingeniería de Software;
- Cuando he consultado el trabajo publicado por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente mía;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué fue contribuido por otros, y qué fue contribuido por mi;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Matías Hernández

27-07-2021

Agradecimientos

Agradezco al Dr. Álvaro Tasistro, quien aceptó ser tutor de mi tesis. Me ayudó a encontrar referencias bibliográficas que sirvieron como base para adquirir el conocimiento necesario para elaborar la tesis y también realizó revisiones donde compartió sus conocimientos y sus opiniones que sirvieron como guía durante todo el proceso.

Abstract

Dafny es un lenguaje diseñado por K. Rustan M. Leino en *Microsoft Research* que soporta la especificación formal de código a través del uso de invariantes, pre condiciones, post condiciones y lemas (propiedades). Se realizó una búsqueda de aplicaciones de Dafny a nivel educativo y se encontró que el *Imperial College London* introduce a sus alumnos el concepto de verificación completa de especificaciones de programas utilizando Dafny, lo cual motivó a realizar esta investigación para evaluar la aplicabilidad del lenguaje Dafny para entornos académicos. Para poder realizar la evaluación, se estarán abordando implementaciones de diferentes algoritmos conocidos de ordenamiento y búsqueda, junto a lemas que verifican sus propiedades. Las demostraciones se realizarán utilizando inducción y *calculations* los cuales permiten realizar demostraciones muy intuitivas. Como metodología de desarrollo, se estará aplicando la metodología vista en "Fundamentos de la computación" para los algoritmos funcionales e invariantes al estilo Dijkstra para los algoritmos iterativos. Para finalizar, se analizará también a Dafny como herramienta productiva, reescribiendo los lemas del módulo educativo a su mínima expresión para evaluar el potencial del compilador de Dafny y por ende su aplicabilidad en entornos productivos. Se concluirá que Dafny tiene mucho potencial principalmente como herramienta educativa, pero además cierto nicho de mercado también podría beneficiarse del potencial del compilador en un entorno productivo. Como implicancia de esta tesis, se evaluará agregar Dafny en materias existentes del área de "Teoría de la computación" de la Universidad ORT Uruguay o la creación de una materia electiva con Dafny como principal herramienta de apoyo en el curso.

Palabras clave
algoritmos; dafny; estructuras de datos; especificación; verificación automática

Índice general

1 Introducción

1.1 Objetivos

A continuación se estarán detallando los dos objetivos de la tesis:

1. Dafny como herramienta educativa:

   Mediante la realización de algoritmos funcionales utilizando el método visto en "Fundamentos de la Computación" y algoritmos iterativos utilizando invariantes al estilo Dijkstra, concluir si Dafny es una herramienta apropiada para ser utilizada como herramienta educativa.

2. Dafny como herramienta productiva:

   Mostrar como la verificación automática de programas de Dafny está suficientemente desarrollada como para poder omitir la demostración de muchas propiedades utilizadas en el módulo educativo. Concluir si Dafny es una herramienta útil para el desarrollo productivo de *software*.

1.2 Estado del arte

En esta sección se estarán comentando diferentes documentos de diversas partes del mundo, que fueron tenidos en cuenta para la realización de la tesis. Se describirán brevemente y se indicarán referencias a los mismos.

Invariants: A Generative Approach to Programming **[1]**

Daniel Zingaro

El autor refleja en este libro un estudio muy amplio sobre la utilización de *invariantes* para demostrar el correcto funcionamiento de los algoritmos.

Es una lectura muy amena y fácil de seguir que te introduce al mundo de las invariantes en algoritmos realizados en el lenguaje *Java* y fue de mucha utilidad a la hora de pensar los algoritmos imperativos que fueron demostrados utilizando invariantes.

Programación Basada en Invariantes: Un Enfoque Didáctico [2]

Alejandro Milieris y Eitan Fogel — Universidad ORT Uruguay

Se presenta una metodología para la derivación de algoritmos a partir de pre y post condiciones con base en el uso de invariantes para su resolución.

Como aporte significativo, realizan un método que ha servido como base para realizar las demostraciones de los algoritmos imperativos realizados en mi tesis.

Verified Programming in Dafny [3]

Will Sonnex y Sophia Drossopoulou — Imperial College London

Es un curso introductorio para el uso de Dafny como lenguaje para desarrollar especificaciones totalmente verificadas.

Fue una de las grandes motivaciones para la realización de esta tesis ya que el curso se pone en práctica en la prestigiosa universidad *Imperial College London.*

También se puede notar con la lectura, que Dafny funcional es muy similar a Haskell y Dafny imperativo es muy similar a .NET, siendo estos lenguajes enseñados en la Universidad ORT Uruguay.

De este documento también se extrae la idea de generar dos grandes módulos, uno funcional y otro imperativo, para poder mostrar ambas metodologías de demostraciones.

Verificación de Algoritmos y Estructuras de Datos en Dafny [4]

Rubén Rafael Rubio Cuéllar — Universidad Complutense de Madrid

Es una tesis muy completa que muestra diversos algoritmos y estructuras de datos conocidas utilizando Dafny y realizando las especificaciones de sus pre y post condiciones. Además hace una muy buena introducción al funcionamiento interno de Dafny como herramienta de verificación automática de códigos.

***Accessible Software Verification with Dafny* [5]**

K. Rustan M. Leino — IEEE Software

El diseñador de Dafny comenta en este artículo los aspectos fundamentales de la verificación formal de *software* que ofrece su lenguaje. Muestra ejemplos de verificaciones tanto para el paradigma funcional como para el imperativo.

***Well-founded Functions and Extreme Predicates in Dafny: A Tutorial* [6]**

K. Rustan M. Leino — Microsoft Research

El diseñador de Dafny muestra el potencial que tiene su lenguaje para establecer demostraciones de lemas para algoritmos funcionales. Fue un documento muy importante para entender el potencial de la herramienta y conocer el uso de las *calculations* que serán utilizadas en todas las demostraciones que se realizarán en la tesis para el módulo funcional.

1.3 Estructura de la tesis

El capítulo 2 presenta una introducción a Dafny como herramienta de desarrollo. Este capítulo es una base de conocimiento para el resto de los capítulos posteriores.

Módulo educativo

El capítulo 3 presenta al algoritmo funcional *TreeSort* junto a las demostraciones de la propiedades de integridad y ordenamiento mediante el uso de lemas.

El capítulo 4 presenta algoritmos imperativos de búsqueda y de ordenamiento conocidos que se auto verifican utilizando invariantes.

El capítulo 5 presenta varias implementaciones de la sucesión de Fibonacci y luego se comparan mediante propiedades con una implementación denominada base para demostrar que son equivalentes.

Módulo productivo

El capítulo 6 presenta los lemas del módulo educativo, realizados para un ambiente productivo.

Repositorio de códigos

Los códigos utilizados en la tesis están disponibles públicamente en el siguiente enlace: `https://github.com/matiashrnndz/programming-logic-with-dafny`

2 Dafny como herramienta

Dafny es una herramienta de verificación de *software* que incluye un lenguaje de programación especializado en la verificación formal de código. Fue creado por el grupo *Research in Software Engineering (RiSE)* de *Microsoft Research* y diseñado por K. Rustan M. Leino. Dafny cuenta con un manual de referencia oficial el cual fue utilizado para conocer en detalle el funcionamiento del lenguaje. [7] [8]

El usuario escribe tanto la especificación como la implementación de los programas. Es un lenguaje de programación orientado a objetos, imperativo, secuencial, soporta clases genéricas y asignación dinámica. También soporta la escritura de algoritmos funcionales. La especificación de estos algoritmos se construye incluyendo pre y post condiciones y determinando una métrica de terminación. Los programas hechos con Dafny son estáticamente verificados y esto ayuda a verificar una corrección total del mismo. [9]

La verificación de programas funciona compilando el programa Dafny a un lenguaje de verificación intermediario denominado Boogie, de tal forma que si Boogie asegura la corrección de un programa, también estaría asegurando la corrección del programa hecho en Dafny. La herramienta Boogie es utilizada para generar verificaciones de las condiciones utilizando su motor de razonamiento lógico. [9]

A continuación se pasarán a explicar los principales aspectos de Dafny que se deben conocer para poder realizar la lectura de la tesis.

Lemas

Los lemas, también nombrados *ghost methods*, son funciones que garantizan que cierta propiedad se cumpla.

Las pre condiciones de dicho lema pueden ser determinadas mediante el uso de la instrucción *requires*.

Las post condiciones pueden ser determinadas mediante el uso de la instrucción *ensures*.

Existen anotaciones que son utilizadas para indicar en dónde se está realizando la inducción. Por ejemplo, si la inducción es en una lista, podría aparecer una anotación del siguiente estilo {*:induction list*}.

En esta tesis, se utilizarán *calculations* que son denominadas por la instrucción *calc*, que sirven para realizar una demostración "paso a paso".

Invariantes

Las invariantes sirven para probar que cierta propiedad se mantenga sin variación en determinado *scope*.

De utilizarse una invariante en un *loop*, se le denomina *loop invariant* y sirve para especificarle al verificador de Dafny, que esa propiedad debe cumplirse en cada iteración del *loop*.

Se suele utilizar el método de invariantes para poder realizar la especificación de métodos iterativos.

Métrica de terminación

En Dafny se debe declarar mediante el uso de una instrucción *decreases*, la forma en que un *loop* o una recursión va decreciendo para asegurar su finalización.

Existen casos que Dafny detecta la forma de decrecimiento automáticamente y no necesita que le sea especificado.

Tipo de datos *multiset*

Un multiset es un tipo de dato nativo de Dafny, que cuenta con operaciones de conjunto que pueden ser aplicadas sobre los mismos.

Por definición, son conjuntos de elementos que pueden tener elementos repetidos.

Este tipo de datos será muy utilizado en la tesis para poder demostrar de forma más sencilla los lemas que traten sobre la integridad de los elementos.

3 Programación Funcional en Dafny

En este capítulo se abordará el algoritmo *TreeSort* funcional junto a dos propiedades que aseguran el correcto funcionamiento del mismo.

Para poder probar que el algoritmo definido en la primera sección sea correcto, es necesario asegurar la integridad de los elementos que intervienen y que el resultado esté ordenado.

Se mostrarán también estructuras de datos auxiliares, funciones para dichas estructuras de datos y lemas que aseguren ciertas propiedades de dichas funciones.

Una vez terminado el capítulo, se habrá culminado con este ejemplo completo de programación funcional en Dafny.

3.1 Algoritmo *TreeSort*

En esta sección se definirá la función de ordenamiento *TreeSort*.

TreeSort es un algoritmo de ordenamiento basado en la estructura de datos árbol binario de búsqueda, que se lo denominará *BST* por su nombre en inglés *binary search tree*.

- La lógica es la siguiente:

    1. Recibe por parámetro la lista de elementos a ser ordenada.
    2. Carga un *BST* con los elementos de dicha lista (se insertan ordenadamente por razones inherentes al *BST*).
    3. Aplica un recorrida del *BST* de forma transversal *in order*.

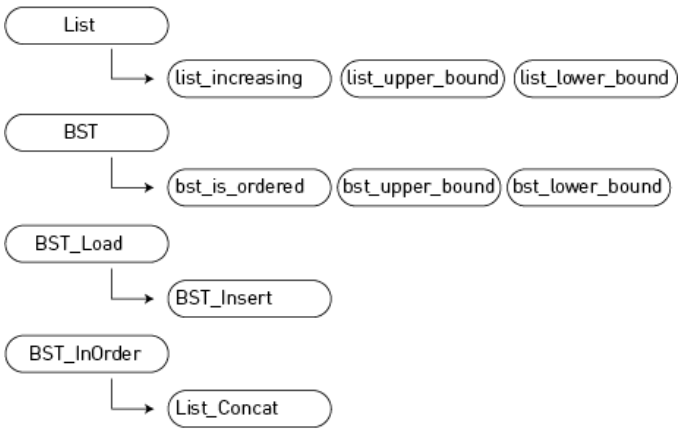
```
1 function method TreeSort(list:List<int>) : (sortedList:List<int>)
2   decreases list
3 {
4   BST_InOrder(BST_Load(list))
5 }
```

Como se puede observar, para que la función compile, estaría faltando definir las siguientes implementaciones:

- *List*: representa una lista de elementos, la cual será utilizada tanto para recibir los elementos del *TreeSort* como para devolver dichos elementos de manera ordenada.
- *BST*: representa un árbol binario de búsqueda, el cuál será utilizado para procesar el ordenamiento de los elementos del *TreeSort*.
- *BST_Load*: función que recibe una lista de elementos y retorna un *BST* cargado con dichos elementos.
- *BST_InOrder*: función que recibe un *BST* y ordena sus elementos en una lista.

En la próxima sub sección se ilustrará un esquema que muestra las implementaciones que estarán siendo abordadas a continuación para dar por finalizada la implementación del algoritmo *TreeSort*.

3.1.1 Esquema de implementaciones



3.1.2 Estructura de datos *List*

Una lista puede ser construida vacía (*Nil*) o agregándole un elemento (*Cons*) constituído por un valor (*head*) y el resto de la lista que continúa a partir de dicho elemento (*tail*).

```
datatype List<T> = Nil | Cons(head:T, tail:List<T>)
```

A continuación, se definirán predicados que serán utilizados para validar que una lista está ordenada de manera creciente o si un elemento es cota superior o inferior de todos los elementos de una lista.

