LECTURE NOTES
DISCRETE MATHEMATICS

CHAPTER 12

Recursion



Updated: 19 November 2026

12 Recursion

A task either finishes immediately or splits into exactly two smaller tasks, each of which follows the same rule. Suppose every resulting task eventually finishes. How does the number of finished end tasks compare with the number of splits? A tree drawing can record one execution, but we want an answer covering every possible finite pattern of splits.

We will define counting functions on these trees and prove a relationship between their values. First we sharpen the function language from Chapters 1 and 4, including what it means to cover every output and how to find all inputs leading to a chosen output. The tree construction then brings together Chapter 6's rooted trees, Chapter 11's recursive descriptions and Chapter 5's induction.

What you should be able to do

After working through this chapter, you should be able to:

- distinguish injectivity, surjectivity and bijectivity;
- calculate images and inverse images on stated domains;
- distinguish an inverse image from an inverse function;
- describe full binary trees through a base object and a constructor;
- translate a recursive counting procedure into a mathematical function;
- trace recursive evaluation on a finite tree;
- prove a tree property using both subtree hypotheses in structural induction.

12.1 Injectivity and surjectivity

A function $f : X \to Y$ assigns exactly one output in Y to each input in X. The **domain** X lists the permitted inputs; the **codomain** Y lists the permitted outputs. Its **range** $f(X) = \{f(x) : x \in X\}$ is the set of outputs actually obtained, which may be smaller than Y.

Chapter 4 studied injectivity. We now place that property beside surjectivity, which asks whether any element of the codomain is missed.

❄ Definition 12.1 — Injective, surjective and bijective

Let $f : X \to Y$. The function is **injective** if equal outputs force equal inputs:

$$\text{for all } a, b \in X, \quad f(a) = f(b) \Longrightarrow a = b.$$

It is **surjective**, or **onto**, if

$$\text{for every } y \in Y, \text{ there is an } x \in X \text{ with } f(x) = y.$$

It is **bijective** if it is both injective and surjective.

For a particular output in Y, injectivity permits at most one input leading to it; surjectivity requires at least one. A bijection gives exactly one. Every function already gives one output per input. That requirement alone does not settle either property.

Example 12.1. Use the following new data. Let $U = \{\text{Alice}, \text{Bob}, \text{Charlie}, \text{David}\}$, $J = \{10, 20, 30, 40, 50\}$ and $P = \{\text{Printer 1}, \text{Printer 2}, \text{Printer 3}\}$. The complete table defines $i : U \to J$ and $d, h : U \to P$.

Input	Alice	Bob	Charlie	David
i	10	20	30	40
d	Printer 1	Printer 1	Printer 2	Printer 3
h	Printer 1	Printer 1	Printer 2	Printer 2

The identifier function i is injective: every input has a different output. It is not surjective onto J, because $50 \in J$ is not the output of any user. One missed codomain element is enough to refute surjectivity.

Example 12.2. The function d in Example 12.1 is surjective: Printer 1 is reached by Alice, Printer 2 by Charlie, and Printer 3 by David. These witnesses account for every element of P. But d is not injective, since Alice and Bob are distinct and share Printer 1.

Surjectivity asks whether an output is reached, not whether it is reached only once.

Example 12.3. Keep the identifier assignments, but define $j : U \to J'$ with $J' = \{10, 20, 30, 40\}$ and $j(u) = i(u)$. The outputs remain distinct, so j is injective. They also cover every element of the stated codomain J', so j is surjective and therefore bijective.

The rules for i and j agree at every input. The codomain change alters the surjectivity question. We have specified a function with a new codomain, not proved that the unused value 50 was reached.

Example 12.4. For h in Example 12.1, Alice and Bob share Printer 1, so h is not injective. No user is assigned Printer 3, so h is not surjective.

Together, i, d, j, h show all four possibilities: injective only, surjective only, both, and neither. Injectivity and surjectivity are not opposites.

