Fundamentals

SFWRENG 2CO3: Data Structures and Algorithms

Jelle Hellings

Department of Computing and Software
McMaster University

Winter 2024

Focus of this course: From hacking to engineering

Engineering is the application of science and mathematics to solve practical problems.

Focus of this course: From hacking to engineering

Engineering is the application of science and mathematics to solve practical problems.

Software engineering requires

- a deep understanding of *what software (programs) do*;
- mastery of a toolbox of *fundamental tools* to tackle programming challenges;
- capability to *analyze* software in depth.

Focus of this course: From hacking to engineering

Engineering is the application of science and mathematics to solve practical problems.

Software engineering requires

- a deep understanding of *what software (programs) do*;
- mastery of a toolbox of *fundamental tools* to tackle programming challenges;
- capability to *analyze* software in depth.

This course introduces the analysis of software by studying and analyzing fundamental tools.

Focus of this course: From hacking to engineering

Engineering is the application of science and mathematics to solve practical problems.

Software engineering requires

- a deep understanding of *what software (programs) do*;
- mastery of a toolbox of *fundamental tools* to tackle programming challenges;
- capability to *analyze* software in depth.

This course introduces the analysis of software by studying and analyzing fundamental tools.

- Analysis of algorithms and data structures: *correctness* and *complexity*.
- Common design strategies for algorithms and data structures.
- A useful toolbox of standard fundamental algorithms and data structures.
- Graph representations and fundamental graph algorithms.

Focus of this course: From hacking to engineering

Engineering is the application of science and mathematics to solve practical problems.

Software engineering requires

- a deep understanding of *what software (programs) do*;
- mastery of a toolbox of *fundamental tools* to tackle programming challenges;
- capability to *analyze* software in depth.

This course introduces the analysis of software by studying and analyzing fundamental tools.

- Analysis of algorithms and data structures: *correctness* and *complexity*.
- Common design strategies for algorithms and data structures.
- A useful toolbox of standard fundamental algorithms and data structures.
- Graph representations and fundamental graph algorithms.

This course is *not* about learning how to program (basic programming is prior knowledge).

Algorithms and data structures

The basic building blocks of any problem that can be solved by a computer program.

Algorithms and data structures

The basic building blocks of any problem that can be solved by a computer program.

Definition (Algorithm)

Procedures for solving problems that are suited for computer implementation.

Algorithms and data structures

The basic building blocks of any problem that can be solved by a computer program.

Definition (Algorithm)

Procedures for solving problems that are suited for computer implementation.

An algorithm takes one-or-more values as input
and produces an output via a *well-defined computational procedure*.

Algorithms and data structures

The basic building blocks of any problem that can be solved by a computer program.

Definition (Algorithm)

Procedures for solving problems that are suited for computer implementation.

An algorithm takes one-or-more values as input
and produces an output via a *well-defined computational procedure*.

Definition (Data structure)

Scheme to store and organize data in order to facilitate *efficient* access and modification.

About Programming Languages

We all have our own favorites.

About Programming Languages

We all have our own favorites.

For the study of data structures and algorithms:
Choice of programming language does *not really* matter (mostly).

About Programming Languages

We all have our own favorites.

For the study of data structures and algorithms:
Choice of programming language does *not really* matter (mostly).

For *optimal* implementations, we sometimes need a lower-level toolbox.
E.g., references or pointers when implementing data structures.

About Programming Languages

We all have our own favorites.

For the study of data structures and algorithms:
Choice of programming language does *not really* matter (mostly).

For *optimal* implementations, we sometimes need a lower-level toolbox.
E.g., references or pointers when implementing data structures.

Many programming languages suffice, e.g.,

- the book has many examples in Java;
- I will provide some examples in C++.

Feel free to experiment in your programming language of choice.

A simple algorithm: Contains

Problem

Given a list L and value v, return $v \in L$.

