VANDERBILT UNIVERSITY  School of Engineering

Discrete Structures

CS 2212

(Fall 2020)

17 – Induction and Recurrence

Reminders and Recap

Last time:

Chapter 7

- Weak Induction
- Strong Induction

Today:

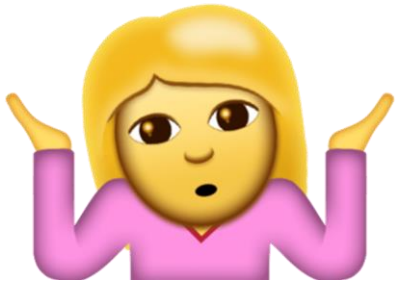
Chapter 7

- Well Ordering Principle (WOP)
- Loop Invariants
- Recursive Definitions

Well Ordering Principle (WOP)

Every nonempty set of nonnegative integers has a smallest element.

- Obvious and straight-forward ….
- How can such an "innocent" fact be of any use???



Well, if you believe induction is a powerful technique, then you also believe that WOP is quite powerful. Why?

The principle of mathematical induction is logically equivalent to the well-ordering principle.

Proof by Well Ordering Principle (WOP)

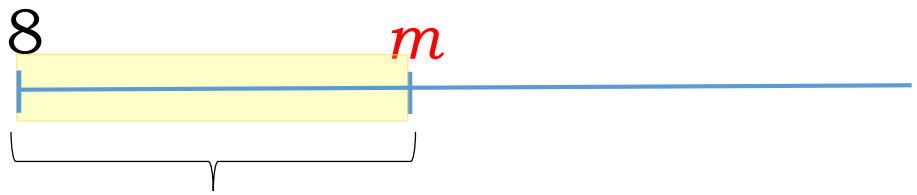
First, lets see an example of how **well ordering principle can be used for proving theorems**.

Example: Prove that any amount of postage worth 8 cents or more can be made from 3-cent or 5-cent stamps.

(Basically a proof by contradiction that uses WOP)

- Let **S** be the set of all integers for which the statement is not true.
- **S** has a smallest element m. Why? Because of the **WOP**.
- It means that it is not possible to make m cents worth of stamps using only 3-cent and 5-cent stamps.

Proof by Well Ordering Principle (WOP)



For any j in the range, it is possible to make j cents worth of stamps using 3- and 5-cents stamps.

We will show that for any m, it is possible to write it as a sum of 3's and 5's, which will be a contradiction.

- $m = 8$: then $m = 3 + 5$.
- $m = 9$: then $m = 3 + 3 + 3$.
- $m = 10$: then $m = 5 + 5$.
- $m \geq 11$: It means $(m\text{-}3) \geq 8$. Thus, it is possible to write $(m\text{-}3)$ as a sum of 3's and 5's. Now, add one more 3 to $(m\text{-}3)$ and we get m, which means m can be written as a sum of 3's and 5's.

Template of Proofs by WOP

Prove that P(n) is true for all $n \in Z^+$ using WOP.

Step 1: Define the set S of counterexamples, that is

$$S = \{ i \in Z^+ \text{ such that } P(i) \text{ is false} \}$$

Step 2: Assume for contradiction that S is nonempty.

Step 3: By the **WOP**, S will have a smallest element m.

Step 4: Reach a contradiction somehow – often by showing that $P(m)$ is actually true. (This step requires work)

Step 5: Conclude that S must be empty, that is, no counterexample exists.

Proof by WOP – Argument Flow…

Prove that P(n) is true for all $n \in Z^+$ using WOP.

By contradiction

If P(n) is not true for all n, then there are some n values for which P(n) is false.

↓

Let S be the set of all such n's.

↓

If we can show S is empty, we will have shown P(n) is true for all n.

↓

So, now the goal is to show S is empty.

→

Assume S is non-empty.

↑

Then it has the smallest element. Why? By **WOP**

↑

Let m be that smallest element. Note $m \in S$, so the assumption is that P(m) is false.

↑

Thus, if we can show P(m) is actually true, then we will have shown $m \notin S$.

→

Consequently, we will have shown our assumption about S being non-empty is incorrect.

↓

Thus, S is empty.

↓

Hence, P(n) is true for all n.

Well Ordering Principle (WOP)

Every nonempty set of nonnegative integers has a smallest element.

The principle of mathematical induction is logically equivalent to the well-ordering principle.

WOP implies the Principle of Mathematical Induction.

WOP: Every nonempty set of nonnegative integers has a smallest element.