Example 12.5. Let $g : \mathbb{Z} \to \mathbb{Z}$ be given by $g(k) = k + 3$. Prove that g is bijective.

Injectivity. If $g(a) = g(b)$ for integers a, b, then $a + 3 = b + 3$, hence $a = b$.

Surjectivity. Let y be an arbitrary integer in the codomain. Choose $x = y - 3$, which is an integer and therefore lies in the domain. Then $g(x) = (y - 3) + 3 = y$. Every permitted output has an input witness, so g is onto. Both properties hold. □

For example, the witness for output -2 is input -5. The general choice $x = y - 3$ covers all outputs; the single numerical example would not do so.

12.2 Images and inverses

Recall that for $A \subseteq X$, the **image** under $f : X \to Y$ is

$$f(A) = \{f(a) : a \in A\} \subseteq Y.$$

An image follows inputs forwards. An inverse image instead collects all inputs whose outputs belong to a specified subset of Y.

❄ Definition 12.2 — Inverse image

For $f : X \to Y$ and $B \subseteq Y$, its **inverse image**, or **preimage**, is

$$f^{-1}(B) = \{x \in X : f(x) \in B\} \subseteq X.$$

This notation defines a set of inputs for any function, whether or not it is injective or surjective.

To find the inputs leading to one output y, use the singleton set $\{y\}$ and write $f^{-1}(\{y\})$. We retain the braces so that a set of inputs is visibly distinguished from an inverse-function value. The superscript -1 here does not mean a reciprocal.

Example 12.6. Use $h : U \to P$ from Example 12.1. For $A = \{\text{Alice}, \text{Bob}, \text{Charlie}\}$,

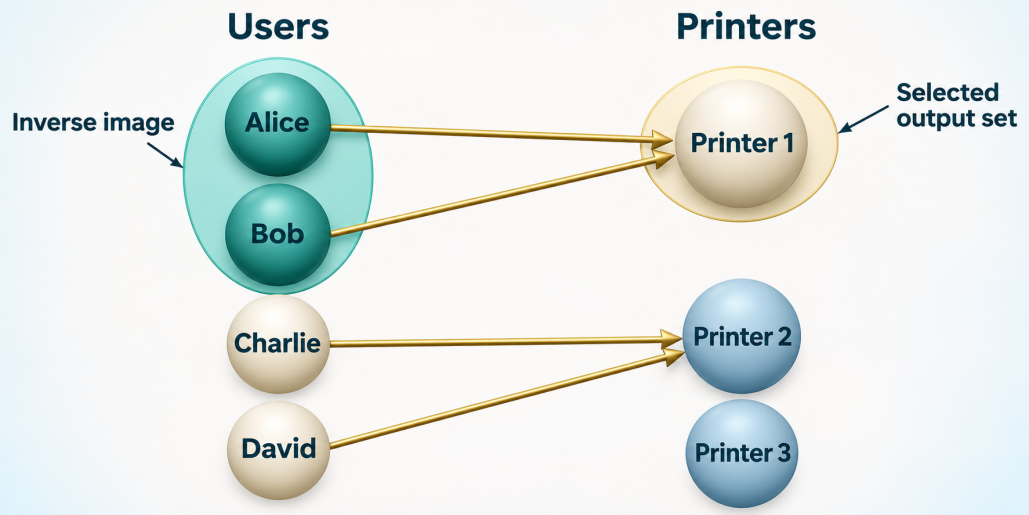
$$h(A) = \{\text{Printer 1}, \text{Printer 2}\}.$$

Repeated outputs are included only once. For an individual input, $h(\text{Alice}) = \text{Printer 1}$, whereas $h(\{\text{Alice}\}) = \{\text{Printer 1}\}$. The first is one printer label; the second is a set.

Example 12.7. For the same function h,

$$h^{-1}(\{\text{Printer 1}\}) = \{\text{Alice}, \text{Bob}\},$$
$$h^{-1}(\{\text{Printer 1}, \text{Printer 2}\}) = U.$$

In the second calculation, a user is included if their output belongs to either of the two listed categories. Every user satisfies that condition. The result is a subset of the domain, not a set of printers.