A simple algorithm: CONTAINS

Problem

Given a list L and value v, return $v \in L$.

A simple algorithm: CONTAINS

Problem

Given a list L and value v, return $v \in L$.

Algorithm CONTAINS(L, v):

$i, r := 0, \texttt{false}$.
while $i \neq |L|$ **do**
    if $L[i] = v$ **then**
        $r := \texttt{true}$.
        $i := i + 1$.
    else
        $i := i + 1$.
return r.

A simple algorithm: Contains

Problem

Given a list L and value v, return $v \in L$.

Algorithm Contains(L, v):

$i, r := 0,$ `false`.
while $i \neq |L|$ **do**
    if $L[i] = v$ **then**
        $r :=$ `true`.
        $i := i + 1$.
    else
        $i := i + 1$.
return r.

Result: return `true` if $v \in L$ and `false` otherwise.

A simple algorithm: Contains

Problem

Given a list L and value v, return $v \in L$.

Algorithm Contains(L, v):

Input: L is an *array*, v a value.

```
i, r := 0, false.
while i ≠ |L| do
    if L[i] = v then
        r := true.
        i := i + 1.
    else
        i := i + 1.
return r.
```

Result: return `true` if $v \in L$ and `false` otherwise.

A simple algorithm: CONTAINS

Problem

Given a list L and value v, return $v \in L$.

Algorithm EVILCONTAINS(L, v):

Input: L is an *array*, v a value.

1: $L := []$.

2: **return** `false`.

Result: return `true` if $v \in L$ and `false` otherwise.

A simple algorithm: Contains

Problem

Given a list L and value v, return $v \in L$.

Algorithm Contains(L, v):

$i, r := 0,$ `false`.

while $i \neq |L|$ **do**

    if $L[i] = v$ **then**

        $r :=$ `true`.

        $i := i + 1.$

    else

        $i := i + 1.$

return r.

A simple algorithm: CONTAINS

Problem

Given a list L and value v, return $v \in L$.

Algorithm CONTAINS(L, v):

$i, r := 0, \texttt{false}$.
/* L is an *array*, v a value, $i = 0$, and $r = \texttt{false}$. */

while $i \neq |L|$ **do**
    if $L[i] = v$ **then**
        $r := \texttt{true}$.
        $i := i + 1$.
    else
        $i := i + 1$.
/* r is $\texttt{true}$ if $v \in L$ and $\texttt{false}$ otherwise. */
return r.

A simple algorithm: CONTAINS

Problem

Given a list L and value v, return $v \in L$.

Algorithm CONTAINS(L, v):

```
i, r := 0, false.
/* L is an array, v a value, i = 0, and r = false. */
/* inv: 0 ≤ i ≤ |L|, v ∈ L[0, i) implies r = true, v ∉ L[0, i) implies r = false. */
while i ≠ |L| do
    if L[i] = v then
        r := true.
        i := i + 1.
    else
        i := i + 1.
/* r is true if v ∈ L and false otherwise. */
return r.
```

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Proof by induction

Intermezzo: The invariant of Contains holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Proof by induction

Base case Prove invariant holds before the loop.

Hypothesis The invariant holds after the j-th, $j < m$, repetition of the loop.

Step Assume invariant holds when we start the m-th repetition of the loop.
Prove invariant holds again when we reach the end of the m-th repitition.

Intermezzo: The invariant of Contains holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Base case: Prove invariant holds before the loop

Input: L is an *array*, v a value.

$i, r := 0, \texttt{false}$.

/* L is an *array*, v a value, $i = 0$, and $r = \texttt{false}$. */

while ....

Argument

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Base case: Prove invariant holds before the loop

Input: L is an *array*, v a value.

1: $i, r := 0, \texttt{false}$.
/* L is an *array*, v a value, $i = 0$, and $r = \texttt{false}$. */
2: **while** ....

Argument

1. $L[0, i)$ with $i = 0$ is $L[0, 0)$.