Induction: P(n) be a predicate that is parameterized by non-negative integers n. If the following two conditions hold:

- Base case: P(1) is true
- Inductive step: For all $k \geq 1$, P(k) implies P(k+1)

then for all $n \geq 1$, P(n) is true.

WOP implies the Principle of Mathematical Induction.

Proof by Contrapositive: If WOP is true, and P(n) is false for some $n \geq 1$, then at least one of the two conditions for induction must fail.

Suppose that P(n) is false for some $n \geq 1$.

- Let S = Set of integers ≥ 1 for which P(n) is false.
- By WOP, S has smallest element, say $m \neq 0$.
- If $m = 1$: P(1) is false, which means base case fails.
- If $m \geq 2$: P(m-1) is true but P(m) is false.

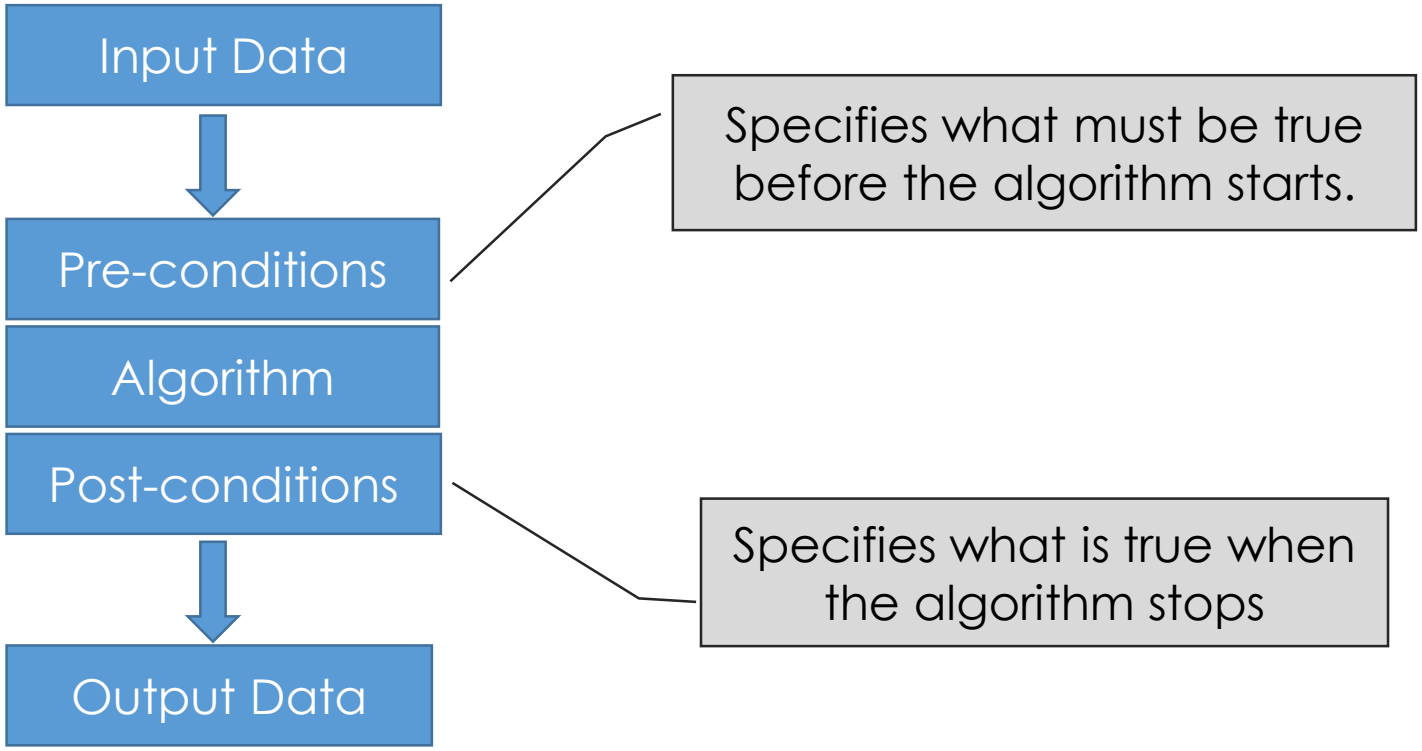
  Let $k = m-1$. P(k) is true, but P(k+1) is false. Thus, the inductive step fails for some $k \geq 1$.

Thus, we showed that P(n) is false for some $n \geq 1$, then one of the two conditions for the principle of mathematical induction must be false.

Loop Invariants and Program Verification

Program Verification

The field of program verification is concerned with formally proving that programs perform correctly.