The inverse image of {Printer 1} under h contains both Alice and Bob.

Example 12.8. No input of h leads to Printer 3, so $h^{-1}(\{\text{Printer 3}\}) = \emptyset$. This is not a failure of the inverse-image definition. The set simply has no matching inputs.

Also, $h^{-1}(\emptyset) = \emptyset$: no function output belongs to the empty set. By contrast, $h^{-1}(P) = U$, since every output of the function lies in its codomain.

Example 12.9. Define $q : \mathbb{Z} \to \mathbb{Z}$ by $q(k) = k^2$. Solving the relevant integer equations gives

$$q^{-1}(\{4\}) = \{-2, 2\}, \qquad q^{-1}(\{0\}) = \{0\}, \qquad q^{-1}(\{-1\}) = \emptyset.$$

Both -2 and 2 square to 4; only 0 squares to 0; no integer square is negative. Even a one-element inverse image is written as a set. The failure of injectivity does not prevent any of these inverse images from being defined.

❄ Definition 12.3 — Inverse function

If $f : X \to Y$ is bijective, its **inverse function** $f^{-1} : Y \to X$ sends each $y \in Y$ to the unique $x \in X$ with $f(x) = y$. It satisfies

$$f^{-1}(f(x)) = x \quad (x \in X), \qquad f(f^{-1}(y)) = y \quad (y \in Y).$$

Surjectivity supplies an input for every y; injectivity makes that input unique.

Example 12.10. For the bijection $g(k) = k + 3$ in Example 12.5, the inverse function is $g^{-1} : \mathbb{Z} \to \mathbb{Z}$, $g^{-1}(y) = y - 3$. In particular,

$$g^{-1}(7) = 4, \qquad g^{-1}(\{7\}) = \{4\}.$$

The first expression evaluates the inverse function at one integer. The second gives the inverse image of a singleton set. Also, $g(g^{-1}(7)) = g(4) = 7$, as required. The inverse is not $1/g(7) = 1/10$.

Example 12.11. For $h : U \to P$, an attempted inverse $P \to U$ would have two possible users for Printer 1 and no possible user for Printer 3. It cannot assign every printer the unique user required to undo h.

The set $h^{-1}(\{\text{Printer 1}\}) = \{\text{Alice}, \text{Bob}\}$ remains meaningful. It reports both inputs rather than pretending there is one inverse-function value.

12.3 Full binary trees

Recall that, in a rooted tree, a leaf has no children and an internal vertex has at least one child. A full binary tree has exactly two children at every internal vertex. Its one-vertex tree has one leaf and no internal vertices.

For the recursive description, distinguish a left child from a right child. This makes our trees **ordered**: swapping different left and right subtrees changes the ordered tree. The counting results below do not depend on that order.

❄ Definition 12.4 — The family $\mathcal{T}$ of finite full binary trees

The family $\mathcal{T}$ is specified by the following rules.

a) **Base object.** The one-vertex rooted tree • belongs to $\mathcal{T}$.

b) **Constructor.** If $S, R \in \mathcal{T}$, then $\mathrm{Node}(S, R) \in \mathcal{T}$. It has a new root whose left subtree is a copy of S and whose right subtree is a separate copy of R.

c) **Exclusion.** Only trees obtained by finitely many applications of these rules belong to $\mathcal{T}$.

Here tree objects record the rooted, left/right shape. Renaming vertices or moving a drawing does not create a different member of $\mathcal{T}$. The copies have disjoint vertex sets, even if S and R have the same shape. The constructor joins their roots to the new root by two new edges. Every non-leaf tree in the family has a unique left subtree and right subtree at its root, so we can read its construction back from the tree.

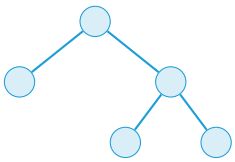
Example 12.12. The base object • has one vertex. Using it twice gives $B = \mathrm{Node}(\bullet, \bullet)$.



