Dafny

Demonstration of two Search Algorithms implemented in Dafny

Holzinger Jan-Michael

Seminar Formal Methods

22. Juni 2016

The purpose of this talk is to give (toy-)examples on how to implement and verify algorithms in Dafny. The examples of the Search Algorithms are chosen, as how they work and their properties are widely known. Still they are not just trivial examples.
We will see some Annotations and Keywords highly used in Dafny. Also we will encounter some errors, that will occur while (live-)verification. We will see, how „thinking before coding“ can help, and also, that sometimes these errors also can be helpful in how to develop code, and to not forget parts of an algorithm.

Search Algorithms

Purpose of a Search Algorithm:

Search Algorithms

Purpose of a Search Algorithm:

Given

Search Algorithms

Purpose of a Search Algorithm:

Given

- a Data Structure **Input** (1)

Search Algorithms

Purpose of a Search Algorithm:

Given

- a Data Structure **Input** (1)

(1) in the demonstration this will be an Array of Integers

Search Algorithms

Purpose of a Search Algorithm:

Given

- a Data Structure **Input** (1)
- and a Key **find** (2)

(1) in the demonstration this will be an Array of Integers

Search Algorithms

Purpose of a Search Algorithm:

Given

- a Data Structure **Input** (1)
- and a Key **find** (2)

(1) in the demonstration this will be an Array of Integers

(2) here an Integer

Search Algorithms

Purpose of a Search Algorithm:

Given

- a Data Structure **Input** (1)
- and a Key **find** (2)

Return

(1) in the demonstration this will be an Array of Integers

(2) here an Integer

Search Algorithms

Purpose of a Search Algorithm:

Given

- a Data Structure **Input** (1)
- and a Key **find** (2)

Return

- **i** the position of the Key (3)

(1) in the demonstration this will be an Array of Integers

(2) here an Integer

Search Algorithms

Purpose of a Search Algorithm:

Given

- a Data Structure **Input** (1)
- and a Key **find** (2)

Return

- **i** the position of the Key (3)

(1) in the demonstration this will be an Array of Integers

(2) here an Integer

(3) Typically also an Integer

Search Algorithms

Purpose of a Search Algorithm:

Given

- a Data Structure **Input** (1)
- and a Key **find** (2)

Return

- **i** the position of the Key, if it is contained in the Data Structure.(3)

(1) in the demonstration this will be an Array of Integers

(2) here an Integer

(3) Typically also an Integer

Linear Search

The probably most „natural“ approach is the following:

Linear Search

The probably most „natural“ approach is the following:

Linear Search

Linear Search

The probably most „natural“ approach is the following:

Linear Search

```
Initialize i with 0
```

Linear Search

The probably most „natural“ approach is the following:

Linear Search

```
1 Initialize i with 0
2 Check if the element Input[i] has the desired value find
```

Linear Search

The probably most „natural“ approach is the following:

Linear Search

```
  1 Initialize i with 0
  2 Check if the element Input[i] has the desired value find
yes Return i
```

Linear Search

The probably most „natural“ approach is the following:

Linear Search

```
  1 Initialize i with 0
  2 Check if the element Input[i] has the desired value find
yes Return i
 no As long as we are not at the last element of Input:
```

Linear Search

The probably most „natural“ approach is the following:

Linear Search

```
  1 Initialize i with 0
  2 Check if the element Input[i] has the desired value find
yes Return i
 no As long as we are not at the last element of Input:
    Increase i by 1, Go to Step 2
```

Linear Search

The probably most „natural“ approach is the following:

Linear Search

```
  1 Initialize i with 0
  2 Check if the element Input[i] has the desired value find
yes Return i
 no As long as we are not at the last element of Input:
    Increase i by 1, Go to Step 2
  3 We only come here, when Input does not contain find
```

Linear Search

The probably most „natural“ approach is the following:

Linear Search

```
  1 Initialize i with 0
  2 Check if the element Input[i] has the desired value find
yes Return i
 no As long as we are not at the last element of Input:
    Increase i by 1, Go to Step 2
  3 We only come here, when Input does not contain find
    Return that information
```

Linear Search

The probably most „natural“ approach is the following:

Linear Search

```
  1 Initialize i with 0
  2 Check if the element Input[i] has the desired value find
yes Return i
 no As long as we are not at the last element of Input:
    Increase i by 1, Go to Step 2
  3 We only come here, when Input does not contain find
    Return that information
```

One advantage of this Algorithm is, that it has (almost) no precondition on the Dataset Input. It belongs to the complexity class $\mathcal{O}(n)$

Remember

These are some Dafny concepts/keywords, we will need for this Algorithm.

Remember

These are some Dafny concepts/keywords, we will need for this Algorithm.

`requires`	Used to state Preconditions

Remember

These are some Dafny concepts/keywords, we will need for this Algorithm.

`requires`	Used to state Preconditions
`ensures`	Used to state Postconditions

Remember

These are some Dafny concepts/keywords, we will need for this Algorithm.