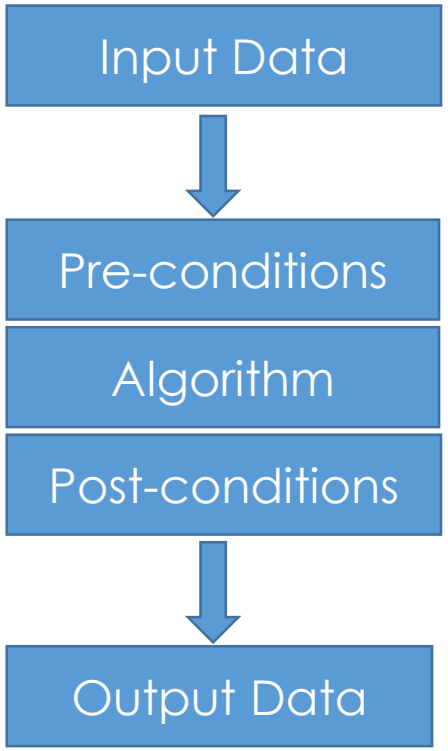
Loop Invariants and Program Verification

Correct Program

A program's correct behavior is defined by stating that,

if a pre-condition is true before the program starts, then

- the program will end after a finite number of steps, and
- a post-condition is true after the program ends.



Loop Invariants and Program Verification

Example:

Program description: Compute the square root of a non-negative real number

Pre-condition: A real number x, such that $x \geq 0$

… Algorithm steps …

Post-condition: A real number y, such that $y \geq 0$ and $y^2 = x$.

Loop Invariants and Program Verification

Example (Program involving loops):

Program description: Compute the n^{th} power of real number x

Pre-condition: n is a non-negative integer, x is a real number
$j = 0$, power = 1

```
While (j < n)
      power := power * x
      j := j + 1
End-while
```

Post-condition: power = x^n

Loop Invariants and Program Verification

Question: How can we guarantee (prove) that the program is correct, that is, for any input the code will provide the right output?

(For our example, how can we ensure that the code it indeed computes the n^{th} power of x.)

Program Verification – Big Picture

Pre-conditions: Inputs and initial values of variables

Program:

Loops

Variables' values are changing in each iteration

Post-conditions: Final values of variables, and the outputs.

How the input is related to the final output is a (global behavior).

Algorithm (loops) just describes how the values are changing in each iteration (local behavior).

Program verification means guarantee that local steps are indeed leading us to the desired global objective. (Fill the gap between above two). How ?

Idea: By showing that a "desired property" is maintained in each iteration of the loop, which terminates eventually

~ **Loop Invariants**

Loop Invariants and Program Verification

Loop Invariant: It is an assertion that is true before each iteration of a loop.

Loop Invariant:

It is a proposition **P** which

- is true for the pre-conditions
- Remains true within the loop
- Is such that **P** $\wedge$ (loop exit) $\rightarrow$ post-condition

Loop Invariants and Program Verification

Example:

Program description: Compute the n^{th} power of real number x.

Pre-condition: n is a non-negative integer,
x is a real number, $j = 0$, power = 1

```
While (j < n)
      power := power * x
      j := j + 1
End-while
```

Loop Invariant:
j is an integer such that $j \leq n$, and power = x^j.

Post-condition: power = x^n

Proof Using Loop Invariants

Proof of Correctness Using Loop Invariant:

Given a loop condition C and a loop invariant I, the following steps are sufficient to establish that

If the pre-condition is true before the loop, then the post-condition is true after the loop:

1. **Show that:** if the pre-condition is true before the loop begins, then I is also true.
2. **Show that:** if C and I are both true before an iteration of the loop, then I is true after the iteration.
3. **Show that:** the condition C will eventually be false.
4. **Show that:** if ¬C and I are both true, then the post-condition is true.

Proof Using Loop Invariants

Proof of Correctness Using Loop Invariant:

Given a loop condition C and a loop invariant I, the following steps are sufficient to establish that

If the pre-condition is true before the loop, then the post-condition is true after the loop:

1. **Show that:** if the pre-condition is true before the loop begins, then I is also true.

Initialization

2. **Show that:** if C and I are both true before an iteration of the loop, then I is true after the iteration.
3. **Show that:** the condition C will eventually be false.
4. **Show that:** if ¬C and I are both true, then the post-condition is true.

Proof Using Loop Invariants

Proof of Correctness Using Loop Invariant:

Given a loop condition C and a loop invariant I, the following steps are sufficient to establish that

If the pre-condition is true before the loop, then the post-condition is true after the loop:

1. **Show that:** if the pre-condition is true before the loop begins, then I is also true.

Initialization

2. **Show that:** if C and I are both true before an iteration of the loop, then I is true after the iteration.