$B = \mathrm{Node}(\bullet, \bullet)$

The tree B has two distinct leaves, one in each copy of •, and one internal vertex. Using the same base shape twice does not identify its two vertices with one another.

Example 12.13. Using B from Example 12.12, form $A = \mathrm{Node}(\bullet, B)$. Its left subtree is a single leaf and its right subtree contains a branch.



$A = \mathrm{Node}(\bullet, B)$

There are three leaves and two internal vertices. The leaves have different depths, but every internal vertex has exactly two children, so A is full binary. The ordered tree $\mathrm{Node}(B, \bullet)$ has the opposite left/right arrangement and the same counts.

Example 12.14. A root with exactly one leaf child is not in $\mathcal{T}$. It is not the base object, and the constructor always gives its new root two children. Likewise, a root with three children cannot be obtained.

An infinite branching tree is also excluded. Every construction allowed here uses finitely many steps. This finiteness will let a recursive evaluation reach base objects rather than continue indefinitely.

12.4 Recursive counting functions

A mathematical function may take a tree as its input. The input need not be a number. Our opening question needs a function that returns the number of leaves of a tree in $\mathcal{T}$.

Example 12.15. Counting leaves. Consider this pseudocode, with input a tree $T \in \mathcal{T}$. Here `Node(S,R)` means that S and R are its left and right subtrees.

```
CountLeaves(T):
    if T is a single leaf:
        return 1
    else:
        write T as Node(S,R)
        return CountLeaves(S) + CountLeaves(R)
```

To translate it, name its output $L(T)$ and state the function's type: $L : \mathcal{T} \to \mathbb{N}$, where $\mathbb{N} = \{0, 1, 2, \ldots\}$. The first branch says $L(\bullet) = 1$. The other branch says

$$L(\mathrm{Node}(S, R)) = L(S) + L(R).$$

Every leaf of the combined tree lies in exactly one child subtree; the new root is not a leaf. Thus the two returned counts are added without including an extra leaf for the root. These rules describe the procedure mathematically, with an input domain, an output set and both cases.

Each recursive call uses a proper child subtree, which has fewer vertices than its parent tree. Since the input tree is finite, successive calls eventually reach the one-vertex base case. The base rule and the unique root decomposition therefore determine one value for every allowed input.

❄ Definition 12.5 — Leaf and internal-vertex counting functions

For $T \in \mathcal{T}$, let $L(T)$ count its rooted leaves and $I(T)$ its internal vertices. Their recursive descriptions are

$$L(\bullet) = 1, \quad L(\mathrm{Node}(S, R)) = L(S) + L(R),$$
$$I(\bullet) = 0, \quad I(\mathrm{Node}(S, R)) = 1 + I(S) + I(R),$$

for all $S, R \in \mathcal{T}$. Both functions have domain $\mathcal{T}$ and codomain $\mathbb{N}$.

The new root is an internal vertex, which accounts for the added 1 in the rule for I. All other internal vertices occur in the two disjoint child subtrees.

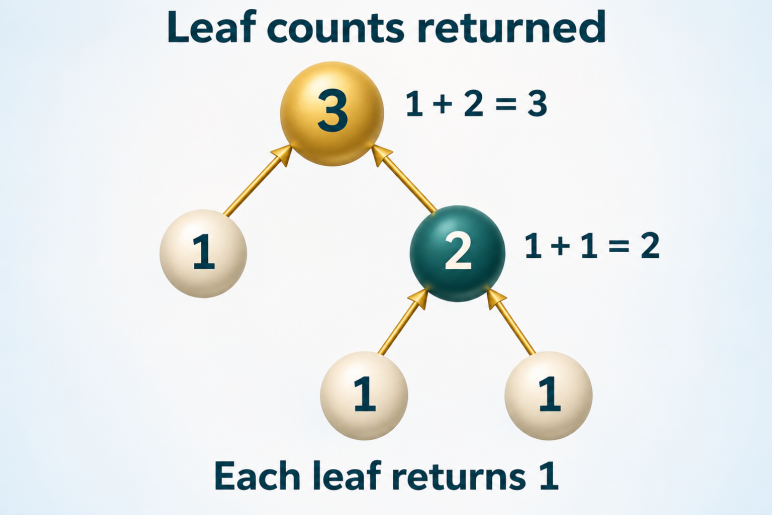
Example 12.16. Recursive evaluation. For $A = \mathrm{Node}(\bullet, B)$ with $B = \mathrm{Node}(\bullet, \bullet)$, the procedure makes one call on each child of the root. The call on B itself makes two leaf calls. Its numerical evaluation is