Predicado que indica si una lista está ordenada de manera creciente

Nombre del predicado: list_increasing

Predicado: retorna verdadero cuando la lista está ordenada de manera creciente.

```
predicate list_increasing(list:List<T>)
  decreases list
{
  match list {
    case Nil => true
    case Cons(head, Nil) => true
    case Cons(head, Cons(ht, tail)) => head <= ht && list_increasing(Cons(ht, tail))
  }
}
```

Predicado que indica si un elemento es cota superior de una lista

Nombre del predicado: list_upper_bound

Predicado: retornar verdadero cuando el elemento "d", es cota superior a todos los elementos de una lista.

```
predicate list_upper_bound(list:List<T>, d:T)
  decreases list
{
  match list {
    case Nil => true
    case Cons(head, tail) => (d >= head) && list_upper_bound(tail, d)
  }
}
```

Predicado que indica si un elemento es cota inferior de una lista

Nombre del predicado: list_lower_bound

Predicado: retornar verdadero cuando el elemento "d", es cota inferior a todos los elementos de una lista.

```
predicate list_lower_bound(list:List<T>, d:T)
  decreases list
{
  match list {
    case Nil => true
    case Cons(head, tail) => (d <= head) && list_lower_bound(tail, d)
  }
}
```

3.1.3 Estructura de datos *BST*

Un *BST* puede ser construido vacío (*Nil*) o agregándole un elemento(*Node*) constituido por un valor (*data*), su árbol hijo izquierdo (*left*) y su árbol hijo derecho (*right*).

```
datatype BST<T> = Nil | Node(left:BST<T>, data:T, right:BST<T>)
```

A continuación, se definirán predicados que serán utilizados para validar que un árbol efectivamente es un árbol binario de búsqueda.

Predicado que indica si un árbol binario es un árbol binario de búsqueda

Nombre del predicado: *bst_ordered*

Predicado: retorna un booleano dependiendo si se cumple o no la condición de ordenación de un *BST*.

Para que un *BST* sea considerado como ordenado, debe estar vacío o cumplir las siguientes propiedades:

1. El sub árbol izquierdo debe estar ordenado (*left*)
2. El sub árbol derecho debe estar ordenado (*right*)

3. El valor del nodo debe ser mayor o igual a todos los valores de los nodos del lado izquierdo.

4. El valor del nodo debe ser menor a todos los valores de los nodos del lado derecho.

```
predicate bst_ordered(tree:BST<T>)
  decreases tree
{
  match tree {
    case Nil => true
    case Node(left, x, right) =>
      bst_ordered(left) &&
      bst_ordered(right) &&
      bst_upper_bound(left, x) &&
      bst_lower_bound(right, x)
  }
}
```

A continuación se definirán los dos predicados utilizados por *bst_ordered*.

Predicado que indica si un elemento es cota superior de un árbol binario de búsqueda

Nombre del predicado: *bst_upper_bound*

Predicado: retorna verdadero cuando el elemento "d", es cota superior a todos los elementos del *BST*.

```
predicate bst_upper_bound(tree:BST<T>, d:T)
  decreases tree
{
  match tree {
    case Nil => true
    case Node(left, x, right) => d >= x && bst_upper_bound(left, d) && bst_upper_bound(right, d)
  }
}
```

Predicado que indica si un elemento es cota inferior de un árbol binario de búsqueda

Nombre del predicado: *bst_lower_bound*

Predicado: retorna verdadero cuando el elemento "d", es cota inferior a todos los elementos del *BST*.

```
predicate bst_lower_bound(tree:BST<T>, d:T)
  decreases tree
{
  match tree {
    case Nil => true
    case Node(left, x, right) => d <= x && bst_lower_bound(left, d) && bst_lower_bound(right, d)
  }
}
```

A continuación se realizará la primera de las dos funciones que restan por implementar para finalizar *TreeSort*.

3.1.4 Función de carga de elementos en un árbol binario de búsqueda

Nombre de la función: *BST_Load*

Funcionalidad: recibe como parámetro una lista y genera un *BST* cargando dichos elementos.

Se utilizará la función *BST_Insert* para realizar cada una de las inserciones de *BST_Load* y se aplicarán llamadas recursivas para poder recorrer todos los elementos de una lista.

```
function method BST_Load(list:List<T>) : (tree:BST<T>)
  decreases list
{
  match list {
    case Nil => BST.Nil
    case Cons(head, tail) => BST_Insert(BST_Load(tail), head)
  }
}
```

Como se puede observar, falta definir la función *BST_Insert*, por lo cual se procederá a definir a continuación.

Función de inserción de un elemento en un árbol binario de búsqueda

Nombre de la función: *BST_Insert*

Funcionalidad: recibe como parámetro un *BST* y un valor y retorna un *BST* con el valor insertado de manera ordenada.

```
function method BST_Insert(tree:BST<T>, d:T) : (result:BST<T>)
  decreases tree
{
  match tree {
    case Nil => Node(BST.Nil, d, BST.Nil)
    case Node(left, x, right) =>
      if (d < x) then
        Node(BST_Insert(left, d), x , right)
      else
        Node(left, x, BST_Insert(right, d))
  }
}
```

Explicación del criterio de ordenamiento del *BST*:

1. Para el caso en que el dato sea menor al nodo actual, se realiza un paso recursivo hacia la izquierda.
2. Para el caso en que el dato sea mayor o igual al nodo actual, se realiza un paso recursivo hacia la derecha.
3. Se inserta el dato cuando se llega a una hoja del árbol binario de búsqueda representada por *Nil*.

Finalizado *BST_Load*, sólo resta por definir la función *BST_InOrder*, para que compile la implementación de *TreeSort* realizada al comienzo del capítulo.

3.1.5 Función de recorrida transversal *in order* de un árbol binario de búsqueda

Nombre de la función: *BST_InOrder*

Funcionalidad: recibe como parámetro un *BST* y retorna una lista con los elementos ordenados de manera creciente.

BST_InOrder será utilizada por el algoritmo de *TreeSort* para obtener la lista ordenada de manera in order del *BST* que fue cargado previamente.

Para su implementación, se realizará inducción sobre el *BST* y se tendrá que utilizar la función *List_Concat* que todavía no ha sido implementada.

- Caso base (*Nil*):
  1. Se devuelve la lista vacía.

- Paso recursivo (*Node(left, x, right)*):

    1. Se crea un *Cons* con el dato del nodo como *head* y la llamada recursiva de *BST_InOrder* para el *BST* de la derecha como *tail*.
    2. Se concatena con *List_Concat* la llamada recursiva de *BST_InOrder* para el *BST* de la izquierda, con el *Cons* previamente construido.

```
function method BST_InOrder(tree:BST<T>) : (result:List<T>)
  decreases tree
{
  match tree {
    case Nil => List.Nil
    case Node(left, x, right) => List_Concat(BST_InOrder(left), Cons(x, BST_InOrder(right)))
  }
}
```

Se procederá a implementar *List_Concat* a continuación.

Función de concatenación de listas

Nombre de la función: *List_Concat*

Funcionalidad: recibe como parámetros dos listas y como resultado concatena al final de la primera lista, toda la segunda lista.

Para realizar la implementación, se crea un *Cons* con el dato del primer elemento de la lista como *head* y la llamada recursiva de *List_Concat* para el *tail* de la lista a y b.

```
function method List_Concat(a:List<T>, b:List<T>) : List<T>
  decreases a
{
  match a {
    case Nil => b
    case Cons(head, tail) => Cons(head, List_Concat(tail, b))
  }
}
```

Con estas implementaciones finalizadas, se da por finalizado el algoritmo funcional *TreeSort* definido al comienzo del capítulo.

3.2 Propiedad de integridad de los elementos del *TreeSort*

Nombre del lema: *Lemma_TreeSortIntegrity*

En esta sección se estará abordando la propiedad del *TreeSort* que verifica que los elementos de la lista inicial, sean los mismos que los de la lista ordenada.

Como se puede observar en el código, este lema utiliza otros dos lemas que se verán en las siguientes secciones.

- *Lemma_BSTInOrderIntegrity*: asegura que los elementos retornados por *BST_InOrder* sean los mismos que los elementos del *BST* de entrada.
- *Lemma_BSTLoadIntegrity*: asegura que los elementos del resultado de *BST_Load*, sean los mismos que los de la lista de entrada de la función.

```
1  lemma Lemma_TreeSortIntegrity(list:List<T>)
2    ensures List_ToMultiset(TreeSort(list)) == List_ToMultiset(list)
3  {
4    calc == {
5      List_ToMultiset(TreeSort(list));
6        { assert TreeSort(list) == BST_InOrder(BST_Load(list)); }
7      List_ToMultiset(BST_InOrder(BST_Load(list)));
8        { Lemma_BSTInOrderIntegrity(BST_Load(list)); }
9      BST_ToMultiset(BST_Load(list));
10       { Lemma_BSTLoadIntegrity(list); }
11     List_ToMultiset(list);
12   }
13 }
```

Asegurando que las funciones *BST_InOrder* y *BST_Load* mantengan la integridad de los datos, se puede comprobar que se mantiene la integridad de datos en el resultado del *TreeSort*.

Para que se pueda realizar estas demostraciones se necesitará poder convertir los elementos del árbol binario de búsqueda y los elementos de las listas en *multiset*.

Función de conversión de lista a *multiset*

Nombre de la función: *List_ToMultiset*

Funcionalidad: convertir los elementos de una lista en un *multiset* de sus elementos.

```
function method List_ToMultiset(list:List<T>) : (m:multiset<T>)
  decreases list
{
  match list {
    case Nil => multiset{}
    case Cons(head, tail) => multiset{head} + List_ToMultiset(tail)
  }
}
```

Función de conversión de árbol binario de búsqueda a multiset

Nombre de la función: *BST_ToMultiset*

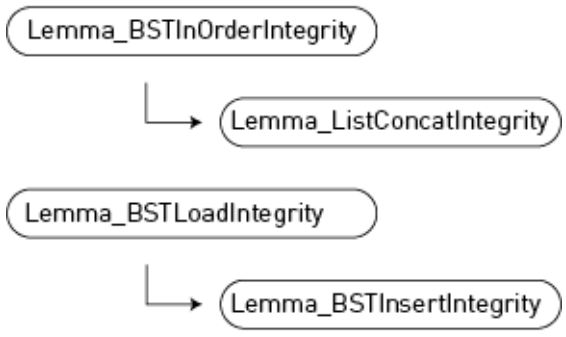
Funcionalidad: recibe como parámetro un *BST* y lo transforma en un *multiset* de sus elementos.

```
function method BST_ToMultiset(tree:BST<T>) : multiset<T>
  decreases tree
{
  match tree {
    case Nil => multiset{}
    case Node(left, x, right) => multiset{x} + BST_ToMultiset(left) + BST_ToMultiset(right)
  }
}
```

Se podrá observar que la única diferencia con *List_ToMultiset* es que se realizan dos llamadas recursivas porque la estructura de datos no es lineal.

En la próxima sub sección se ilustrará un esquema que muestra las implementaciones que estarán siendo abordadas a continuación para dar por finalizada la implementación del *Lemma_TreeSortIntegrity*.

3.2.1 Esquema de implementaciones



3.2.2 Propiedad de integridad de elementos en recorrida transversal *in order*

Nombre del lema: *Lemma_BSTInOrderIntegrity*

Propiedad: asegurar que se mantenga la integridad de los datos entre el *BST* inicial y la lista retornada por la función *BST_InOrder*.

Se realizará la demostración por inducción en el árbol binario de búsqueda.

```
1  lemma {:induction tree} Lemma_BSTInOrderIntegrity(tree:BST<T>)
2    ensures BST_ToMultiset(tree) == List_ToMultiset(BST_InOrder(tree))
3    decreases tree
4  {
5    match tree {
6      case Nil =>
7        calc == {
8          BST_ToMultiset(tree);
9            { assert tree == BST.Nil; }
10         BST_ToMultiset(BST.Nil);
11           { assert BST_ToMultiset(BST.Nil) == multiset{}; }
12         multiset{};
13           { assert multiset{} == List_ToMultiset(List.Nil); }
14         List_ToMultiset(List.Nil);
15           { assert List.Nil == BST_InOrder(BST.Nil); }
16         List_ToMultiset(BST_InOrder(BST.Nil));
17           { assert BST.Nil == tree; }
18         List_ToMultiset(BST_InOrder(tree));
19       }
20     case Node(left, x, right) =>
21       calc == {
22         List_ToMultiset(BST_InOrder(tree));
23           { assert tree == Node(left, x, right); }
24         List_ToMultiset(BST_InOrder(Node(left, x, right)));
25           { assert List_ToMultiset(BST_InOrder(Node(left, x, right)))
26                 == List_ToMultiset(List_Concat(BST_InOrder(left), Cons(x, BST_InOrder(right)))); }
27         List_ToMultiset(List_Concat(BST_InOrder(left), Cons(x, BST_InOrder(right))));
28           { Lemma_ListConcatIntegrity(BST_InOrder(left), Cons(x, BST_InOrder(right))); }
29         List_ToMultiset(BST_InOrder(left)) + List_ToMultiset(Cons(x, BST_InOrder(right)));
30           { assert List_ToMultiset(Cons(x, BST_InOrder(right)))
31                 == List_ToMultiset(Cons(x, List.Nil)) + List_ToMultiset(BST_InOrder(right)); }
32         List_ToMultiset(BST_InOrder(left)) + List_ToMultiset(Cons(x, List.Nil)) + List_ToMultiset(
       BST_InOrder(right));
33           { assert List_ToMultiset(Cons(x, List.Nil))
34                 == multiset{x} + List_ToMultiset(List.Nil); }
35         List_ToMultiset(BST_InOrder(left)) + multiset{x} + List_ToMultiset(List.Nil) +
       List_ToMultiset(BST_InOrder(right));
36           { assert List_ToMultiset(List.Nil) == multiset{}; }
37         List_ToMultiset(BST_InOrder(left)) + multiset{x} + multiset{} + List_ToMultiset(
       BST_InOrder(right));
38           { assert multiset{x} + multiset{} == multiset{x}; }
39         List_ToMultiset(BST_InOrder(left)) + multiset{x} + List_ToMultiset(BST_InOrder(right));
40           { Lemma_BSTInOrderIntegrity(left); }
41           { Lemma_BSTInOrderIntegrity(right); }
42         BST_ToMultiset(tree);
43       }
44   }
```

```
45 }
```

En la linea 28, se puede notar como se utiliza *Lemma_ListConcatIntegrity* que prueba que los elementos de las dos listas concatenadas por *List_Concat*, sean los mismos que la unión de ambas listas.