Maintenance

3. **Show that:** the condition C will eventually be false.
4. **Show that:** if ¬C and I are both true, then the post-condition is true.

Proof Using Loop Invariants

Proof of Correctness Using Loop Invariant:

Given a loop condition C and a loop invariant I, the following steps are sufficient to establish that

If the pre-condition is true before the loop, then the post-condition is true after the loop:

1. **Show that:** if the pre-condition is true before the loop begins, then I is also true.

Initialization

2. **Show that:** if C and I are both true before an iteration of the loop, then I is true after the iteration.

Maintenance

3. **Show that:** the condition C will eventually be false.
4. **Show that:** if ¬C and I are both true, then the post-condition is true.

Termination

Loop Invariants and Program Verification

```
While (j < n)
      power := power * x
      j := j + 1
End-while
```

Pre-condition: $n \geq 0$, $x \in \mathrm{R}$, $j = 0$, **power = 1**

Post-condition: **power** $= x^n$

Loop Invariant: $j \leq n$, and **power** $= x^j$.

Step 1 (Initialization):

Assume: Pre-condition true

Prove: Loop invariant true

Assume: n is a non-negative integer, x is a real number, $j = 0$, **power = 1**

Prove: $j = 0$, (so j is an integer).

Also, $j = 0 \leq n$.

power $= x^j = x^0 = 1 =$ power

Loop Invariants and Program Verification

```
While (j < n)
      power := power * x
      j := j + 1
End-while
```

Pre-condition: $n \geq 0$, $x \in R$, $j = 0$, **power = 1**

Post-condition: **power** $= x^n$

Loop Invariant: $j \leq n$, and **power** $= x^j$.

Step 2 (Maintenance):

Assume: Loop condition and loop invariant true before the iteration.

Prove: Loop invariant true after the iteration.

j_1 and $power_1$ denote values **before** the iteration.
j_2 and $power_2$ denote values **after** the iteration.

Assume: $j_1 < n$, and j_1 is an integer such that $j_1 \leq n$, and $power_1 = x^{j_1}$.

Prove: j_2 is an integer such that $j_2 \leq n$, and $power_2 = x^{j_2}$.

- $j_1 < n$, which means $j_1 \leq n - 1$. Thus, $j_1 + 1 \leq n$, which implies $j_2 \leq n$.
- $power_2 = power_1 * x = x^{j_1} * x = x^{1+j_1} = x^{j_2}$

Loop Invariants and Program Verification

```
While (j < n)
      power := power * x
      j := j + 1
End-while
```

Pre-condition: $n \geq 0$, $x \in R$, $j = 0$, **power** = 1

Post-condition: **power** = x^n

Loop Invariant: $j \leq n$, and **power** = x^j.

Step 3 (Termination):

Prove: Loop will terminate.

Prove: $j < n$ becomes false.

Since the value of j is n after n iterations of the loop, the condition $j < n$ will be false after n iterations.

Loop Invariants and Program Verification

```
While (j < n)
      power := power * x
      j := j + 1
End-while
```

Pre-condition: $n \geq 0$, $x \in R$, $j = 0$, power = 1

Post-condition: power = x^n

Loop Invariant: $j \leq n$, and power = x^j.

Step 4 (Termination):

Assume: Loop condition is false and the loop invariant is true.

Prove: Post-condition is true.

Assume: $\neg (j < n)$, and j is an integer such that $j \leq n$, and power = x^j.

Prove: power = x^n

$j \leq n$, and $(j < n)$ is false. It means $j = n$.

Thus, power = $x^j = x^n$

Loop Invariants and Program Verification

Lets look at another example.

The division algorithm is supposed to take a nonnegative integer **a** and a positive integer **d** and compute nonnegative integers **q** and **r** such that $a = dq + r$ and $0 \leq r < d$.

```
r = a;  q = 0
While (r ≥ d)
      r = r - d
      q = q + 1
End-while
```

Loop Invariants and Program Verification

```
r = a;   q = 0
While (r ≥ d)
        r = r - d
        q = q + 1
End-while
```

Pre-condition: **d > 0**, **a ≥ 0**, **r = a**, **q = 0**

Post-condition: **a = qd + r**,
s.t. **q ≥ 0**, and **0 ≤ r < d**.

Loop Invariant: ???

Loop Invariants and Program Verification

```
r = a;   q = 0
While (r ≥ d)
        r = r - d
        q = q + 1
End-while
```

Pre-condition: **d > 0**, **a ≥ 0**, **r = a**, **q = 0**

Post-condition: **a = qd + r**,
s.t. **q ≥ 0**, and **0 ≤ r < d**.