$$\begin{aligned} L(A) &= L(\bullet) + L(B) \\ &= 1 + \big(L(\bullet) + L(\bullet)\big) \\ &= 1 + (1 + 1) = 3. \end{aligned}$$

The two leaf calls inside B each return 1; the call on B returns 2; the outer call returns $1 + 2 = 3$. The matching internal count is

$$I(A) = 1 + I(\bullet) + I(B) = 1 + 0 + (1 + 0 + 0) = 2.$$

The recursive values agree with the drawing.



Each node displays its subtree's leaf count: the right subtree returns 2 and the root returns $1 + 2 = 3$.

Python (non-examinable)

To run the procedure, represent a single leaf by `None` and $\mathrm{Node}(S, R)$ by the ordered pair `(S, R)`, called a *tuple* in Python. Here `None` represents one vertex, not an empty tree. The line beginning `def` defines the function; `return` gives its output. The condition `tree is None` recognises the base leaf.

```
def count_leaves(tree):
    if tree is None:
        return 1
    left, right = tree
    return count_leaves(left) + count_leaves(right)

B = (None, None)
A = (None, B)
print(count_leaves(B), count_leaves(A))
```

Output.

```
2 3
```

Python (non-examinable) — continued

The assignment `left, right = tree` reads the two child subtrees. Each recursive call applies the same function to a child, and its returned value supplies one of the two counts being added. These are exactly the trees B and A traced above. The code assumes an input built from the stated leaf and two-child rules; it does not check arbitrary Python data. The example computes two values, while the mathematical definitions and proofs describe the whole family.

Example 12.17. Repeated subtrees. For $D = \mathrm{Node}(B, B)$,

$$L(D) = 2 + 2 = 4, \qquad I(D) = 1 + 1 + 1 = 3.$$

Each occurrence of B supplies a separate subtree. Both must contribute to the count. The whole tree has $4 + 3 = 7$ vertices. The formula $L(D) = L(B)$ would omit all leaves in the second copy.

Example 12.18. Suppose someone writes only $F(\mathrm{Node}(S, R)) = F(S) + F(R)$ for $F : \mathcal{T} \to \mathbb{N}$. There is no rule yet for $F(\bullet)$. Setting that base value to 0 gives zero at every tree; setting it to 1 gives the leaf count. Both choices obey the displayed constructor rule.

The function description needs its base case, just as a sequence recurrence in Chapter 11 needs its initial information. The tree domain also excludes the one-child inputs for which the constructor rule supplies no case.

Example 12.19. Define $W : \mathcal{T} \to \mathbb{N}$ by $W(\bullet) = 1$ and $W(\mathrm{Node}(S, R)) = 1 + W(S) + W(R)$. Then $W(B) = 1 + 1 + 1 = 3$, although B has only two leaves. The extra 1 counts the new root. In fact, this rule counts all vertices rather than leaves. The recurrence can be complete and evaluable while describing a quantity different from the intended one.

12.5 Structural induction

Ordinary induction follows the construction of integers from a starting value and a next step. Here an object is built from a leaf or from two already constructed child trees. Our proof can follow those same construction rules.