En la linea 39, se puede observar que luego de aplicar algunas definiciones, se despeja el elemento actual "x" como *multiset*$\{x\}$.

En las lineas 40 y 41, se realizan dos llamadas recursivas una para el sub árbol izquierdo y otra para el sub árbol derecho.

Al finalizar las llamadas recursivas, quedará como resultado la unión de cada uno de los *multiset* unitarios de los elementos del *BST*. Con esto, se puede demostrar que dicha unión de *multiset* es igual al resultado de aplicar *BST_ToMultiset* al mismo árbol.

Nos resta por implementar *Lemma_ListConcatIntegrity* que es utilizado en la demostración.

Propiedad de integridad de elementos en la concatenación de listas

Nombre del lema: *Lemma_ListConcatIntegrity*

Propiedad: asegurar que la unión de los elementos de ambas listas que recibe como parámetro *List_Concat*, sean los mismos que los elementos de las listas concatenadas.

En las lineas 28 y 29, se llega a la hipótesis de inducción llamando recursivamente al lema para completar el paso inductivo.

```
1  lemma {:induction a} Lemma_ListConcatIntegrity(a:List<T>, b:List<T>)
2    ensures List_ToMultiset(List_Concat(a, b)) == List_ToMultiset(a) + List_ToMultiset(b)
3    decreases a, b
4  {
5    match a {
6      case Nil =>
7        calc == {
8          List_ToMultiset(List_Concat(a, b));
9            { assert a == List.Nil; }
10         List_ToMultiset(List_Concat(List.Nil, b));
11           { assert List_ToMultiset(List_Concat(List.Nil, b))
12                 == List_ToMultiset(List.Nil) + List_ToMultiset(b); }
13         List_ToMultiset(List.Nil) + List_ToMultiset(b);
14           { assert List_ToMultiset(List.Nil) == multiset{}; }
15         multiset{} + List_ToMultiset(b);
16           { assert multiset{} == List_ToMultiset(List.Nil); }
```

```
17          List_ToMultiset(List.Nil) + List_ToMultiset(b);
18            { assert List_ToMultiset(List.Nil) == List_ToMultiset(a); }
19          List_ToMultiset(a) + List_ToMultiset(b);
20        }
21      case Cons(ha, ta) =>
22        calc == {
23          List_ToMultiset(List_Concat(a, b));
24            { assert a == Cons(ha, ta); }
25          List_ToMultiset(List_Concat(Cons(ha, ta), b));
26            { assert List_ToMultiset(List_Concat(Cons(ha, ta), b))
27                  == List_ToMultiset(Cons(ha, ta)) + List_ToMultiset(b); }
28          List_ToMultiset(Cons(ha, ta)) + List_ToMultiset(b);
29            { Lemma_ListConcatIntegrity(ta, b); }
30          List_ToMultiset(a) + List_ToMultiset(b);
31        }
32    }
33  }
```

Al haber sido probado *Lemma_BSTInOrderIntegrity*, se puede asegurar que se mantiene la integridad de los elementos al pasar por la función *BST_InOrder*.

Resta por demostrar que al pasar por la función *BST_Load*, también se mantenga la integridad de los elementos.

3.2.3 Propiedad de integridad de la carga de elementos en un árbol binario de búsqueda

Nombre del lema: *Lemma_BSTLoadIntegrity*

Propiedad: asegurar que el *multiset* de elementos de la lista inicial, sea el mismo que el *multiset* de elementos luego de haberse aplicado *BST_Load*.

Es importante poder asegurar esta propiedad porque se desea mantener la integridad de los datos en el proceso de carga del *BST*.

```
1   lemma {:induction list} Lemma_BSTLoadIntegrity(list:List<T>)
2     ensures BST_ToMultiset(BST_Load(list)) == List_ToMultiset(list)
3     decreases list
4   {
5     match list {
6       case Nil =>
7         calc == {
8           BST_ToMultiset(BST_Load(list));
9             { assert list == List.Nil; }
10          BST_ToMultiset(BST_Load(List.Nil));
11            { assert BST_Load(List.Nil) == BST.Nil; }
12          BST_ToMultiset(BST.Nil);
13            { assert BST_ToMultiset(BST.Nil) == multiset{}; }
14          multiset{};
15            { assert multiset{} == List_ToMultiset(List.Nil); }
16          List_ToMultiset(List.Nil);
```

```
17          { assert List.Nil == list; }
18        List_ToMultiset(list);
19      }
20    case Cons(head, tail) =>
21      calc == {
22        BST_ToMultiset(BST_Load(list));
23          { assert list == Cons(head, tail); }
24        BST_ToMultiset(BST_Load(Cons(head, tail)));
25          { assert BST_Load(Cons(head, tail)) == BST_Insert(BST_Load(tail), head); }
26        BST_ToMultiset(BST_Insert(BST_Load(tail), head));
27          { Lemma_BSTInsertIntegrity(BST_Load(tail), head); }
28        BST_ToMultiset(BST_Load(tail)) + multiset{head};
29          { Lemma_BSTLoadIntegrity(tail); }
30        List_ToMultiset(list);
31      }
32  }
33 }
```

En la linea 27, se utiliza *Lemma_BSTInsertIntegrity*, que servirá para poder separar en un *multiset* aparte el elemento actual (*head*) del resto que falta por cargar. En la sección siguiente se estará comentando este lema.

En las lineas 28 y 29, se llega a la hipótesis de inducción llamando recursivamente al lema para completar el paso inductivo.

Propiedad de integridad de los elementos pre-existentes de una lista en la inserción

Nombre del lema: *Lemma_BSTInsertIntegrity*

Propiedad: asegurar que el multiset de elementos resultante al haber sido insertado un elemento en un *BST* utilizando la función *BST_Insert*, sea equivalente a la unión del *multiset* del *BST* y del *multiset* del elemento.

```
1  lemma {:induction tree} Lemma_BSTInsertIntegrity(tree:BST<T>, d:T)
2    ensures BST_ToMultiset(BST_Insert(tree, d)) == BST_ToMultiset(tree) + multiset{d}
3    decreases tree
4  {
5    match tree {
6      case Nil =>
7        calc == {
8          BST_ToMultiset(BST_Insert(tree, d));
9            { assert tree == BST.Nil; }
10         BST_ToMultiset(BST_Insert(BST.Nil, d));
11           { assert BST_Insert(BST.Nil, d) == Node(BST.Nil, d, BST.Nil); }
12         BST_ToMultiset(Node(BST.Nil, d, BST.Nil));
13           { assert BST_ToMultiset(Node(BST.Nil, d, BST.Nil))
14                 == BST_ToMultiset(BST.Nil) + multiset{d} + BST_ToMultiset(BST.Nil); }
15         BST_ToMultiset(BST.Nil) + multiset{d} + BST_ToMultiset(BST.Nil);
16           { assert BST_ToMultiset(BST.Nil) == multiset{}; }
17         BST_ToMultiset(BST.Nil) + multiset{d} + multiset{};
18           { assert multiset{d} + multiset{} == multiset{d}; }
```

```
19          BST_ToMultiset(BST.Nil) + multiset{d};
20            { assert BST.Nil == tree; }
21          BST_ToMultiset(tree) + multiset{d};
22        }
23      case Node(left, x, right) =>
24        calc == {
25          BST_ToMultiset(BST_Insert(tree, d));
26            { assert tree == Node(left, x, right); }
27          BST_ToMultiset(BST_Insert(Node(left, x, right), d));
28          { if (d < x) {
29            calc == {
30              BST_ToMultiset(BST_Insert(Node(left, x, right), d));
31                { assert BST_Insert(Node(left, x, right), d)
32                      == Node(BST_Insert(left, d), x , right); }
33              BST_ToMultiset(Node(BST_Insert(left, d), x , right));
34                { assert BST_ToMultiset(Node(BST_Insert(left, d), x , right))
35                      == BST_ToMultiset(BST_Insert(left, d)) + multiset{x} + BST_ToMultiset(right);}
36              BST_ToMultiset(BST_Insert(left, d)) + multiset{x} + BST_ToMultiset(right);
37                { Lemma_BSTInsertIntegrity(left, d); }
38            }
39          } else {
40            calc == {
41              BST_ToMultiset(BST_Insert(Node(left, x, right), d));
42                { assert BST_Insert(Node(left, x, right), d)
43                      == Node(left, x, BST_Insert(right, d)); }
44              BST_ToMultiset(Node(left, x, BST_Insert(right, d)));
45                { assert BST_ToMultiset(Node(BST_Insert(left, d), x , right))
46                      == BST_ToMultiset(left) + multiset{x} + BST_ToMultiset(BST_Insert(right, d));}
47              BST_ToMultiset(left) + multiset{x} + BST_ToMultiset(BST_Insert(right, d));
48                { Lemma_BSTInsertIntegrity(right, d); }
49            }
50          } }
51          BST_ToMultiset(tree) + multiset{d};
52        }
53    }
54  }
```

En las lineas 28 y 39, al ser *BST_Insert* una función que tiene una condicional en su implementación, será necesario utilizar la cláusula *if* y *else* para abarcar ambos casos posibles.

En las lineas 29 y 40, por requerimiento de la herramienta Dafny, se debe escribir nuevamente una instrucción *calc* para continuar con la demostración dentro del *if* y *else*.

Como se puede ver, en cada uno de estos casos, se realiza un paso de deducción aislando al elemento actual como $multiset\{x\}$. Luego, se aplicará una llamada recursiva sólo para el sub árbol correspondiente.

Con estas implementaciones finalizadas, se da por finalizado *Lemma_TreeSortIntegrity* definido al comienzo del capítulo.

3.3 Propiedad de ordenamiento del resultado del *TreeSort*

Nombre del lema: Lemma_TreeSortOrdering

En esta sección, se estará abordando la propiedad del *TreeSort* que verifica que los elementos retornados estén ordenados de manera creciente.

Como se puede observar en el código, este lema utiliza otros dos lemas que se verán en las siguientes secciones.

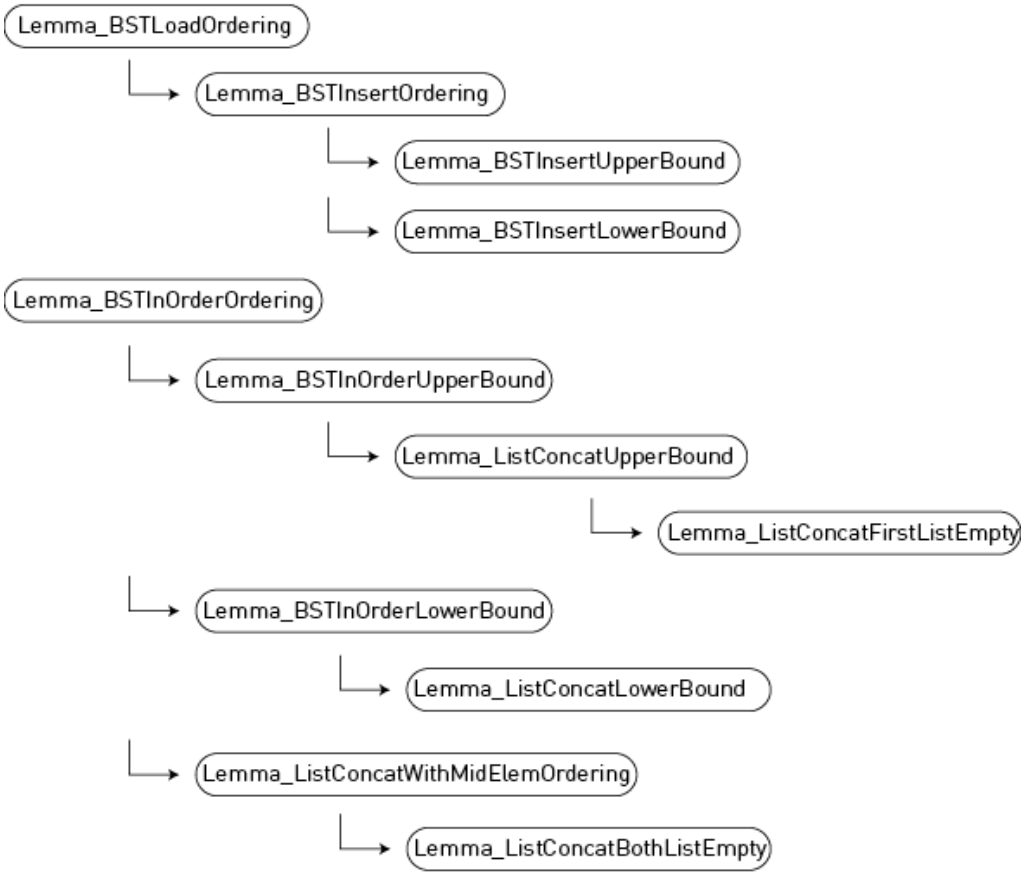
- *Lemma_BSTLoadOrdering*: asegura que el *BST* retornado por *BST_Load* esté ordenado.
- *Lemma_BSTInOrderOrdering*: asegura que la lista retornada por *BST_InOrder* esté ordenada de manera creciente.