Loop Invariant: **r = (a – qd) ≥ 0**

Loop Invariants and Program Verification

```
r = a;   q = 0
While (r ≥ d)
        r = r - d
        q = q + 1
End-while
```

Pre-condition: **d > 0**, **a ≥ 0**, **r = a**, **q = 0**

Post-condition: **a = qd + r**,
s.t. **q ≥ 0**, and **0 ≤ r < d**.

Loop Invariant: **r = (a – qd) ≥ 0**

Step 1 (Initialization):

Assume: Pre-condition true

Prove: Loop invariant true

Assume: d > 0, a ≥ 0, r = a, q = 0

Prove: r = a – (0 × d) = a (true from pre-condition)

Loop Invariants and Program Verification

```
r = a;   q = 0
While (r ≥ d)
        r = r - d
        q = q + 1
End-while
```

Pre-condition: **d > 0**, **a ≥ 0**, **r = a**, **q = 0**

Post-condition: **a = qd + r**,
s.t. **q ≥ 0**, and **0 ≤ r < d**.

Loop Invariant: **r = (a – qd) ≥ 0**

Step 2 (Maintenance):

Assume: Loop condition and loop invariant true before the iteration.

Prove: Loop invariant true after the iteration.

r_1 and q_1 denote values before the iteration.
r_2 and q_2 denote values after the iteration.

Assume: $r_1 \geq d$, and $r_1 = (a - q_1 d) \geq 0$

Prove: $r_2 = (a - q_2 d) \geq 0$??

For this, $r_2 = r_1 - d = a - q_1 d - d$

$= a - d(q_1 + 1) \geq 0$

Thus, $r_2 = (a - dq_2) \geq 0$

Loop Invariants and Program Verification

```
r = a;   q = 0
While (r ≥ d)
        r = r - d
        q = q + 1
End-while
```

Pre-condition: d > 0, a ≥ 0, r = a, q = 0

Post-condition: a = qd + r, such that q ≥ 0, and 0 ≤ r < d.

Loop Invariant: r = (a – qd) ≥ 0

Step 3 (Termination):

Prove: Loop will terminate after a finite number of steps.

Loop condition: r ≥ d

Since d > 0, r is essentially decreasing in each iteration. Also d is fixed. Thus, there will be a point eventually when r < d. At this point, the while loop condition becomes false.

Loop Invariants and Program Verification

```
r = a;   q = 0
While (r ≥ d)
        r = r - d
        q = q + 1
End-while
```

Pre-condition: d > 0, a ≥ 0, r = a, q = 0

Post-condition: a = qd + r, such that q ≥ 0, and 0 ≤ r < d.

Loop Invariant: r = (a – qd) ≥ 0

Step 4 (Termination):

Assume: Loop condition is false and the loop invariant is true.

Prove: Post-condition is true.

Assume: r < d, r = (a – qd) ≥ 0

Prove: a = qd + r, such that q ≥ 0, and 0 ≤ r < d.

Loop Invariants and Program Verification

- Did you notice the similarities in the plan of action of proof using loop invariant and mathematical induction.
- They are very similar.
- Think of loop invariant as an inductive hypothesis

Recursive (Inductive) Definition and Structural Induction

Recursive Definition

Recall the set of **even** integers > 0

- One way to define the set is: $A = \{2, 4, 6, \ldots\}$
- Another way to define is: $A = \{ 2k + 2 \mid k \in N \}$
- And yes, there is another way known as **recursive definition.**

Basis: $2 \in A$

Recursive rule: If $x \in A$, then $x + 2 \in A$. Nothing else is in A unless it is obtained from the basis and recursion.

Recursive Definition

In a recursive definition of a function, the value of the function is defined in terms of the output value of the function on smaller input values.

Factorial of *n* = *n*!

Basis: $0! = 1$

Recursive rule: $n! = n \times (n\text{-}1)!$

Components of Recursive Definition

Defining a set S recursively includes three parts:

1. **Basis:** Specify one or more elements of S.
2. **Recursive (or Inductive) rule:** Specify one or more rules to construct elements of S from existing elements of S.
3. **Exclusion statement:** Specify that no other elements are in S.
    - This last step is often assumed.
    - We will make it optional once you've learned the basics.

Recursive Definition

Example: Write an inductive definition for

$$S = \{ 3, 16, 29, 42, \ldots \}.$$

Basis: $3 \in S$.

Recursive (inductive) rule: If $x \in S$ then $(x + 13) \in S$.

Closure: Nothing else is in S unless it is obtained from the basis and recursive rule.