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Base case: Prove invariant holds before the loop

Input: L is an *array*, v a value.

1: $i, r := 0, \texttt{false}$.
   /* L is an *array*, v a value, $i = 0$, and $r = \texttt{false}$. */
2: **while** ....

Argument

1. $L[0, i)$ with $i = 0$ is $L[0, 0)$.
2. $L[0, 0)$ is empty, hence $v \notin L[0, 0)$.

Intermezzo: The invariant of Contains holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Base case: Prove invariant holds before the loop

Input: L is an *array*, v a value.

$i, r := 0, \texttt{false}$.

/* L is an *array*, v a value, $i = 0$, and $r = \texttt{false}$. */

while ....

Argument

1. $L[0, i)$ with $i = 0$ is $L[0, 0)$.
2. $L[0, 0)$ is empty, hence $v \notin L[0, 0)$.
3. Hence, $r = \texttt{false}$ must hold (which is the case).

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Step: Prove invariant holds again when we reach the end of the *m*-th repitition.

```
while i ≠ |L| do
    /* Invariant and i ≠ |L|. */
    if L[i] = v then
        r := true.
        i := i + 1.
    else
        i := i + 1.
    /* Invariant. */
```

Argument

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Step: Prove invariant holds again when we reach the end of the *m*-th repitition.

```
while i ≠ |L| do
    /* Invariant and i ≠ |L|. */
    if L[i] = v then
        r := true.
        i := i + 1.
    else
        i := i + 1.
    /* Invariant. */
```

Argument

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Step: Prove invariant holds again when we reach the end of the *m*-th repitition.

while $i \neq |L|$ **do**
    /* Invariant and $i \neq |L|$. */
    if $L[i] = v$ **then**
        $r := \texttt{true}$.
        $i := i + 1$.
    else
        $i := i + 1$.
    /* Invariant. */

Argument

If-statement: Case distinction.

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: If-case ($L[i] = v$ holds).

3: **if** $L[i] = v$ **then**
   /* Invariant, $i \neq |L|$, and $L[i] = v$ */
4:     $r := \texttt{true}$.
5:     $i := i + 1$.
   /* Invariant. */

Argument

After Line 5: prove that Invariant holds for the *updated* values r_{new}, i_{new} of r and i.

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: If-case ($L[i] = v$ holds).

```
3: if L[i] = v then
       /* Invariant, i ≠ |L|, and L[i] = v */
4:     r := true.
5:     i := i + 1.
       /* Invariant. */
```

Argument

After Line 5: prove that Invariant holds for the *updated* values r_{new}, i_{new} of r and i.

1. $L[i] = v$, hence, $v \in L[0, i]$.

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \le i \le |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: If-case ($L[i] = v$ holds).

```
3: if L[i] = v then
      /* Invariant, i ≠ |L|, and L[i] = v */
4:    r := true.
5:    i := i + 1.
      /* Invariant. */
```

Argument

After Line 5: prove that Invariant holds for the *updated* values r_{new}, i_{new} of r and i.

1. $L[i] = v$, hence, $v \in L[0, i]$.
2. $i_{\text{new}} = i + 1$, hence, $v \in L[0, i_{\text{new}})$.

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: If-case ($L[i] = v$ holds).

3: **if** $L[i] = v$ **then**
    /* Invariant, $i \neq |L|$, and $L[i] = v$ */
4:     $r := \texttt{true}$.
5:     $i := i + 1$.
    /* Invariant. */

Argument

After Line 5: prove that Invariant holds for the *updated* values r_{new}, i_{new} of r and i.

1. $L[i] = v$, hence, $v \in L[0, i]$.
2. $i_{\text{new}} = i + 1$, hence, $v \in L[0, i_{\text{new}})$.
3. Hence, $r_{\text{new}} = \texttt{true}$ must hold (which is the case).

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: Else-case ($L[i] \neq v$ holds).