❄ Definition 12.6 — Structural induction on $\mathcal{T}$

To prove a property $Q(T)$ for every $T \in \mathcal{T}$:

a) **Base case.** Prove $Q(\bullet)$.

b) **Constructor case.** Let $S, R \in \mathcal{T}$ be arbitrary. Assume $Q(S)$ and $Q(R)$, and use them to prove $Q(\mathrm{Node}(S, R))$.

These checks establish Q for every tree generated by the defining rules.

The principle works because every allowed tree has a finite construction. The property starts at each base leaf and is preserved whenever two constructed trees are joined. Following the construction to its root therefore establishes the property of the finished tree. The exclusion clause ensures that no other kinds of object need a separate case.

The two subtree hypotheses may concern trees of different shapes and sizes. We assume the property for the child trees, then prove it for their newly constructed parent. Assuming it for that parent instead would assume the target we are trying to establish.

Example 12.20. Leaves and internal vertices. Prove that for every $T \in \mathcal{T}$,

$$L(T) = I(T) + 1.$$

Proof. We use structural induction.

Base case. For the single-leaf tree, $L(\bullet) = 1 = 0 + 1 = I(\bullet) + 1$.

Constructor case. Let $S, R \in \mathcal{T}$ and assume both

$$L(S) = I(S) + 1, \qquad L(R) = I(R) + 1.$$

The target is $L(\mathrm{Node}(S, R)) = I(\mathrm{Node}(S, R)) + 1$. Using the recursive definitions and the two hypotheses,

$$\begin{aligned} L(\mathrm{Node}(S, R)) &= L(S) + L(R) \\ &= (I(S) + 1) + (I(R) + 1) \\ &= I(S) + I(R) + 2 \\ &= \big(1 + I(S) + I(R)\big) + 1 \\ &= I(\mathrm{Node}(S, R)) + 1. \end{aligned}$$

The base and constructor cases cover the defining rules of $\mathcal{T}$, so the claim holds for every $T \in \mathcal{T}$. □

Each child tree supplies its own extra leaf in the induction hypotheses. The constructor adds one internal root, leaving one extra leaf for the whole tree.

Example 12.21. If a tree in $\mathcal{T}$ has 18 internal vertices, it has 19 leaves and $18 + 19 = 37$ vertices in total. Conversely, 24 leaves require 23 internal vertices. We can make these deductions without knowing the particular shape, because the proof covers every tree in the specified family.

In the opening task model, internal vertices record splits and leaves record finished end tasks. There is therefore one more finished end task than split, provided the execution is finite and each split creates exactly two child tasks.

Example 12.22. A proposed proof handles the base case and then checks only $\mathrm{Node}(S, S)$. That does not establish the claim for $\mathrm{Node}(S, R)$ when S and R differ. For example, $A = \mathrm{Node}(\bullet, B)$ has different child shapes.

The constructor permits any pair of trees from $\mathcal{T}$. Its proof case must therefore allow arbitrary S, R and use the property of each. The two hypotheses need not be numerically identical.

Example 12.23. Consider the rooted tree with a root and exactly one leaf child. It has one leaf and one internal vertex, so $L = I + 1$ would read $1 = 2$. This does not refute Example 12.20: the tree is not full binary and is outside $\mathcal{T}$.

Its leaf count is still an ordinary count, but our two-child constructor and recursive specification do not describe this input. Changing the family requires changing the rules and checking which conclusions remain valid.

12.6 Exercises

12.1. Let $X = \{0, 1, 2\}$ and $Y = \{a, b, c\}$. Three functions $X \to Y$ are defined by the complete table.

Input	0	1	2
f	a	b	c
g	a	a	b
h	c	b	a

For each function, give the range and decide whether it is injective, surjective and bijective. Give witnesses for any failed property.

12.2. Let $i : \{1, 2, 3\} \to \{10, 20, 30, 40\}$ be $i(k) = 10k$. Let $j : \{1, 2, 3\} \to \{10, 20, 30\}$ use the same rule. Classify each function as injective, surjective, both or neither. Explain exactly what changed.