Recursive Definition

Example: Write a recursive definition for

$S = \{ 3, 4, 5, 8, 9, 12, 16, 17, 20, 24, 28, 32, 33, \ldots \}$.

Hint: To simplify things, we write S as the union of familiar sets:

$S = \{3, 5, 9, 17, 33, \ldots\} \cup \{4, 8, 12, 16, 20, 24, \ldots\}$.

Basis: $3, 4 \in S$.

Recursive rule: If $x \in S$ then

$2x - 1 \in S$ (when x is odd)

$x + 4 \in S$ (when x is even)

Closure: Nothing else is in S unless it is obtained from the basis and recursion.

Recursive Definition

Example: Describe the elements in the set S which is defined as follows:

Basis: $2 \in S$.

Recursive rule: $x \in S$ implies $x \pm 3 \in S$.

Closure: Nothing else is $\in S$ unless it is obtained from the basis and induction.

Answer:

$S = \{2, 5, 8, 11, \ldots\} \cup \{-1, -4, -7, -10, \ldots\}$

Recursive Definition

Example: Write a recursive definition for

$$S = \{ ac, \quad aacc, \quad aaaccc, \quad \dots \}$$

which can be define recursively as the set

$$S = \{ a^n c^n \mid n \in \mathbf{N}^+ \}$$

Answer:

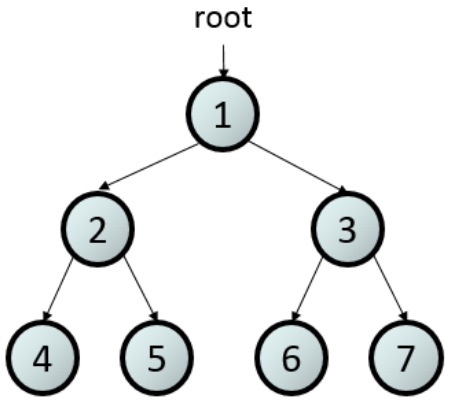
Basis: $ac \in S$

Recursion: If $\boldsymbol{y} \in S$ then $a\boldsymbol{y}c \in S$

Closure: Nothing else is in S unless it is obtained from the basis and recursion.

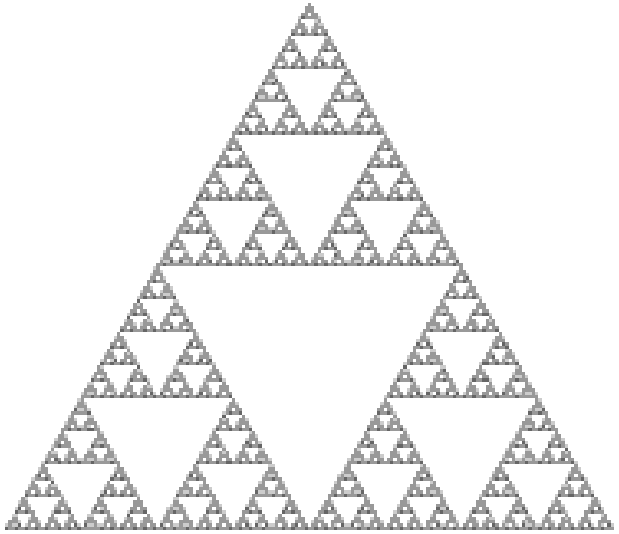
Recursively Defined Structures

Apart from sets, there are other objects in Computer Science that can be defined using this **recursive** approach, e.g., binary trees.



These recursively defined objects and structures have the property that

parts of them exhibit the same characteristics and have the same properties as the whole.