6: **if** $L[i] = v$ **then** … **else**
    /* Invariant, $i \neq |L|$, and $L[i] \neq v$ */
7: $i := i + 1$.
    /* Invariant. */

Argument

After Line 7: prove that Invariant holds for the *updated* value i_{new} of i.

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: Else-case ($L[i] \neq v$ holds).

6: **if** $L[i] = v$ **then** … **else**
   /* Invariant, $i \neq |L|$, and $L[i] \neq v$ */
7:   $i := i + 1$.
   /* Invariant. */

Argument

After Line 7: prove that Invariant holds for the *updated* value i_{new} of i.

1. Assume $r = \texttt{true}$. Hence, $v \in L[0, i)$ by the invariant.

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: Else-case ($L[i] \neq v$ holds).

6: **if** $L[i] = v$ **then** …**else**
    /* Invariant, $i \neq |L|$, and $L[i] \neq v$ */
7:     $i := i + 1$.
    /* Invariant. */

Argument

After Line 7: prove that Invariant holds for the *updated* value i_{new} of i.

1. Assume $r = \texttt{true}$. Hence, $v \in L[0, i)$ by the invariant.
2. $i_{\text{new}} = i + 1$, hence, $v \in L[0, i_{\text{new}})$.

Intermezzo: The invariant of CONTAINS holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: Else-case ($L[i] \neq v$ holds).

6: **if** $L[i] = v$ **then** …**else**
    /* Invariant, $i \neq |L|$, and $L[i] \neq v$ */
7:     $i := i + 1$.
    /* Invariant. */

Argument

After Line 7: prove that Invariant holds for the *updated* value i_{new} of i.

1. Assume $r = \texttt{true}$. Hence, $v \in L[0, i)$ by the invariant.
2. $i_{\text{new}} = i + 1$, hence, $v \in L[0, i_{\text{new}})$.
3. Hence, $r = \texttt{true}$ must hold (which is the case).

Intermezzo: The invariant of Contains holds

Prove the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

Case distinction: Else-case ($L[i] \neq v$ holds).

6: **if** $L[i] = v$ **then** … **else**
    /* Invariant, $i \neq |L|$, and $L[i] \neq v$ */
7: $i := i + 1$.
    /* Invariant. */

Argument

After Line 7: prove that Invariant holds for the *updated* value i_{new} of i.

1. Assume $r = \texttt{false}$. Hence, $v \notin L[0, i)$ by the invariant.
2. $i_{\text{new}} = i + 1$ and $L[i] \neq v$, hence, $v \notin L[0, i_{\text{new}})$.
3. Hence, $r = \texttt{false}$ must hold (which is the case).

Intermezzo: The correctness of CONTAINS

We have proven the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */
while $i \neq |L|$ **do** … **end while**
/* Invariant and $\neg(i \neq |L|)$. */
/* r is `true` if $v \in L$ and `false` otherwise. */
return r.

Questions

Intermezzo: The correctness of Contains

We have proven the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */
while $i \neq |L|$ **do** … **end while**
/* Invariant and $\neg(i \neq |L|)$. */
/* r is `true` if $v \in L$ and `false` otherwise. */
return r.

Questions

1. Do we reach the end of the loop?

Intermezzo: The correctness of Contains

We have proven the invariant holds

/* inv: $0 \leq i \leq |L|$, $v \in L[0, i)$ implies $r = \texttt{true}$, $v \notin L[0, i)$ implies $r = \texttt{false}$. */

while $i \neq |L|$ **do** … **end while**

/* Invariant and $\neg(i \neq |L|)$. */

/* r is `true` if $v \in L$ and `false` otherwise. */

return r.

Questions

1. Do we reach the end of the loop?
2. Assuming /* Invariant and $\neg(i \neq |L|)$ */,
   Do we have /* r is `true` if $v \in L$ and `false` otherwise */?