`requires`	Used to state Preconditions
`ensures`	Used to state Postconditions
`invariant`	Used to verify Code within a Loop. Needed as Dafny considers Loops „Black-Boxes“ while verification.

Binary Search

Next is a more elevated approach:

Binary Search

Next is a more elevated approach: Suppose the Input Data is already sorted by size.

Binary Search

Next is a more elevated approach: Suppose the Input Data is already sorted by size. Then one could as well apply Linear Search, but (except for special cases) there is a faster way to do this.

Binary Search

Next is a more elevated approach: Suppose the Input Data is already sorted by size. Then one could as well apply Linear Search, but (except for special cases) there is a faster way to do this. Suppose we check the element in the middle of the Input Data.

Binary Search

Next is a more elevated approach: Suppose the Input Data is already sorted by size. Then one could as well apply Linear Search, but (except for special cases) there is a faster way to do this. Suppose we check the element in the middle of the Input Data.Then there are 3 possible results:

Binary Search

Next is a more elevated approach: Suppose the Input Data is already sorted by size. Then one could as well apply Linear Search, but (except for special cases) there is a faster way to do this. Suppose we check the element in the middle of the Input Data.Then there are 3 possible results:

1. We are VERY lucky, and we already found the desired value. But as we have to expect all sizes and kinds of Input, this will be a case we can neglect.

Binary Search

Next is a more elevated approach: Suppose the Input Data is already sorted by size. Then one could as well apply Linear Search, but (except for special cases) there is a faster way to do this. Suppose we check the element in the middle of the Input Data.Then there are 3 possible results:

1. We are VERY lucky, and we already found the desired value. But as we have to expect all sizes and kinds of Input, this will be a case we can neglect.
2. The Key we are looking for is **greater**, then the value of the current element.

Binary Search

Next is a more elevated approach: Suppose the Input Data is already sorted by size. Then one could as well apply Linear Search, but (except for special cases) there is a faster way to do this. Suppose we check the element in the middle of the Input Data.Then there are 3 possible results:

1. We are VERY lucky, and we already found the desired value. But as we have to expect all sizes and kinds of Input, this will be a case we can neglect.
2. The Key we are looking for is **greater**, then the value of the current element.
3. The Key we are looking for is **smaller**, then the value of the current element.

Binary Search

Next is a more elevated approach: Suppose the Input Data is already sorted by size. Then one could as well apply Linear Search, but (except for special cases) there is a faster way to do this. Suppose we check the element in the middle of the Input Data.Then there are 3 possible results:

1. We are VERY lucky, and we already found the desired value. But as we have to expect all sizes and kinds of Input, this will be a case we can neglect.
2. The Key we are looking for is **greater**, then the value of the current element.
3. The Key we are looking for is **smaller**, then the value of the current element.

In the remaining Cases 2 and 3, there is no need to search the first/last half of the Input, as we expect the Input to be sorted, respectively.

Binary Search

Next is a more elevated approach: Suppose the Input Data is already sorted by size. Then one could as well apply Linear Search, but (except for special cases) there is a faster way to do this. Suppose we check the element in the middle of the Input Data.Then there are 3 possible results:

1 We are VERY lucky, and we already found the desired value. But as we have to expect all sizes and kinds of Input, this will be a case we can neglect.
2 The Key we are looking for is **greater**, then the value of the current element.
3 The Key we are looking for is **smaller**, then the value of the current element.

In the remaining Cases 2 and 3, there is no need to search the first/last half of the Input, as we expect the Input to be sorted, respectively. So we cut down the Problem into a Problem of half size.

Binary Search

Next is a more elevated approach: Suppose the Input Data is already sorted by size. Then one could as well apply Linear Search, but (except for special cases) there is a faster way to do this. Suppose we check the element in the middle of the Input Data.Then there are 3 possible results:

1. We are VERY lucky, and we already found the desired value. But as we have to expect all sizes and kinds of Input, this will be a case we can neglect.
2. The Key we are looking for is **greater**, then the value of the current element.
3. The Key we are looking for is **smaller**, then the value of the current element.

In the remaining Cases 2 and 3, there is no need to search the first/last half of the Input, as we expect the Input to be sorted, respectively. So we cut down the Problem into a Problem of half size. And of course we can apply these steps to the remaining Input Data again, and iteratively we get an algorithm. (Of course, the smaller the Input Data, the more likely we directly find the desired Key - if it is contained.)

Remember

These are some Dafny concepts/keywords, we will need for this Algorithm. Additionally to those presented for LinearSearch, these are:

Remember

These are some Dafny concepts/keywords, we will need for this Algorithm. Additionally to those presented for LinearSearch, these are:

`predicate`	A Function, that returns a Boolean Value

Remember

These are some Dafny concepts/keywords, we will need for this Algorithm. Additionally to those presented for LinearSearch, these are:

`predicate`	A Function, that returns a Boolean Value
`reads`	Annotation to allow a function to read (not `modify`) data in a Heap-Allocated Datastructure.