Una vez se cumplan esas dos propiedades, se puede demostrar que la lista retornada por *TreeSort* también está ordenada crecientemente.

```
1  lemma Lemma_TreeSortOrdering(list:List<T>)
2    ensures list_increasing(TreeSort(list))
3  {
4    calc == {
5      list_increasing(TreeSort(list));
6        { assert TreeSort(list) == BST_InOrder(BST_Load(list)); }
7      list_increasing(BST_InOrder(BST_Load(list)));
8        { Lemma_BSTLoadOrdering(list); }
9        { assert bst_ordered(BST_Load(list)); }
10       { Lemma_BSTInOrderOrdering(BST_Load(list)); }
11     true;
12   }
13 }
```

En la próxima sub sección se ilustrará un esquema que muestra las implementaciones que estarán siendo abordadas a continuación para dar por finalizada la implementación del *Lemma_TreeSortOrdering*.

3.3.1 Esquema de implementaciones



3.3.2 Propiedad de ordenamiento de la carga de un árbol binario de búsqueda

Nombre del lema: *Lemma_BSTLoadOrdering*

Propiedad: asegurar que el resultado de la función *BST_Load* esté ordenado.

Se utilizará el predicado *bst_ordered* para poder determinar que se cumple la propiedad de ordenamiento de la función *BST_Load*, según fue definido en la sección anterior.

Esto es importante porque *TreeSort* utiliza *BST_InOrder* de *BST_Load* para su funcionamiento.

```
1  lemma {:induction list} Lemma_BSTLoadOrdering(list:List<T>)
2    ensures bst_ordered(BST_Load(list))
3    decreases list
4  {
5    match list {
6      case Nil =>
7        calc == {
8          bst_ordered(BST_Load(list));
9            { assert list == List.Nil; }
10         bst_ordered(BST_Load(List.Nil));
11           { assert BST_Load(List.Nil) == BST.Nil; }
12         bst_ordered(BST.Nil);
13           { assert bst_ordered(BST.Nil) == true; }
14         true;
15       }
16     case Cons(head, tail) =>
17       calc == {
18         bst_ordered(BST_Load(list));
19           { assert list == Cons(head, tail); }
20         bst_ordered(BST_Load(Cons(head, tail)));
21           { assert BST_Load(Cons(head, tail))
22                 == BST_Insert(BST_Load(tail), head); }
23         bst_ordered(BST_Insert(BST_Load(tail), head));
24           { Lemma_BSTInsertOrdering(BST_Load(tail), head); }
25         true;
26       }
27   }
28 }
```

En la linea 23, se puede notar que lo que falta para poder probar este lema, es poder asegurar que luego de haber sido aplicado *BST_Insert* de un elemento en un *BST*, el *BST* se mantenga ordenado.

Para poder asegurar dicha propiedad, se realizará a continuación *Lemma_BSTInsertOrdering*.

Propiedad de ordenamiento de la inserción en un árbol binario de búsqueda

Nombre del lema: *Lemma_BSTInsertOrdering*

Propiedad: asegurar que el *BST* resultante de la inserción de un elemento utilizando la función *BST_Insert*, esté ordenado.

```
1  lemma {:induction tree} Lemma_BSTInsertOrdering(tree:BST<T>, d:T)
2    requires bst_ordered(tree)
3    ensures bst_ordered(BST_Insert(tree, d))
4    decreases tree
5  {
6    match tree {
```

```
7       case Nil =>
8         calc == {
9           bst_ordered(BST_Insert(tree, d));
10            { assert tree == BST.Nil; }
11          bst_ordered(BST_Insert(BST.Nil, d));
12            { assert BST_Insert(BST.Nil, d) == Node(BST.Nil, d, BST.Nil); }
13          bst_ordered(Node(BST.Nil, d, BST.Nil));
14          true;
15        }
16      case Node(left, x, right) =>
17        calc == {
18          bst_ordered(BST_Insert(tree, d));
19            { assert tree == Node(left, x, right); }
20          bst_ordered(BST_Insert(Node(left, x, right), d));
21          { if (d < x) {
22              calc == {
23                bst_ordered(Node(BST_Insert(left, d), x, right));
24                  { Lemma_BSTInsertUpperBound(left, d, x); }
25                  { assert bst_upper_bound(BST_Insert(left, d), x); }
26                  { Lemma_BSTInsertOrdering(left, d); }
27                bst_ordered(Node(BST.Nil, d, BST.Nil));
28              }
29            } else {
30              calc == {
31                bst_ordered(Node(left, x, BST_Insert(right, d)));
32                  { Lemma_BSTInsertLowerBound(right, d, x); }
33                  { assert bst_lower_bound(BST_Insert(right, d), x); }
34                  { Lemma_BSTInsertOrdering(right, d); }
35                bst_ordered(Node(BST.Nil, d, BST.Nil));
36              }
37            } }
38          true;
39        }
40    }
41  }
```

Dada la condición de *BST_Insert*, que evalúa hacia qué lado del *BST* se debe continuar la recurrencia según el valor del nodo, se debe realizar la misma condición en la demostración del lema.

Para poder asegurar que el *BST* esté ordenado según su definición, deben estar ordenados también sus sub árboles y se deben cumplir las condiciones de cota (superior e inferior).

Por esa razón, a continuación se deberán realizar otros dos lemas, uno que compruebe que el elemento a ser insertado sea cota superior del sub árbol izquierdo y otro que compruebe que el elemento a ser insertado sea cota inferior del sub árbol derecho.

Propiedad de cota superior en la inserción a un árbol binario de búsqueda

Nombre del lema: *Lemma_BSTInsertUpperBound*

Propiedad: dado un elemento "b" que pertenezca al *BST* y sea cota superior del mismo y otro elemento "d" a insertar por *BST_Insert* que sea menor a "b", asegura que el elemento "b" también sea cota superior del *BST* con el elemento "d" insertado.

```
lemma {:induction tree} Lemma_BSTInsertUpperBound(tree:BST<T>, d:T, b:T)
  requires bst_ordered(tree)
  requires bst_upper_bound(tree, b)
  requires b > d
  ensures bst_upper_bound(BST_Insert(tree, d), b)
  decreases tree
{
  match tree {
    case Nil =>
      calc == {
        bst_upper_bound(BST_Insert(tree, d), b);
          { assert tree == BST.Nil; }
        bst_upper_bound(BST_Insert(BST.Nil, d), b);
          { assert bst_upper_bound(BST_Insert(BST.Nil, d), b)
                == bst_upper_bound(Node(BST.Nil, d, BST.Nil), b); }
        bst_upper_bound(Node(BST.Nil, d, BST.Nil), b);
          { assert bst_upper_bound(Node(BST.Nil, d, BST.Nil), b)
                == (b >= d && bst_upper_bound(BST.Nil, b) && bst_upper_bound(BST.Nil, b)); }
        (b >= d && bst_upper_bound(BST.Nil, b) && bst_upper_bound(BST.Nil, b));
          { assert bst_upper_bound(BST.Nil, b) == true; }
        (b >= d && true && true);
          { assert (b >= d && true && true) == (b >= d); }
        (b >= d);
          { assert b > d; }
        true;
      }
    case Node(left, x, right) =>
      calc == {
        bst_upper_bound(BST_Insert(tree, d), b);
          { assert tree == Node(left, x, right); }
        bst_upper_bound(BST_Insert(Node(left, x, right), d), b);
          { Lemma_BSTInsertUpperBound(left, d, b); }
          { Lemma_BSTInsertUpperBound(right, d, b); }
        true;
      }
  }
}
```

Propiedad de cota inferior en la inserción a un árbol binario de búsqueda

Nombre del lema: *Lemma_BSTInsertLowerBound*

Propiedad: dado un elemento "b" que pertenezca al *BST* y sea cota inferior del mismo y otro elemento "d" a insertar por *BST_Insert* que sea mayor o igual a "b", asegura que el elemento "b" también sea cota inferior del *BST* con el elemento "d" insertado.

```
lemma {:induction tree} Lemma_BSTInsertLowerBound(tree:BST<T>, d:T, b:T)
  requires bst_ordered(tree)
  requires bst_lower_bound(tree, b)
  requires b <= d
  ensures bst_lower_bound(BST_Insert(tree, d), b)
  decreases tree
{
  match tree {
    case Nil =>
      calc == {
        bst_lower_bound(BST_Insert(tree, d), b);
          { assert tree == BST.Nil; }
        bst_lower_bound(BST_Insert(BST.Nil, d), b);
          { assert bst_lower_bound(BST_Insert(BST.Nil, d), b)
                == bst_lower_bound(Node(BST.Nil, d, BST.Nil), b); }
        bst_lower_bound(Node(BST.Nil, d, BST.Nil), b);
          { assert bst_lower_bound(Node(BST.Nil, d, BST.Nil), b)
                == (b <= d && bst_lower_bound(BST.Nil, b) && bst_lower_bound(BST.Nil, b)); }
        (b <= d && bst_lower_bound(BST.Nil, b) && bst_lower_bound(BST.Nil, b));
          { assert bst_lower_bound(BST.Nil, b) == true; }
        (b <= d && true && true);
          { assert (b <= d && true && true) == (b <= d); }
        (b <= d);
          { assert b <= d; }
        true;
      }
    case Node(left, x, right) =>
      calc == {
        bst_lower_bound(BST_Insert(tree, d), b);
          { assert tree == Node(left, x, right); }
        bst_lower_bound(BST_Insert(Node(left, x, right), d), b);
          { Lemma_BSTInsertLowerBound(left, d, b); }
          { Lemma_BSTInsertLowerBound(right, d, b); }
        true;
      }
  }
}
```

Con la finalización de estos dos lemas referentes a las cotas de la función *BST_Insert*, se puede dar por finalizado el lema de ordenamiento del mismo y por ende, también se puede dar por finalizado el lema de ordenamiento de la función *BST_Load*.

Resta por probar el último lema utilizado por *Lemma_TreeSortOrdering* sobre el ordenamiento de la función *BST_InOrder*.

3.3.3 Propiedad de ordenamiento en recorrida transversal *in order*

Nombre del lema: *Lemma_BSTInOrderOrdering*

Propiedad: asegurar que el resultado de aplicar *BST_InOrder* sobre un *BST*, retorne una lista ordenada creciente.

Primero se comentarán ciertos lemas que serán utilizados en la demostración:

- *Lemma_BSTInOrderUpperBound*: asegura que si un elemento es cota superior del *BST*, también lo es de la lista ordenada *in order* de dicho *BST*.
- *Lemma_BSTInOrderLowerBound*: asegura que si un elemento es cota inferior del *BST*, también lo es de la lista ordenada *in order* de dicho *BST*.
- *Lemma_ListConcatWithMidElemOrdering*: dadas dos listas ordenadas crecientemente y un elemento que sea cota superior de la primera y cota inferior de la segunda, asegura que la concatenación de las dos listas y el elemento en el medio, también esté ordenada crecientemente.

La solución a la demostración se basa en poder llamar al lema *Lemma_ListConcatWithMidElemOrdering*, por esa razón, se satisfacen los requerimientos de dicho lema en las lineas 24 y 26.

```
1  lemma {:induction tree} Lemma_BSTInOrderOrdering(tree:BST<T>)
2    requires bst_ordered(tree)
3    ensures list_increasing(BST_InOrder(tree))
4  {
5    match tree {
6      case Nil =>
7        calc == {
8          list_increasing(BST_InOrder(tree));
9            { assert tree == BST.Nil; }
10         list_increasing(BST_InOrder(BST.Nil));
11           { assert BST_InOrder(BST.Nil) == List.Nil; }
12         list_increasing(List.Nil);
13         true;
14       }
15     case Node(left, x, right) =>
16       calc == {
17         list_increasing(BST_InOrder(tree));
18           { assert tree == Node(left, x, right); }
19         list_increasing(BST_InOrder(Node(left, x, right)));
20           { assert BST_InOrder(Node(left, x, right))
21                 == List_Concat(BST_InOrder(left), Cons(x, BST_InOrder(right))); }
22         list_increasing(List_Concat(BST_InOrder(left), Cons(x, BST_InOrder(right))));
23           { assert bst_lower_bound(right, x); }
24           { Lemma_BSTInOrderLowerBound(right, x); }
```

```
25            { assert bst_upper_bound(left, x); }
26            { Lemma_BSTInOrderUpperBound(left, x); }
27            { Lemma_ListConcatWithMidElemOrdering(BST_InOrder(left), x, BST_InOrder(right)); }
28          true;
29        }
30    }
31  }
```

A continuación se implementarán los tres lemas restantes para finalizar la demostración de la propiedad de ordenamiento de la función *BST_InOrder*.