12.3. Use complete tables to give:

a) an injective but non-surjective function $\{1, 2\} \to \{a, b, c\}$;

b) a surjective but non-injective function $\{1, 2, 3\} \to \{a, b\}$;

c) a function $\{1, 2, 3\} \to \{a, b, c\}$ with neither property.

12.4. For $f : \mathbb{Z} \to \mathbb{Z}$, $f(k) = k - 4$, prove injectivity and surjectivity separately. Give the inverse function, including its domain and codomain, and verify its value at 7.

12.5. Let $f : \mathbb{Z} \to \mathbb{Z}$ be $f(k) = 2k$. Prove that f is injective, but give a codomain element showing it is not surjective. David proposes the inverse rule $y \mapsto y/2$ on all of $\mathbb{Z}$. Explain why it does not define an inverse function $\mathbb{Z} \to \mathbb{Z}$.

12.6. Let $U = \{\text{Alice}, \text{Bob}, \text{Charlie}, \text{David}\}$ and $P = \{\text{Printer 1}, \text{Printer 2}, \text{Printer 3}\}$. Define $p : U \to P$ by

User	Alice	Bob	Charlie	David
p	Printer 2	Printer 1	Printer 2	Printer 1

Find each set:

a) $p(\{\text{Alice}, \text{Charlie}\})$;

b) $p(\{\text{Bob}, \text{Charlie}\})$;

c) $p^{-1}(\{\text{Printer 2}\})$;

d) $p^{-1}(\{\text{Printer 3}\})$;

e) $p^{-1}(\{\text{Printer 1}, \text{Printer 3}\})$.

12.7. For the function p in Exercise 12.6, find $p(\varnothing)$, $p^{-1}(\varnothing)$, $p^{-1}(P)$ and $p(U)$. For each expression, say whether its definition selects inputs or outputs.

12.8. Let $q : \mathbb{Z} \to \mathbb{Z}$ be $q(k) = k^2$. Find the following:

a) $q(\{-3, 0, 3\})$.

b) $q^{-1}(\{9\})$.

c) $q^{-1}(\{0\})$.

d) $q^{-1}(\{-4, 4\})$.

Does q have an inverse function $\mathbb{Z} \to \mathbb{Z}$? Justify your answer using the stated domain and codomain.

12.9. Let $b : \mathbb{Z} \to \mathbb{Z}$, $b(k) = k - 4$. Find $b(7)$, $b(\{7\})$, $b^{-1}(7)$ and $b^{-1}(\{7\})$. Explain the role of the braces in each expression and why $b^{-1}(7)$ is not $1/b(7)$.

12.10. For p in Exercise 12.6, Faith writes $p^{-1}(\{\text{Printer 2}\}) = \text{Alice}$. Identify both the omitted input and the type of object that the answer must be. Give the corrected set and explain why its existence does not give p an inverse function.

12.11. Let $B = \text{Node}(\bullet, \bullet)$. Draw each ordered tree with its root at the top:

a) $\bullet$;

b) B;

c) $\text{Node}(B, \bullet)$;

d) $\text{Node}(\bullet, B)$;

e) $\text{Node}(B, B)$.

For each drawing, identify its leaves and internal vertices.

12.12. Which descriptions belong to $\mathcal{T}$? Justify each answer from the base or constructor rules.

a) A single rooted vertex.

b) A root with exactly one leaf child.

c) A root with two leaf children.

d) A root with three leaf children.

e) A finite full binary tree whose leaves have different depths.

12.13. For $\bullet$, give the root's graph degree, number of children, leaf count and internal count. Why would treating a leaf as only a degree-one vertex give the wrong base value for L on this family?

12.14. Let $B = \text{Node}(\bullet, \bullet)$, $A = \text{Node}(B, \bullet)$ and $C = \text{Node}(A, B)$. Find L and I for B, A, C, showing the substitution into each constructor rule. Check the resulting values against $L = I + 1$.