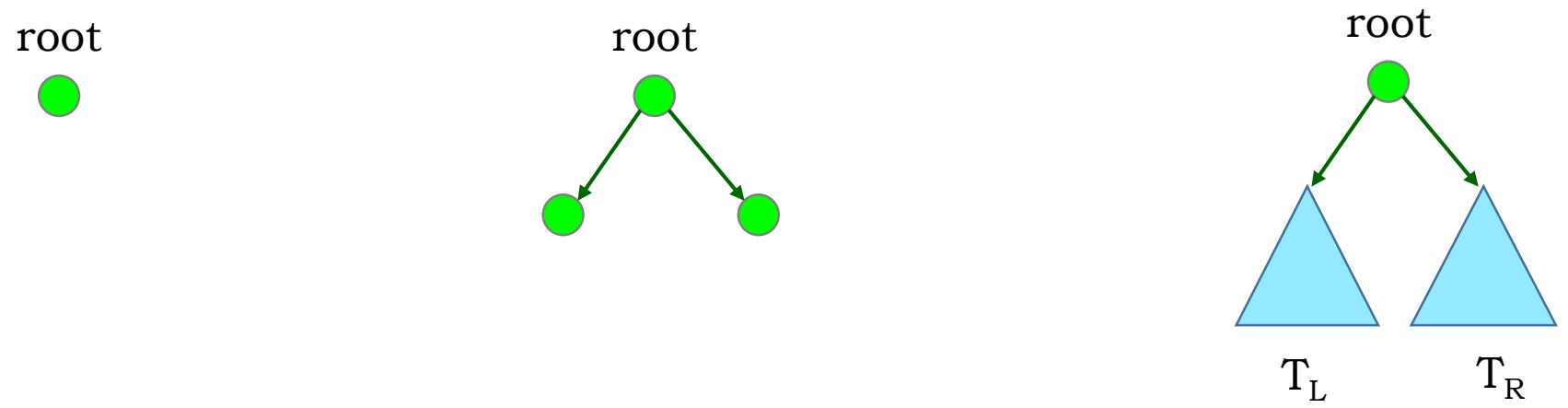
Recursive Definition

Example: Binary Tree

A non-empty binary tree is either:

Basis: A root node r with no pointers, or

Recursive (or Inductive) rule: A root node r pointing to 2 non-empty binary trees T_L and T_R.



Recursive Definition – Binary Trees

Example: Height of a Non-empty Binary Tree

$$h: \quad T \quad \rightarrow \quad N$$

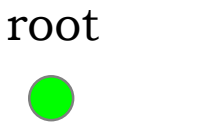
The height h(T) of a non-empty binary tree T is a function defined recursively as follows:

Basis: If T is a single root node, then $h(T) = 0$

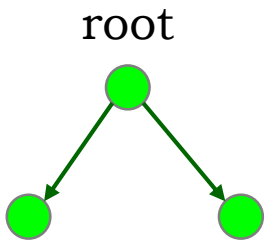
Recursive (or Inductive) rule: If T is a root node connected to two "sub-trees" T_L and T_R, then

$$h(T) = \max \{ h(T_L), h(T_R) \} + 1.$$

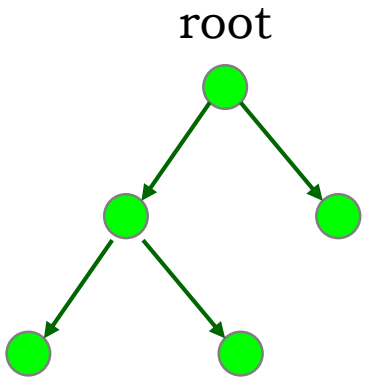
Recursive Definition – Binary Trees



Height = 0



Height = 1



Height = 2

Structural Induction

- Next, we will see how to prove theorems for **recursively defined structures** using structural induction.
- **Structural induction** is a proof methodology similar to standard induction except that it works in the domain of recursively defined structures, like trees, graphs, sets, etc.
- It's just another tool in the toolbox

Structural Induction

How it usually goes: The general steps in a structural induction proof go something like this:

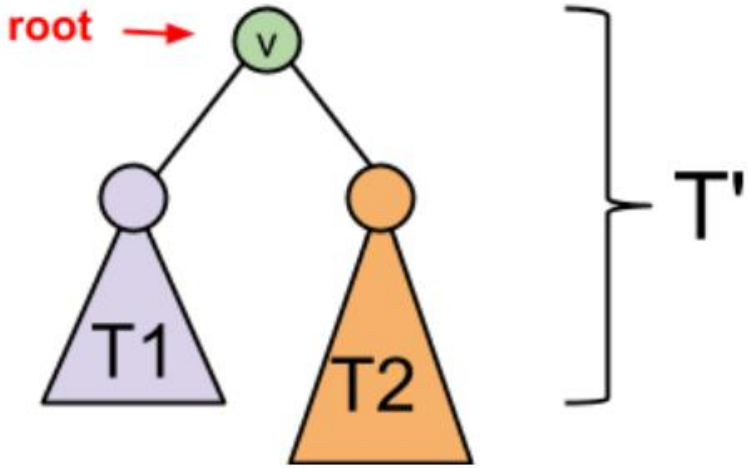
- **Step 1: Prove P(r) for the base case**. The base case might be an empty structure or a trivial structure with a single node or vertex. For example, an empty tree, or tree with a single node r.
- **Step 2: Assume the inductive hypothesis** for an arbitrary structure. For example, assume that the statement holds for some tree T (i.e., assume P(T)).
- **Step 3: Inductive step (proof).** Use the recursive part of the structure's definition to show that a new structure, say T*, that contains the existing structure T, satisfies P(T*).