3.3.3.1 Propiedad de cota superior en ordenamiento de recorrida transversal *in order*

Nombre del lema: *Lemma_BSTInOrderUpperBound*

Propiedad: asegurar que si un elemento es cota superior del *BST*, también lo es de la lista ordenada *in order* de dicho *BST*.

```
1   lemma {:induction tree} Lemma_BSTInOrderUpperBound(tree:BST<T>, d:T)
2     requires bst_ordered(tree)
3     requires bst_upper_bound(tree, d)
4     ensures list_upper_bound(BST_InOrder(tree), d)
5   {
6     match tree {
7       case Nil =>
8         calc == {
9           list_upper_bound(BST_InOrder(tree), d);
10            { assert tree == BST.Nil; }
11          list_upper_bound(BST_InOrder(BST.Nil), d);
12            { assert BST_InOrder(BST.Nil) == List.Nil; }
13          list_upper_bound(List.Nil, d);
14          true;
15        }
16        case Node(left, x, right) =>
17          calc == {
18            list_upper_bound(BST_InOrder(tree), d);
19              { assert tree == Node(left, x, right); }
20            list_upper_bound(BST_InOrder(Node(left, x, right)), d);
21              { assert BST_InOrder(Node(left, x, right))
22                    == List_Concat(BST_InOrder(left), Cons(x, BST_InOrder(right))); }
23            list_upper_bound(List_Concat(BST_InOrder(left), Cons(x, BST_InOrder(right))), d);
24              { assert list_upper_bound(BST_InOrder(left), d); }
25              { assert list_upper_bound(BST_InOrder(right), d); }
26              { assert d >= x; }
27              { Lemma_ListConcatUpperBound(BST_InOrder(left), BST_InOrder(right), d, x); }
28            true;
29          }
30    }
31  }
```

Hay que tener en cuenta que *BST_InOrder* retorna una lista ordenada de elementos, por lo tanto, cuando se apliquen las definiciones sobre el caso *Node(left, x, right)*, habrá que asegurar lo siguiente:

1. "d" debe ser mayor o igual a "x".
2. "d" debe ser cota superior a *BST_InOrder* del sub árbol izquierdo (*left*).
3. "d" debe ser cota superior al *BST_InOrder* del sub árbol derecho (*right*).

Una vez cumplidas dichas condiciones, se podrá aplicar *Lemma_ListConcatUpperBound*, que demuestra lo que necesitamos probar en esta propiedad y se pasará a explicar a continuación.

Propiedad de cota superior en la concatenación de listas

Nombre del lema: *Lemma_ListConcatUpperBound*

Propiedad: asegurar que si un elemento es cota superior de dos listas, también es cota superior de la concatenación de ambas listas.

Como particularidad, se puede notar que se deben anidar un *match* dentro de otro, para generar una nueva posibilidad de finalización del lema. De esta forma, los tres posibles caminos que pueden tomar las instrucciones son los siguientes:

1. La primera lista está vacía, entonces se demuestra directamente dada la hipótesis del lema.
2. La segunda lista está vacía, entonces se demuestra directamente dada la hipótesis del lema.
3. Cuando las dos listas contienen algún elemento, se realiza un paso recursivo con la finalidad de iterar sobre las listas hasta llegar a una de las dos condiciones de finalización nombradas anteriormente.

```
1 lemma {:induction listLeft} Lemma_ListConcatUpperBound(listLeft:List<T>, listRight:List<T>, d:T, x
      :T)
2   requires list_upper_bound(listLeft, d)
3   requires list_upper_bound(listRight, d)
4   requires d >= x
5   ensures list_upper_bound(List_Concat(listLeft, Cons(x, listRight)), d)
6   decreases listLeft
```

```
7  {
8    match listLeft {
9      case Nil =>
10       calc == {
11         list_upper_bound(List_Concat(listLeft, Cons(x, listRight)), d);
12           { assert listLeft == List.Nil; }
13         list_upper_bound(List_Concat(List.Nil, Cons(x, listRight)), d);
14           { Lemma_ListConcatFirstListEmpty(List.Nil, Cons(x, listRight)); }
15         list_upper_bound(Cons(x, listRight), d);
16           { assert list_upper_bound(listRight, d); }
17           { assert d >= x; }
18         true;
19       }
20     case Cons(lh, lt) =>
21       match lt {
22         case Nil =>
23           calc == {
24             list_upper_bound(List_Concat(listLeft, Cons(x, listRight)), d);
25               { assert listLeft == Cons(lh, List.Nil); }
26             list_upper_bound(List_Concat(Cons(lh, List.Nil), Cons(x, listRight)), d);
27               { assert List_Concat(Cons(lh, List.Nil), Cons(x, listRight))
28                     == Cons(lh, List_Concat(List.Nil, Cons(x, listRight))); }
29             list_upper_bound(Cons(lh, List_Concat(List.Nil, Cons(x, listRight))), d);
30               { Lemma_ListConcatFirstListEmpty(List.Nil, Cons(x, listRight)); }
31             list_upper_bound(Cons(lh, Cons(x, listRight)), d);
32               { assert d >= x; }
33               { assert list_upper_bound(listRight, d); }
34             true;
35           }
36         case Cons(tah, tat) =>
37           calc ==> {
38             list_upper_bound(List_Concat(listLeft, Cons(x, listRight)), d);
39               { assert listLeft == Cons(lh, lt); }
40             list_upper_bound(List_Concat(Cons(lh, lt), Cons(x, listRight)), d);
41               { assert List_Concat(Cons(lh, lt), Cons(x, listRight))
42                     == Cons(lh, List_Concat(lt, Cons(x, listRight))); }
43             list_upper_bound(Cons(lh, List_Concat(lt, Cons(x, listRight))), d);
44               { assert lt == Cons(tah, tat); }
45             list_upper_bound(Cons(lh, List_Concat(Cons(tah, tat), Cons(x, listRight))), d);
46               { assert (d >= lh && list_upper_bound(List_Concat(Cons(tah, tat), Cons(x, listRight)
     ), d)); }
47             d >= x && list_upper_bound(List_Concat(Cons(tah, tat), Cons(x, listRight)), d);
48               { Lemma_ListConcatUpperBound(Cons(tah, tat), Cons(x, listRight), d, x); }
49             true;
50           }
51       }
52   }
53 }
```

En las lineas 14 y 29, se utiliza *Lemma_ListConcatFirstListEmpty* que se demostrará a continuación.

Propiedad de concatenación con la primera lista vacía

Nombre del lema: *Lemma_ListConcatFirstListEmpty*

Propiedad: asegurar que al aplicarse la concatenación de listas con la primera lista vacía, el resultado sea la segunda lista.

```
lemma {:induction a} Lemma_ListConcatFirstListEmpty(a:List<T>, b:List<T>)
  requires a == List.Nil
  ensures List_Concat(a, b) == b
{
  calc == {
    List_Concat(a, b);
      { assert a == List.Nil; }
    List_Concat(List.Nil, b);
      { assert List_Concat(List.Nil, b) == b; }
    b;
  }
}
```

Se puede dar por finalizado *Lemma_BSTInOrderUpperBound*.

Análogamente se pasará a demostrar la propiedad de cota inferior de la función *BST_InOrder*.

3.3.3.2 Propiedad de cota inferior en recorrida transversal *in order*

Nombre del lema: *Lemma_BSTInOrderLowerBound*

Propiedad: asegurar que si un elemento es cota inferior del *BST*, también lo es de la lista ordenada *in order* de dicho *BST*.

```
lemma {:induction tree} Lemma_BSTInOrderLowerBound(tree:BST<T>, d:T)
  requires bst_ordered(tree)
  requires bst_lower_bound(tree, d)
  ensures list_lower_bound(BST_InOrder(tree), d)
{
  match tree {
    case Nil =>
      calc == {
        list_lower_bound(BST_InOrder(tree), d);
          { assert tree == BST.Nil; }
        list_lower_bound(BST_InOrder(BST.Nil), d);
          { assert BST_InOrder(BST.Nil) == List.Nil; }
        list_lower_bound(List.Nil, d);
        true;
      }
      case Node(left, x, right) =>
        calc == {
          list_lower_bound(BST_InOrder(tree), d);
            { assert tree == Node(left, x, right); }
          list_lower_bound(BST_InOrder(Node(left, x, right)), d);
            { assert BST_InOrder(Node(left, x, right))
                  == List_Concat(BST_InOrder(left), Cons(x, BST_InOrder(right))); }
          list_lower_bound(List_Concat(BST_InOrder(left), Cons(x, BST_InOrder(right))), d);
```

```
24              { assert list_lower_bound(BST_InOrder(left), d); }
25              { assert list_lower_bound(BST_InOrder(right), d); }
26              { assert d <= x; }
27              { Lemma_ListConcatLowerBound(BST_InOrder(left), BST_InOrder(right), d, x); }
28            true;
29          }
30    }
31  }
```

Hay que tener en cuenta que *BST_InOrder* retorna una lista ordenada de elementos, por lo tanto, cuando se apliquen las definiciones sobre el caso *Node(left, x, right)*, habrá que asegurar lo siguiente:

1. "d" debe ser menor o igual a "x".
2. "d" debe ser cota inferior a *BST_InOrder* del sub árbol izquierdo (*left*).
3. "d" debe ser cota inferior al *BST_InOrder* del sub árbol derecho (*right*).

Una vez cumplidas dichas condiciones, se podrá aplicar *Lemma_ListConcatLowerBound*, que demuestra lo que necesitamos probar en esta propiedad y se pasará a explicar a continuación.

Propiedad de cota inferior en la concatenación de listas

Nombre del lema: *Lemma_ListConcatLowerBound*

Propiedad: asegurar que si un elemento es cota inferior de dos listas, también es cota inferior de la concatenación de ambas listas.

Como en el caso del lema que especificaba el caso de la cota superior, se puede notar que se deben anidar un *match* dentro de otro, para generar una nueva posibilidad de finalización del lema. De esta forma, los tres posibles caminos que pueden tomar las instrucciones son los siguientes:

1. La primera lista está vacía, entonces se demuestra directamente dada la hipótesis del lema.
2. La segunda lista está vacía, entonces se demuestra directamente dada la hipótesis del lema.
3. Cuando las dos listas contienen algún elemento, se realiza un paso recursivo con la finalidad de iterar sobre las listas hasta llegar a una de las dos condiciones de finalización nombradas anteriormente.

```
lemma {:induction listLeft} Lemma_ListConcatLowerBound(listLeft:List<T>, listRight:List<T>, d:T, x
    :T)
  requires list_lower_bound(listLeft, d)
  requires list_lower_bound(listRight, d)
  requires d <= x
  ensures list_lower_bound(List_Concat(listLeft, Cons(x, listRight)), d)
  decreases listLeft
{
  match listLeft {
    case Nil =>
      calc == {
        list_lower_bound(List_Concat(listLeft, Cons(x, listRight)), d);
          { assert listLeft == List.Nil; }
        list_lower_bound(List_Concat(List.Nil, Cons(x, listRight)), d);
          { Lemma_ListConcatFirstListEmpty(List.Nil, Cons(x, listRight)); }
        list_lower_bound(Cons(x, listRight), d);
          { assert list_lower_bound(listRight, d); }
          { assert d <= x; }
        true;
      }
    case Cons(lh, lt) =>
      match lt {
        case Nil =>
          calc == {
            list_lower_bound(List_Concat(listLeft, Cons(x, listRight)), d);
              { assert listLeft == Cons(lh, List.Nil); }
            list_lower_bound(List_Concat(Cons(lh, List.Nil), Cons(x, listRight)), d);
              { assert List_Concat(Cons(lh, List.Nil), Cons(x, listRight))
                    == Cons(lh, List_Concat(List.Nil, Cons(x, listRight))); }
            list_lower_bound(Cons(lh, List_Concat(List.Nil, Cons(x, listRight))), d);
              { Lemma_ListConcatFirstListEmpty(List.Nil, Cons(x, listRight)); }
            list_lower_bound(Cons(lh, Cons(x, listRight)), d);
              { assert d <= x; }
              { assert list_lower_bound(listRight, d); }
            true;
          }
        case Cons(tah, tat) =>
          calc ==> {
            list_lower_bound(List_Concat(listLeft, Cons(x, listRight)), d);
              { assert listLeft == Cons(lh, lt); }
            list_lower_bound(List_Concat(Cons(lh, lt), Cons(x, listRight)), d);
              { assert List_Concat(Cons(lh, lt), Cons(x, listRight))
                    == Cons(lh, List_Concat(lt, Cons(x, listRight))); }
            list_lower_bound(Cons(lh, List_Concat(lt, Cons(x, listRight))), d);
              { assert lt == Cons(tah, tat); }
            list_lower_bound(Cons(lh, List_Concat(Cons(tah, tat), Cons(x, listRight))), d);
              { assert (d <= lh && list_lower_bound(List_Concat(Cons(tah, tat), Cons(x, listRight)
    ), d)); }
            d <= lh && list_lower_bound(List_Concat(Cons(tah, tat), Cons(x, listRight)), d);
              { Lemma_ListConcatLowerBound(Cons(tah, tat), Cons(x, listRight), d, x); }
            true;
          }
      }
  }
}
```

Culminadas ambas demostraciones de las propiedades de cotas de la función *BST_InOrder*, resta demostrar el último lema para dar por finalizado el lema que asegura el ordenamiento en el resultado de la función *BST_InOrder*.