12.15. Let $B = \text{Node}(\bullet, \bullet)$ and $T = \text{Node}(B, B)$. Trace `CountLeaves(T)` using Example 12.15. Show the two calls on copies of B, the values returned by their leaf calls, and the final output. Why must the second occurrence of B contribute even though it has the same shape as the first?

12.16. The input of this procedure is $T \in \mathcal{T}$.

```
CountInternal(T):
    if T is a single leaf:
        return 0
    else:
        write T as Node(S,R)
        return 1 + CountInternal(S) + CountInternal(R)
```

Specify its function domain and codomain, base value and constructor rule. Trace it on the tree

$$\text{Node}(\bullet, \text{Node}(\bullet, \bullet)).$$

Explain what the added 1 counts.

12.17. Suppose $F : \mathcal{T} \to \mathbb{N}$ satisfies $F(\text{Node}(S, R)) = F(S) + F(R)$, but no base value is supplied. Choose two different base values and calculate $F(B)$ for $B = \text{Node}(\bullet, \bullet)$. Use the two results to explain the missing information.

12.18. A proposed leaf-counting procedure returns 1 at a leaf, but at $\text{Node}(S, R)$ it returns

$$1 + \texttt{CountLeaves(S)} + \texttt{CountLeaves(R)}.$$

Assume both recursive calls use this same proposed rule. Translate it into a recursive function W, evaluate $W(B)$, and compare with the actual leaf count. Explain which quantity the rule counts.

12.19. In the procedure from Example 12.15, what changes when a call passes to a child subtree? Use the finiteness of the input tree to justify termination. Why would replacing the recursive call on a child by another call on the whole unchanged tree fail to provide this justification?

12.20. For a structural-induction proof of $L(T) = I(T) + 1$ on $\mathcal{T}$, state:

a) the property of the base object;

b) the two subtree hypotheses;

c) the target for $\text{Node}(S, R)$;

d) the recursive identity that introduces the new internal root.

12.21. In an attempted independent proof of $L(T) = I(T) + 1$ by structural induction, a student assumes only $L(S) = I(S) + 1$, then writes

$$L(\text{Node}(S, R)) = L(S) + L(R) = (I(S) + 1) + (I(R) + 1).$$

Which replacement is unsupported by the stated hypothesis? Repair the hypotheses and finish the argument. Explain why restricting to $R = S$ would not repair the proof for all trees in $\mathcal{T}$.

12.22. Judy checks $L = I + 1$ for $\bullet$, $\text{Node}(\bullet, \bullet)$ and $\text{Node}(B, B)$, where $B = \text{Node}(\bullet, \bullet)$. Are these checks a structural-induction proof? Identify the general implication still needed.

12.23. For trees in $\mathcal{T}$, find the requested counts.

a) A tree has 12 internal vertices. Find its leaves and total vertices.

b) A tree has 20 leaves. Find its internal vertices and total vertices.

c) Can a tree have 8 leaves and 8 internal vertices?

12.24. Define $V : \mathcal{T} \to \mathbb{N}$ by $V(\bullet) = 1$ and $V(\text{Node}(S, R)) = 1 + V(S) + V(R)$. Use structural induction to prove $V(T) = 2I(T) + 1$ for every $T \in \mathcal{T}$. State both subtree hypotheses and show how the new-root contribution enters the algebra.

12.25. Give a finite rooted tree outside $\mathcal{T}$ for which the ordinary numbers of leaves and internal vertices do not satisfy $L = I + 1$. Explain why it does not contradict the proved theorem. Identify which recursive construction rule would need to change to include your example.

12.26. Consider the counting function $L : \mathcal{T} \to \mathbb{N}$.

a) Construct two distinct ordered trees with the same leaf count. What does this say about injectivity?

b) Show that 0 is not a leaf count. What does this say about surjectivity onto the stated codomain?

c) Describe $L^{-1}(\{2\})$ by identifying every tree in that inverse image.

Use $L = I + 1$ when it helps, and keep inverse images as sets of tree objects.