Structural Induction – Binary Tree

Recursive definition of a Full Binary

Basis: A single vertex with no edges is a full binary tree.

Recursive rule: T' can be constructed from binary trees T1 and T2 by adding a new vertex v and then adding an edge between v and the root of T1 and an edge between v and the root of T2.



Structural Induction

Theorem: If T is a full binary tree, then

$$c(T) \leq 2^{h(T)+1} - 1.$$

Proof: (structural induction in action)

Basis: For a full binary tree consisting only of a root, c(T) = 1 and h(T) = 0. Now,

$$c(T) = 1 \leq 2^{0+1} - 1 = 1.$$

Hence, base case is satisfied.

Inductive hypothesis: Let's assume:

$$c(T_1) \leq 2^{h(T1)+1} - 1 \text{ and } c(T_2) \leq 2^{h(T2)+1} - 1$$

whenever T_1 and T_2 are full binary trees.

Structural Induction

Inductive step:

We need to show that c(T) is true.

$$c(T) = 1 + c(T_1) + c(T_2) \qquad \text{(definition of c(T))}$$

$$\leq 1 + (2^{h(T_1)+1} - 1) + (2^{h(T_2)+1} - 1) \qquad \text{(inductive hypothesis)}$$

$$\leq 2 \cdot \max(2^{h(T_1)+1}, 2^{h(T_2)+1}) - 1$$

$$= 2 \cdot 2^{\max(h(T_1), h(T_2))+1} - 1 \qquad (\max(2^x, 2^y) = 2^{\max(x,y)})$$

$$= 2 \cdot 2^{h(T)} - 1 \qquad \text{(definition of h(T))}$$

$$= 2^{h(T)+1} - 1$$

This is precisely what we wanted to show. So if T is a full binary tree, then it is the case that

$$c(T) \leq 2^{h(T)+1} - 1.$$

General structure of Inductive Proofs on Sets

To prove that all elements of a set S have a property P, that is,

$\forall x \in S$, P(x) is true.

Step 1: Prove for the base case. Prove that the property holds for each base element of the set.

Step 2: Assume the inductive hypothesis. In the inductive hypothesis, assume $\exists S' : |S'| \geq n_0$ and P(x) is true $\forall x \in S'$.

Step 3: Inductive step (proof). From S', obtain a new set by applying the recursive rule. Then show that all elements in this new set also satisfy the stated property.

In other words, if $x \in S'$ and P(x) holds, then by applying recursive rule r, we get a new element r(x). Then, show that P(r(x)) is also true.

Structural Induction – Sets

Example:

Let S be a set defined as follows:

Base case: $4 \in S$

Recursive rule: If $x \in S$, then $x^2 \in S$

Prove that $\forall x \in S$, x is even.

Basis: In the base case of the definition of S, we have that $4 \in S$. Since 4 is an even number, so the base case holds.

Inductive hypothesis: Assume that $\exists S'$: $|S'| \geq 1$ and all elements in S' are even.

Structural Induction – Sets

Inductive step:

We need to show that the desired property is true for the element obtained by applying a recursive rule to an arbitrarily selected element of S'.

Let x be an arbitrary element in S'. By inductive hypothesis, x is even.

We need to show that a new element obtained by applying recursive rule to x, is also even.

By recursive rule, new element obtained from x is x^2. Since square of even number is even and x is even, so x^2 is even. Hence, the desired property also holds for the new element. QED.

Structural Induction – Sets

Example:

Let S be a subset of $\mathbb{N} \times \mathbb{N}$ defined recursively as follows:

Base case: $(0,0) \in S$

Recursive rule: If $(x, y) \in S$, then $(x+5, y+1) \in S$

Prove that $\forall (x, y) \in S$, $x + y$ is a multiple of 3.

Basis: $(0,0) \in S$. Since $0 + 0 = 0$, which is a multiple of 3, base case is satisfied.

Inductive hypothesis: Assume that $\exists S'$: $|S'| \geq 1$ and if $(x, y) \in S'$, then $x + y$ is a multiple of 3.

Structural Induction – Sets

Inductive step:

We need to show that the desired property is true for the element obtained by applying a recursive rule to an arbitrarily selected element of S'.

Let (x, y) be an arbitrary element in S'.

By inductive hypothesis, x + y is a multiple of 3.

By the recursive rule (x+5, y+1) is a new element. We need to show that x + y + 6 is a multiple of 3.

x + y is a multiple of 3, and 6 is also a multiple of 3, therefore,

x + y + 6 is also a multiple of 3, which is the desired result. QED.