3.3.3.3 Propiedad de ordenamiento de la concatenación entre dos listas y un elemento

Nombre del lema: *Lemma_ListConcatWithMidElemOrdering*

Propiedad: dadas dos listas ordenadas y un elemento que sea cota superior de la primera y cota inferior de la segunda, asegura que la concatenación de las dos listas y el elemento en el medio, también esté ordenado.

```
lemma {:induction a, b} Lemma_ListConcatWithMidElemOrdering(a:List<T>, x:T, b:List<T>)
  requires list_increasing(a)
  requires list_increasing(b)
  requires list_lower_bound(b, x)
  requires list_upper_bound(a, x)
  ensures list_increasing(List_Concat(a, Cons(x, b)))
  decreases a, b
{
  match a {
    case Nil =>
      calc == {
        list_increasing(List_Concat(a, Cons(x, b)));
          { assert a == List.Nil; }
        list_increasing(List_Concat(List.Nil, Cons(x, b)));
          { Lemma_ListConcatFirstListEmpty(List.Nil, b); }
        list_increasing(Cons(x, b));
          { assert list_lower_bound(b, x); }
          { assert list_increasing(b); }
        true;
      }
    case Cons(ha, ta) =>
      match ta {
        case Nil =>
          calc == {
            list_increasing(List_Concat(a, Cons(x, b)));
              { assert a == Cons(ha, ta); }
            list_increasing(List_Concat(Cons(ha, ta), Cons(x, b)));
              { assert List_Concat(Cons(ha, ta), Cons(x, b))
                    == Cons(ha, List_Concat(ta, Cons(x, b))); }
            list_increasing(Cons(ha, List_Concat(ta, Cons(x, b))));
              { assert ta == List.Nil; }
            list_increasing(Cons(ha, List_Concat(List.Nil, Cons(x, b))));
              { Lemma_ListConcatFirstListEmpty(List.Nil, b); }
            list_increasing(Cons(ha, Cons(x, b)));
              { assert ha <= x; }
              { assert list_lower_bound(b, x); }
              { assert list_increasing(b); }
            true;
          }
        case Cons(tah, tat) =>
          calc ==> {
            list_increasing(List_Concat(a, Cons(x, b)));
              { assert a == Cons(ha, ta); }
            list_increasing(List_Concat(Cons(ha, ta), Cons(x, b)));
              { assert List_Concat(Cons(ha, ta), Cons(x, b))
                    == Cons(ha, List_Concat(ta, Cons(x, b))); }
            list_increasing(Cons(ha, List_Concat(ta, Cons(x, b))));
              { assert ta == Cons(tah, tat); }
            list_increasing(Cons(ha, List_Concat(Cons(tah, tat), Cons(x, b))));
```

```
50              { assert (ha <= tah && list_increasing(List_Concat(Cons(tah, tat), Cons(x, b)))); }
51            ha <= tah && list_increasing(List_Concat(Cons(tah, tat), Cons(x, b)));
52              { Lemma_ListConcatWithMidElemOrdering(Cons(tah, tat), x, Cons(x, b)); }
53            true;
54          }
55      }
56  }
57  }
```

La demostración de esta propiedad está dividida en tres casos posibles:

1. La primera lista está vacía, entonces por hipótesis se demuestra la propiedad.
2. La segunda lista está vacía, entonces por hipótesis se demuestra la propiedad.
3. Cuando las dos listas contienen algún elemento, se realiza un paso recursivo con la finalidad de iterar sobre las listas hasta llegar a una de las dos condiciones de finalización nombradas anteriormente.

Propiedad de concatenación de dos listas vacías

Nombre del lema: *Lemma_ListConcatBothListEmpty*

Propiedad: asegurar que al aplicarse la concatenación de dos listas vacías, el resultado sea una lista vacía.

```
1  lemma {:induction a, b} Lemma_ListConcatBothListEmpty(a:List<T>, b:List<T>)
2    requires a == List.Nil
3    requires b == List.Nil
4    ensures List_Concat(a, b) == List.Nil
5  {
6    calc == {
7      List_Concat(a, b);
8        { assert a == List.Nil; }
9      List_Concat(List.Nil, b);
10       { assert List_Concat(List.Nil, b) == b; }
11     b;
12       { assert b == List.Nil; }
13     List.Nil;
14   }
15 }
```

Con estas implementaciones, se da por finalizada la demostración de *Lemma_BSTInOrderOrdering* que asegura que la lista resultante de haber sido aplicada la función *BST_InOrder* esté ordenada crecientemente.

En la sección anterior se demostró el *Lemma_BSTLoadOrdering* que asegura que el árbol binario de búsqueda resultante de haber sido aplicada la función *BST_Load* también esta ordenado.

Por lo tanto, al poder asegurar el ordenamiento de las estructuras de datos resultantes de haberse aplicado las funciones *BST_Load* y *BST_InOrder*, se finaliza la demostración del *Lemma_TreeSortOrdering* que asegura que el resultado de aplicar la función *TreeSort* sea una lista ordenada creciente.

4 Programación Imperativa en Dafny

En este capítulo se abordarán algoritmos conocidos de búsqueda y ordenamiento que servirán como ejemplo del potencial de Dafny para asegurar la corrección del algoritmo implementado en métodos imperativos.

Es importante definir correctamente la cláusula *ensures* en los métodos porque el compilador de Dafny verificará el código para saber el cumplimiento o no de dichas cláusulas.

Luego se aplicarán invariantes que ayudarán al compilador a demostrar que el método cumple las cláusulas de *ensures* que fueron declaradas.

Primero se definirán ciertos predicados necesarios para determinar si un *array* de elementos está ordenado. Será fundamental para el análisis de los siguientes algoritmos.

Predicados de ordenamiento

Nombre del predicado: *sorted*

Predicado: retorna verdadero si un *array* de enteros está ordenado.

```
1 predicate sorted(A:array<int>)
2     reads A
3 {
4     sorted_between(A, 0, A.Length-1)
5 }
```

Para realizar este predicado, se utiliza otro predicado más genérico que retorna verdadero si el *array* desde la posición *from* hasta la posición *to* está ordenado.

```
predicate sorted_between(A:array<int>, from:int, to:int)
    reads A
{
    forall i, j :: 0 <= i <= j < A.Length && from <= i <= j <= to ==> A[i] <= A[j]
}
```

4.1 Algoritmos de Búsqueda

4.1.1 Algoritmo *LinearSearch*

Nombre del método: *LinearSearch*

Funcionalidad: empieza al comienzo del *array* e itera hasta encontrar el elemento buscado "*key*" retornando el índice donde lo encontró.

```
1  method LinearSearch(A:array<int>, key:int) returns (index:int)
2    ensures 0 <= index ==> index < A.Length && A[index] == key
3    ensures index < 0 ==> key !in A[..]
4  {
5    var N := A.Length;
6    var i := 0;
7    while i < N
8      invariant 0 <= i <= N
9      invariant forall k :: 0 <= k < i ==> A[k] != key
10     decreases N - i
11   {
12     if A[i] == key
13     {
14       return i;
15     }
16     i := i + 1;
17   }
18   return -1;
19 }
```

En la linea 9, se puede ver definida una invariante que comenta que ningún elemento tal que su índice sea menor al actual puede ser el elemento buscado.

4.1.2 Algoritmo *BinarySearch*

Funcionalidad: se divide el intervalo de búsqueda a la mitad repetitivamente, según el valor del elemento buscado y el valor del elemento que se encuentra en la mitad del intervalo. Si el valor buscado es mayor, se toma la primer mitad del intervalo, de lo contrario se toma la segunda mitad.

```
1  method BinarySearch(A:array<int>, key:int) returns (index:int)
2    requires sorted(A)
3    ensures 0 <= index ==> index < A.Length && A[index] == key
4    ensures index < 0 ==> key !in A[..]
5  {
6    var N := A.Length;
7    var low := 0;
8    var high := N;
9    while low < high
10     invariant 0 <= low <= high <= N
11     invariant key !in A[..low]
12     invariant key !in A[high..]
13     decreases high - low
14   {
15     var mid := (low + high) / 2;
16     if key < A[mid] {
17         high := mid;
18     } else if key > A[mid] {
19         low := mid + 1;
20     } else {
21         return mid;
22     }
23   }
24   return -1;
25 }
```

Como dato importante, el funcionamiento del algoritmo tiene como requerimiento que el *array* donde se busca el elemento, esté ordenado. Esto se especifica en la linea 2.

Como aparece en las lineas 3 y 4, si el índice retornado es mayor a cero, es porque existe un elemento tal que su posición en el *array* es la del índice retornado. Por otro lado, si se retorna un número negativo, es porque no existe ningún índice tal que su posición en el *array* retorne al elemento buscado.

Se realizan tres invariantes para poder asegurar que se cumplan las cláusulas *ensures* definidas.

La primer invariante, acota el rango de búsqueda.

Las invariantes de la línea 11 y 12, dejan en claro que el valor buscado no puede estar en un índice que esté a la izquierda de *low* o a la derecha de *high*. Esto es así por razones inherentes al algoritmo, ya que dependiendo del valor buscado y el valor del medio, se itera hacia un lado o hacia el otro.

En caso de que no exista el elemento buscado en el *array*, se retornará -1 como índice para respetar la cláusula *ensures* de la linea 4.

4.2 Algoritmos de ordenamiento

4.2.1 Algoritmo *InsertionSort*

Nombre del método: *InsertionSort*

Funcionalidad: ordena el *array* desde el extremo izquierdo formando un grupo ordenado de elementos desde el índice hacia su izquierda. En cada iteración, va agregando un nuevo elemento a dicho grupo.

```
1  method InsertionSort(A:array<int>)
2    modifies A
3    requires A.Length >= 1
4    ensures multiset(A[..]) == multiset(old(A[..]))
5    ensures sorted(A)
6  {
7    var N := A.Length;
8    var i := 1;
9    while i < N
10     invariant 1 <= i <= N
11     invariant multiset(A[..]) == multiset(old(A[..]))
12     invariant sorted_between(A, 0, i-1)
13     decreases N-i
14   {
15     var j := i;
16     while j > 0 && A[j-1] > A[j]
17       invariant 1 <= i <= N-1
18       invariant 0 <= j <= i
19       invariant multiset(A[..]) == multiset(old(A[..]))
20       invariant forall m, n :: 0 <= m < n < i+1 && n != j ==> A[m] <= A[n]
21       decreases j
22     {
23       A[j], A[j-1] := A[j-1], A[j];
24       j := j-1;
25     }
26     i := i+1;
27   }
28 }
```

Para asegurar que el algoritmo haya realizado el ordenamiento *InsertionSort* correctamente, es fundamental poder asegurar que el *array* esté ordenado y se haya mantenido la integridad de sus elementos al finalizar el algoritmo (linea 4 y 5).

Para cumplir las cláusulas *ensures*, es necesario agregar ciertas invariantes en las dos iteraciones. En la linea 20, se muestra como el *array* debe estar ordenado para cada par de elementos excepto para los que el índice del segundo elemento sea igual a "j". También es importante ver como el *array* debe estar ordenado entre los elementos que vienen quedando a la izquierda del índice actual (por razones inherentes a la lógica del *InsertionSort*).

Por otro lado, para verificar la integridad de los elementos, se agregan dos invariantes que aseguran que los elementos no hayan sido modificados en cada una de las iteraciones. Esto se puede ver en las lineas 11 y 19.

4.2.2 Algoritmo *BubbleSort*

Funcionalidad: ordena el *array* de elementos que recibe por parámetro utilizando como mecanismo la comparación de pares consecutivos de elementos. Cuando el elemento actual es mayor al elemento siguiente, se intercambian posiciones.

```
1  method BubbleSort(A:array<int>)
2    modifies A
3    ensures sorted(A)
4    ensures multiset(A[..]) == multiset(old(A[..]))
5  {
6    var N := A.Length;
7    var i := N-1;
8    while 0 < i
9      invariant multiset(A[..]) == multiset(old(A[..]))
10     invariant sorted_between(A, i, N-1)
11     invariant forall n, m :: 0 <= n <= i < m < N ==> A[n] <= A[m]
12     decreases i
13   {
14     var j := 0;
15     while j < i
16       invariant 0 < i < N
17       invariant 0 <= j <= i
18       invariant multiset(A[..]) == multiset(old(A[..]))
19       invariant sorted_between(A, i, N-1)
20       invariant forall n, m :: 0 <= n <= i < m < N ==> A[n] <= A[m]
21       invariant forall n :: 0 <= n <= j ==> A[n] <= A[j]
22       decreases i - j
23     {
24       if A[j] > A[j+1]
25       {
26           A[j], A[j+1] := A[j+1], A[j];
27       }
28       j := j+1;
29     }
30     i := i-1;
31   }
32 }
```

En las lineas 9 y 18, se verifica que en cada iteración, se mantiene la integridad de los elementos.

En las lineas 11 y 20, la invariante verifica que el *array* “A” esté ordenado para cada par de elementos tal que el primer elemento pertenezca a la partición izquierda de “i” y el segundo elemento pertenezca a la partición derecha de “i”.

En la linea 21, la invariante verifica que existe un supremo definido por el valor que toma el *array* en la posición "j". Por lo tanto, cada valor que toma el *array* para todos los elementos desde 0 hasta "j" son menores o iguales al valor del supremo.

5 Equivalencias entre diferentes implementaciones de Fibonacci

Este capítulo mostrará diferentes implementaciones para obtener el n-ésimo número de la sucesión de Fibonacci.

También se demostrará mediante lemas, que para todo *n*, estas implementaciones son equivalentes a la implementación base *Fibonacci_Recursive*.

5.1 Algoritmo Fibonacci recursivo

Nombre de la función: *Fibonacci_Recursive*

```
1 function method Fibonacci_Recursive(n: nat): nat
2   decreases n
3 {
4   if (n == 0) then 0 else
5   if (n == 1) then 1 else
6   Fibonacci_Recursive(n-2) + Fibonacci_Recursive(n-1)
7 }
```

Clásica implementación de Fibonacci de forma recursiva.

Esta implementación será tomada en cuenta como base para comparar con el resto de las técnicas que se aplicarán a continuación.

5.2 Algoritmo Fibonacci *tail recursive*

Nombre de la función: *Fibonacci_TailRecursive*

Aclaración: se debe realizar la primer llamada con *a=0* y *b=1*.

```
1  function method Fibonacci_TailRecursive(n: nat, a: nat, b: nat): nat
2    decreases n
3  {
4    if (n == 0) then a else
5    Fibonacci_TailRecursive(n-1, b, a+b)
6  }
```

5.2.1 Propiedad de equivalencia entre Fibonacci *tail recursive* y Fibonacci recursivo

Nombre del lema: *Lemma_FibonacciTailRecursiveEqualsFibonacciRecursive*

Propiedad: asegurar para todo n, que la función *Fibonacci_TailRecursive* retorna el mismo resultado que la función *Fibonacci_Recursive*.

```
1  lemma {:induction n, i} Lemma_FibonacciTailRecursiveEqualsFibonacciRecursive(n: nat, i: nat)
2    requires 0 <= n
3    requires 0 <= i <= n
4    ensures Fibonacci_TailRecursive(n-i, Fibonacci_Recursive(i), Fibonacci_Recursive(i+1)) ==
       Fibonacci_Recursive(n)
5    decreases n-i
6  {
7    if (n-i == 0) {
8      calc == {
9        Fibonacci_TailRecursive(n-i, Fibonacci_Recursive(i), Fibonacci_Recursive(i+1));
10         { assert Fibonacci_TailRecursive(n-i, Fibonacci_Recursive(i), Fibonacci_Recursive(i+1))
11               == Fibonacci_Recursive(i); }
12       Fibonacci_Recursive(n);
13     }
14   } else {
15     calc == {
16       Fibonacci_TailRecursive(n-i, Fibonacci_Recursive(i), Fibonacci_Recursive(i+1));
17         { assert Fibonacci_TailRecursive(n-i, Fibonacci_Recursive(i), Fibonacci_Recursive(i+1))
18               == Fibonacci_TailRecursive(n-i-1, Fibonacci_Recursive(i+1), Fibonacci_Recursive(i) +
       Fibonacci_Recursive(i+1)); }
19       Fibonacci_TailRecursive(n-i-1, Fibonacci_Recursive(i+1), Fibonacci_Recursive(i) +
      Fibonacci_Recursive(i+1));
20         { assert Fibonacci_Recursive(i) + Fibonacci_Recursive(i+1)
21               == Fibonacci_Recursive(i+2); }
22       Fibonacci_TailRecursive(n-i-1, Fibonacci_Recursive(i+1), Fibonacci_Recursive(i+2));
23         { Lemma_FibonacciTailRecursiveEqualsFibonacciRecursive(n, i+1); }
24       Fibonacci_Recursive(n);
25     }
26   }
27 }
```

En la línea 19, se muestra como se aplica la definición de *Fibonacci_TailRecursive* quedando una suma de dos términos en el tercer parámetro.

Para respetar la definición de *Fibonacci_Recursive*, es importante llevar el tercer parámetro a una sola función que sea el elemento siguiente en la sucesión de Fibonacci que la que aparece en el segundo parámetro.

Por esa razón, en la linea 22, se agrupan los valores de la suma de ambas funciones del tercer parámetro, para que quede simplificado y se pueda respetar la definición de *Fibonacci_Recursive*.

Realizado esto, se llega a la hipótesis de inducción llamando recursivamente al lema para completar el paso inductivo.

5.3 Algoritmo Fibonacci recursivo con par ordenado

Nombre de la función: *Fibonacci_RecursivePair*

```
1 function method Fibonacci_RecursivePair(n: nat): nat
2 {
3   match Fibonacci_RecursivePairAux(n) {
4     case (a, b) => a
5   }
6 }
```

Para poder realizar la función de Fibonacci utilizando un par ordenado de elementos, se necesitará una función auxiliar.

Dicha función auxiliar siempre retornará el número de Fibonacci solicitado en la primer posición del par de elementos y el número de Fibonacci siguiente al solicitado en la segunda posición.

Por esa razón, esta función siempre retorna el primer elemento del par ordenado.

Algoritmo auxiliar Fibonacci recursivo en pareja

Nombre de la función: *Fibonacci_RecursivePairAux*

```
1 function method Fibonacci_RecursivePairAux(n: nat): (nat, nat)
2   decreases n
3 {
4   if (n == 0) then (0, 1) else
5   match Fibonacci_RecursivePairAux(n-1) {
6     case (a, b) => (b, a+b)
```

```
  }
}
```

La implementación es similar a la solución *tail recursive* con la diferencia que en vez de utilizar parámetros en la llamada recursiva, aquí se utiliza un par ordenado de elementos.

5.3.1 Propiedad de equivalencia entre Fibonacci recursivo en pareja y Fibonacci recursivo

Nombre del lema: *Lemma_FibonacciRecursivePairEqualsFibonacciRecursive*

Propiedad: asegurar para todo *n*, que la función *Fibonacci_RecursivePair* retorna un resultado equivalente a la función *Fibonacci_Recursive*.

```
lemma {:induction n} Lemma_FibonacciRecursivePairEqualsFibonacciRecursive(n: nat)
  ensures Fibonacci_RecursivePair(n) == Fibonacci_Recursive(n)
{
  calc == {
    Fibonacci_RecursivePair(n);
    { Lemma_FibonacciRecursivePairAuxEqualsFibonacciRecursive(n); }
      { assert Fibonacci_RecursivePairAux(n)
            == (Fibonacci_Recursive(n), Fibonacci_Recursive(n+1)); }
    Fibonacci_Recursive(n);
  }
}
```

Para poder probar esta propiedad, es necesario que se cumpla otra propiedad similar que involucra a su función auxiliar.

Una vez probada dicha propiedad a continuación, se puede ver que la demostración es directa.

5.3.2 Propiedad de equivalencia entre Fibonacci recursivo en pareja auxiliar y Fibonacci recursivo

Nombre del lema: *Lemma_FibonacciRecursivePairAuxEqualsFibonacciRecursive*

Propiedad: asegurar para todo *n*, que la función *Fibonacci_RecursivePairAux* retorna como resultado un par ordenado con el primer elemento equivalente a *Fibonacci_Recursive* de *n* y el segundo elemento equivalente a *Fibonacci_Recursive* de *n+1*.

```
lemma {:induction n} Lemma_FibonacciRecursivePairAuxEqualsFibonacciRecursive(n: nat)
  ensures Fibonacci_RecursivePairAux(n) == (Fibonacci_Recursive(n), Fibonacci_Recursive(n+1))
  decreases n
{
  if (n == 0) {
    calc == {
      Fibonacci_RecursivePairAux(n);
        { assert n == 0; }
      Fibonacci_RecursivePairAux(0);
        { assert Fibonacci_RecursivePairAux(0) == (0, 1); }
      (0, 1);
        { assert Fibonacci_Recursive(0) == 0; }
        { assert Fibonacci_Recursive(0+1) == 1; }
        { assert (0, 1) == (Fibonacci_Recursive(0), Fibonacci_Recursive(0+1)); }
      (Fibonacci_Recursive(0), Fibonacci_Recursive(0+1));
    }
  } else {
    calc == {
      Fibonacci_RecursivePairAux(n);
        { assert n > 0; }
    }
    match Fibonacci_RecursivePairAux(n-1) {
      case (a, b) =>
        calc == {
          (b, a+b);
          { Lemma_FibonacciRecursivePairAuxEqualsFibonacciRecursive(n-1); }
        }
    }
  }
}
```

5.4 Algoritmo Fibonacci iterativo

Nombre del método: *Fibonacci_Iterative*

Propósito: mostrar una implementación auto-verificable de Fibonacci realizada de forma iterativa.

```
method Fibonacci_Iterative(n: nat) returns (a: nat)
  ensures a == Fibonacci_Recursive(n)
{
  a := 0;
  var b: nat := 1;
  var i: nat := 0;

  while i < n
    invariant 0 <= i <= n
    invariant a == Fibonacci_Recursive(i)
    invariant b == Fibonacci_Recursive(i+1)
    decreases n-i
  {
    a, b := b, a+b;
    i := i+1;
  }
```

```
17 }
```

Como se puede observar en la linea 2, se asegura que el resultado es equivalente al de la función *Fibonacci_Recursive*.

Para poder demostrar la cláusula de *ensures*, se utilizan tres invariantes en el *loop*.

Una de ellas delimita el alcance de la variable "i", mientras que las otras dos son necesarias para demostrar que en cada iteración *a=Fibonacci_Recursive(i)* y *b=Fibonacci_Recursive(i+1)*.

Las operaciones dentro del *loop* conservan las invariantes, por lo tanto al finalizar el *loop*, queda comprobado que en efecto la cláusula de *ensures* se cumple.

6 Dafny como herramienta de producción

En los capítulos anteriores, se han realizado demostraciones de lemas de una manera muy descriptiva ya que han sido pensadas para un entorno educativo.

En esta sección, se quiere mostrar el potencial de Dafny como herramienta productiva. Se verá que Dafny permite escribir las demostraciones de los lemas, de manera más eficiente y corta. Esto es debido a su potente verificador de código.

Se estarán repasando lemas realizados en los capítulos anteriores, con modificaciones en sus implementaciones, que demuestren lo comentado anteriormente.

6.1 Optimizaciones en los lemas de *TreeSort*

```
1  lemma Lemma_TreeSortIntegrity(list:List<T>)
2    ensures List_ToMultiset(TreeSort(list)) == List_ToMultiset(list)
3  {
4    calc == {
5      List_ToMultiset(TreeSort(list));
6        { Lemma_BSTInOrderIntegrity(BST_Load(list)); }
7        { Lemma_BSTLoadIntegrity(list); }
8      List_ToMultiset(list);
9    }
10 }
```

```
1  lemma Lemma_TreeSortOrdering(list:List<T>)
2    ensures list_increasing(TreeSort(list))
3  {
4    calc == {
5      list_increasing(TreeSort(list));
6        { Lemma_BSTLoadOrdering(list); }
7        { Lemma_BSTInOrderOrdering(BST_Load(list)); }
```

```
    true;
  }
}
```

Para estos casos, sólo se pudo omitir las instrucciones de assert y alguna otra instrucción con un paso muy simple.

6.2 Optimizaciones en los lemas de *BST*

6.2.1 Propiedades de la inserción de elementos

```
lemma {:induction tree} Lemma_BSTInsertOrdering(tree:BST<T>, d:T)
  requires bst_ordered(tree)
  ensures bst_ordered(BST_Insert(tree, d))
  decreases tree
{
  match tree {
    case Nil =>
      calc == {
        bst_ordered(BST_Insert(BST.Nil, d));
        bst_ordered(Node(BST.Nil, d, BST.Nil));
        true;
      }
    case Node(left, x, right) =>
      calc == {
        bst_ordered(BST_Insert(tree, d));
        bst_ordered(BST_Insert(Node(left, x, right), d));
        { if (d < x) {
          calc == {
            bst_ordered(Node(BST_Insert(left, d), x, right));
              { Lemma_BSTInsertUpperBound(left, d, x); }
              { Lemma_BSTInsertOrdering(left, d); }
            bst_ordered(Node(BST.Nil, d, BST.Nil));
          }
        } else {
          calc == {
            bst_ordered(Node(left, x, BST_Insert(right, d)));
              { Lemma_BSTInsertLowerBound(right, d, x); }
              { Lemma_BSTInsertOrdering(right, d); }
            bst_ordered(Node(BST.Nil, d, BST.Nil));
          }
        } }
        true;
      }
  }
}
```

Para este lema en particular, no se pudo omitir demasiadas instrucciones ya que el compilador necesitó más ayuda para poder demostrar la propiedad que se asegura.

```
lemma {:induction tree} Lemma_BSTInsertIntegrity(tree:BST<T>, d:T)
  ensures BST_ToMultiset(BST_Insert(tree, d)) == BST_ToMultiset(tree) + multiset{d}
  decreases tree
{ }
```

```
lemma {:induction tree} Lemma_BSTInsertUpperBound(tree:BST<T>, d:T, b:T)
  requires bst_ordered(tree)
  requires bst_upper_bound(tree, b)
  requires b > d
  ensures bst_upper_bound(BST_Insert(tree, d), b)
  decreases tree
{ }
```

```
lemma {:induction tree} Lemma_BSTInsertLowerBound(tree:BST<T>, d:T, b:T)
  requires bst_ordered(tree)
  requires bst_lower_bound(tree, b)
  requires b <= d
  ensures bst_lower_bound(BST_Insert(tree, d), b)
  decreases tree
{ }
```

No fue necesario demostrar estos tres lemas porque el compilador de Dafny verificó que se cumple la cláusula *ensures* automáticamente.

6.2.2 Propiedades del ordenamiento transversal *in order*

```
lemma {:induction tree} Lemma_BSTInOrderIntegrity(tree:BST<T>)
  ensures BST_ToMultiset(tree) == List_ToMultiset(BST_InOrder(tree))
  decreases tree
{
  match tree {
    case Nil =>
    case Node(left, x, right) =>
      calc == {
        List_ToMultiset(BST_InOrder(Node(left, x, right)));
        List_ToMultiset(List_Concat(BST_InOrder(left), Cons(x, BST_InOrder(right))));
          { Lemma_ListConcatIntegrity(BST_InOrder(left), Cons(x, BST_InOrder(right))); }
        List_ToMultiset(BST_InOrder(left)) + List_ToMultiset(Cons(x, BST_InOrder(right)));
        List_ToMultiset(BST_InOrder(left)) + List_ToMultiset(Cons(x, List.Nil)) + List_ToMultiset(
    BST_InOrder(right));
        List_ToMultiset(BST_InOrder(left)) + multiset{x} + List_ToMultiset(List.Nil) +
    List_ToMultiset(BST_InOrder(right));
        List_ToMultiset(BST_InOrder(left)) + multiset{x} + List_ToMultiset(BST_InOrder(right));
          { Lemma_BSTInOrderIntegrity(left); }
          { Lemma_BSTInOrderIntegrity(right); }
        BST_ToMultiset(tree);
      }
  }
}
```

```
lemma {:induction tree} Lemma_BSTInOrderOrdering(tree:BST<T>)
  requires bst_ordered(tree)
  ensures list_increasing(BST_InOrder(tree))
{
  match tree {
    case Nil =>
    case Node(left, x, right) =>
      calc == {
        list_increasing(BST_InOrder(Node(left, x, right)));
        list_increasing(List_Concat(BST_InOrder(left), Cons(x, BST_InOrder(right))));
          { Lemma_BSTInOrderLowerBound(right, x); }
          { Lemma_BSTInOrderUpperBound(left, x); }
          { Lemma_ListConcatWithMidElemOrdering(BST_InOrder(left), x, BST_InOrder(right)); }
        true;
      }
  }
}
```

```
lemma {:induction tree} Lemma_BSTInOrderUpperBound(tree:BST<T>, d:T)
  requires bst_ordered(tree)
  requires bst_upper_bound(tree, d)
  ensures list_upper_bound(BST_InOrder(tree), d)
{
  match tree {
    case Nil =>
    case Node(left, x, right) =>
      calc == {
        list_upper_bound(BST_InOrder(Node(left, x, right)), d);
        list_upper_bound(List_Concat(BST_InOrder(left), Cons(x, BST_InOrder(right))), d);
          { Lemma_ListConcatUpperBound(BST_InOrder(left), BST_InOrder(right), d, x); }
        true;
      }
  }
}
```

```
lemma {:induction tree} Lemma_BSTInOrderLowerBound(tree:BST<T>, d:T)
  requires bst_ordered(tree)
  requires bst_lower_bound(tree, d)
  ensures list_lower_bound(BST_InOrder(tree), d)
{
  match tree {
    case Nil =>
    case Node(left, x, right) =>
      calc == {
        list_lower_bound(BST_InOrder(Node(left, x, right)), d);
        list_lower_bound(List_Concat(BST_InOrder(left), Cons(x, BST_InOrder(right))), d);
          { Lemma_ListConcatLowerBound(BST_InOrder(left), BST_InOrder(right), d, x); }
        true;
      }
  }
}
```

En estos lemas sólo se pudieron omitir instrucciones *assert* y las demostraciones de cada uno de los casos bases.

6.2.3 Propiedades de la carga de elementos

```
lemma {:induction list} Lemma_BSTLoadIntegrity(list:List<T>)
  ensures BST_ToMultiset(BST_Load(list)) == List_ToMultiset(list)
  decreases list
{
  match list {
    case Nil =>
    case Cons(head, tail) =>
      calc == {
        BST_ToMultiset(BST_Load(Cons(head, tail)));
        BST_ToMultiset(BST_Insert(BST_Load(tail), head));
          { Lemma_BSTInsertIntegrity(BST_Load(tail), head); }
        BST_ToMultiset(BST_Load(tail)) + multiset{head};
          { Lemma_BSTLoadIntegrity(tail); }
        List_ToMultiset(list);
      }
  }
}
```

```
lemma {:induction list} Lemma_BSTLoadOrdering(list:List<T>)
  ensures bst_ordered(BST_Load(list))
  decreases list
{
  match list {
    case Nil =>
    case Cons(head, tail) =>
      calc == {
        bst_ordered(BST_Load(Cons(head, tail)));
        bst_ordered(BST_Insert(BST_Load(tail), head));
          { Lemma_BSTInsertOrdering(BST_Load(tail), head); }
        true;
      }
  }
}
```

Al igual que en el caso anterior, para estos lemas sólo se pudieron omitir instrucciones *assert* y las demostraciones de cada uno de los casos bases.

6.3 Optimizaciones en los lemas de *List*

6.3.1 Propiedades de la inserción de elementos

En el capítulo educativo se había utilizado sólo un lema para la inserción de elementos. En este caso, dicho lema no fue necesario utilizarlo:

- *Lemma_ListInsertIntegrity*

6.3.2 Propiedades de la concatenación de listas

```
lemma {:induction a} Lemma_ListConcatIntegrity(a:List<T>, b:List<T>)
  ensures List_ToMultiset(List_Concat(a, b)) == List_ToMultiset(a) + List_ToMultiset(b)
  decreases a, b
{ }
```

```
lemma {:induction a, b} Lemma_ListConcatWithMidElemOrdering(a:List<T>, x:T, b:List<T>)
  requires list_increasing(a)
  requires list_increasing(b)
  requires list_lower_bound(b, x)
  requires list_upper_bound(a, x)
  ensures list_increasing(List_Concat(a, Cons(x, b)))
  decreases a, b
{ }
```

```
lemma {:induction listLeft} Lemma_ListConcatUpperBound(listLeft:List<T>, listRight:List<T>, d:T, x
    :T)
  requires list_upper_bound(listLeft, d)
  requires list_upper_bound(listRight, d)
  requires d >= x
  ensures list_upper_bound(List_Concat(listLeft, Cons(x, listRight)), d)
  decreases listLeft
{ }
```

```
lemma {:induction listLeft} Lemma_ListConcatLowerBound(listLeft:List<T>, listRight:List<T>, d:T, x
    :T)
  requires list_lower_bound(listLeft, d)
  requires list_lower_bound(listRight, d)
  requires d <= x
  ensures list_lower_bound(List_Concat(listLeft, Cons(x, listRight)), d)
  decreases listLeft
{ }
```

Como se puede notar, no fue necesario implementar ninguna demostración para estos cuatro lemas porque el compilador de Dafny verificó que se cumplen las cláusulas *ensures* automáticamente.

Además, como consecuencia de no haber sido necesario demostrar los lemas anteriores, tampoco fueron necesarios utilizar los siguientes lemas:

- *Lemma_ListConcatBothListEmpty*
- *Lemma_ListConcatFirstListEmpty*

6.4 Optimizaciones en los lemas de Fibonacci

```
1  lemma {:induction n, i} Lemma_FibonacciTailRecursiveEqualsFibonacciRecursive(n: nat, i: nat)
2    requires 0 <= n
3    requires 0 <= i <= n
4    ensures Fibonacci_TailRecursive(n-i, Fibonacci_Recursive(i), Fibonacci_Recursive(i+1)) ==
       Fibonacci_Recursive(n)
5    decreases n-i
6  {
7    if (n-i > 0) {
8      calc == {
9        Fibonacci_TailRecursive(n-i, Fibonacci_Recursive(i), Fibonacci_Recursive(i+1));
10       Fibonacci_TailRecursive(n-i-1, Fibonacci_Recursive(i+1), Fibonacci_Recursive(i+2));
11         { Lemma_FibonacciTailRecursiveEqualsFibonacciRecursive(n, i+1); }
12       Fibonacci_Recursive(n);
13     }
14   }
15 }
```

Para este lema, sólo se pudo omitir la demostración del caso base y las instrucciones *assert*.

```
1  lemma {:induction n} Lemma_FibonacciRecursivePairEqualsFibonacciRecursive(n: nat)
2    ensures Fibonacci_RecursivePair(n) == Fibonacci_Recursive(n)
3  { }
```

```
1  lemma {:induction n} Lemma_FibonacciRecursivePairAuxEqualsFibonacciRecursive(n: nat)
2    ensures Fibonacci_RecursivePairAux(n) == (Fibonacci_Recursive(n), Fibonacci_Recursive(n+1))
3    decreases n
4  { }
```

Para estos dos últimos lemas, no fue necesario implementar ninguna demostración gracias al verificador de código de Dafny.

7 Conclusiones

La conclusión que se estará comentando a continuación, está basada en mi experiencia personal y en lecturas de referencias bibliográficas que he estado recopilando a lo largo de este proyecto (20 semanas).

Habrá una conclusión para el módulo educativo y otra para el módulo productivo, ya que los objetivos establecidos eran diferentes.

Módulo educativo

Como se pudo observar a lo largo de la tesis, mediante el uso de la instrucción *calc*, se pudieron realizar demostraciones muy claras de todas las propiedades que fueron necesarias. Se sintió muy natural la forma de demostrar, muy similar a las demostraciones formales de código en lápiz y papel.

Dafny permite verificar diferentes paradigmas de programación (funcional e iterativo), siendo ambos fundamentales para la carrera de un estudiante.

Cuenta con una sintaxis "amigable", similar a lenguajes *.NET* o *Haskell*, brindando un muy completo abanico de posibilidades para realizar demostraciones.

Un estudiante está acostumbrado a verificar sus códigos mediante la realización de pruebas. Esta metodología nueva "auto verificable" sin dudas aportaría otro enfoque para ser tenido en cuenta.

Se introduciría el concepto de invariantes, algo que no es dado hasta el momento en las materias curriculares de la carrera de Ingeniería de Sistemas en ORT y creo que merece ser tenido en cuenta para formar el pensamiento del estudiante a la hora de resolver un problema.

Con normalidad, la verificación de código que realiza el compilador, suele demorar unos pocos segundos en mostrar en pantalla si está correcto o tiene algún

error. Por otro lado, esporádicamente me ha ocurrido que en la verificación de código de algún lema, el compilador no finaliza y se debe parar el proceso del compilador (*ctrl+c* en la terminal). Estos casos pueden generar confusión al estudiante, ya que el verificador no muestra en pantalla si el programa está correcto o hay un error, sino que queda en *loop* por un tiempo prolongado. La solución es ir retirando instrucciones y volviendo a compilar hasta llegar a una respuesta rápida del compilador. Desde ese punto, se debe realizar los pasos nuevamente de diferente manera porque se debe asumir que se tenía un error que causaba dicho *loop*.

Como conclusión, siento que es recomendable incorporar al menos el concepto de invariantes en la carrera de Ingeniería de Sistemas en ORT.

Como adicional, me parecería valioso e interesante, mostrar demostraciones en Dafny que complementen las demostraciones que se realizan en los primeros cursos de la carrera.

También le veo potencial para ser utilizado como lenguaje en una electiva, principalmente para introducir los conceptos nombrados anteriormente que serán de utilidad a los perfiles más teóricos de estudiantes. Esta electiva podría ser una continuación de la materia "Estructura de Datos y Algoritmos 2".

Módulo productivo

Existen ciertos factores que deben ser tomados en cuenta a la hora de seleccionar un lenguaje para realizar un software productivo. Yo tomaré cuatro factores que considero fundamentales para valorar a Dafny como herramienta de software productivo.

- *Plataforma de destino*: Dafny al ser compilado a ".dll", puede ser utilizado en las mismas plataformas que se utilizan otros lenguajes .NET hoy en día.
- *Bibliotecas*: este factor es fundamental para la optimización de tiempo. Es normal en un entorno productivo, el uso de bibliotecas que solucionan problemas comunes y que tienen el respaldo de toda la comunidad. Dafny en este sentido, siento que tiene una falencia en cantidad de bibliotecas.
- *Comunidad*: un desarrollador constantemente realiza consultas por internet y se apoya en ejemplos para realizar su labor. En este sentido, siento que

Dafny no ha podido construir aún una comunidad numerosa. Personalmente, realicé una consulta en el foro principal de Dafny sobre el comportamiento del compilador y sólo obtuve una respuesta. Inclusive viendo el historial del foro, no se suelen hacer demasiadas consultas.

- *Destrezas del equipo*: se debe tener en cuenta que para desarrollar en Dafny, el equipo debe tener una alta preparación en la programación de invariantes y conocer fundamentos sobre la demostración de lemas. Esto puede acotar bastante el subconjunto de técnicos capacitados para este tipo de trabajos.

Por lo tanto, la conclusión que llego para este módulo es que Dafny puede aplicarse como herramienta de producción para cierto "nicho" de mercado. Ese "nicho" debe tener como factor fundamental la necesidad de que el software sea de precisión absoluta en su comportamiento.

Al compilar un archivo Dafny (.dfy), se obtiene como resultado un archivo biblioteca de .NET denominado ".dll", a través de un código intermediario C#. Esa particularidad puede ser utilizada para realizar cierta biblioteca que deba ser muy precisa y a prueba de errores, en un sistema que está implementado en otras tecnologías .NET.

8 Bibliografía

[1] D. Zingaro, *Invariants: A Generative Approach to Programming*, 2008.

[2] A. Milieris and E. Fogel, *Programación basada en invariantes: un enfoque didáctico*, 2016.

[3] W. Sonnex and S. Drossopoulou, "Verified programming in dafny," 2013. [Online]. Available: http://www.doc.ic.ac.uk/~scd/Dafny_Material/Lectures.pdf

[4] R. R. R. Cuéllar, *Verificación de algoritmos y estructuras de datos en Dafny*, 2016.

[5] M. L. K. Rustan, "Accessible software verification with dafny," *IEEE Software*, vol. 34, no. 6, pp. 94–97, 2017.

[6] ——, "Well-founded functions and extreme predicates in dafny: A tutorial," 2016. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml250.pdf

[7] P. Lucio, *A Tutorial on Using Dafny to Construct Verified Software*, 2016.

[8] "Rise4fun - dafny tutorial," [Jul 27, 2021]. [Online]. Available: https://rise4fun.com/Dafny/tutorial/Guide

[9] K. R. M. Leino, R. L. Ford, and D. R. Cok, *Dafny Reference Manual*, 2021.