RECURSIVE DEFINITIONS AND STRUCTURAL INDUCTION

Lecture 13

RECURSION

RECURSIVELY DEFINED FUNCTIONS

Definition: A recursive or inductive definition of a function consists of two steps.

- BASIS STEP: Specify the value of the function at zero.
- RECURSIVE STEP: Give a rule for finding its value at an integer from its values at smaller integers.

- A function f(n) is the same as a sequence a_0, a_1, … , where a_i, where f(i) = a_i.
- We use recursion to define sequence, functions and sets.
- The terms of the sequences of power of 2 was previously defined as $a_n = 2^n \; for \; n = 0, 1, 2 \ldots.$
- We can define terms recursively by specifying how terms of the sequence are formed from the previous terms.
- Thus, the sequence to the power of 2 can be defined by giving the first term $a_0 = 1$ and then a rule for finding a term of the sequence from the previous terms, $a_{n+1} = 2a_n \; for \; n = 0, 1, 2 \ldots$

RECURSIVELY DEFINED FUNCTIONS

Example: Give a recursive definition of: $\sum_{k=0}^{n} a_k.$

Solution: The first part of the definition is $\sum_{k=0}^{0} a_k = a_0.$

The second part is $\sum_{k=0}^{n+1} a_k = \left(\sum_{k=0}^{n} a_k\right) + a_{n+1}.$

RECURSIVELY DEFINED FUNCTIONS

Example: Suppose f is defined by:

$f(0) = 3,$

$f(n + 1) = 2f(n) + 3$

Find $f(1), f(2), f(3), f(4)$

Solution: From the recursive definition it follows that:

- $f(1) = 2f(0) + 3 = 2\cdot 3 + 3 = 9$

PYTHON CODE

```
• def f(n):
•     if n == 0:
•         return 3
•     return 2 * f(n - 1) + 3

• # Compute values
• print("f(1) =", f(1))  # ➞ 9
• print("f(2) =", f(2))  # ➞ 21
• print("f(3) =", f(3))  # ➞ 45
• print("f(4) =", f(4))  # ➞ 93
```

WHAT IS FACTORIAL?

- Definition:
- The factorial of a number n, written as n!, is the product of all positive integers less than or equal to n.

- Examples:
- - 3! = 3 × 2 × 1 = 6
- - 5! = 5 × 4 × 3 × 2 × 1 = 120
- - 0! = 1 (by definition)

RECURSIVELY DEFINED FUNCTIONS

Example: Give a recursive definition of the factorial function $n!$

Solution:

$factorial(0) = 1$

$factorial(n + 1) = (n + 1)\cdot \text{factorial}(n)$

STEP-BY-STEP TRACE OF FACTORIAL

- factorial(4)
- = 4 * factorial(3)
- = 4 * (3 * factorial(2))
- = 4 * (3 * (2 * factorial(1)))
- = 4 * (3 * (2 * (1 * factorial(0))))
- = 4 * (3 * (2 * (1 * 1)))   ← base case!
- = 4 * 3 * 2 * 1 = 24

PYTHON CODE FOR RECURSIVE FACTORIAL

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

TEST IT!

- print(factorial(0)) # ➞ 1
- print(factorial(3)) # ➞ 6
- print(factorial(5)) # ➞ 120

THINGS YOU SHOULD KNOW

- Must have a base case (if n == 0)
- Factorial isn’t defined for negative numbers
- Python has a recursion limit (~1000 calls)

ITERATIVE VERSION OF FACTORIAL

```
• def factorial_iterative(n):
•     result = 1
•     for i in range(2, n + 1):
•         result *= i
•     return result
```

OPTIONAL ERROR HANDLING

- def factorial(n):
-     if n < 0:
-         raise ValueError("Factorial is not defined for negative numbers")
-     if n == 0:
-         return 1
-     return n * factorial(n - 1)

WHAT IS THE FIBONACCI SEQUENCE?

- The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones.

- Starts with:
- 0, 1, 1, 2, 3, 5, 8, 13, ...

RECURSIVE FORMULA FOR FIBONACCI

- Base cases:
- F(0) = 0
- F(1) = 1

- Recursive case:
- F(n) = F(n-1) + F(n-2)

RECURSIVE PYTHON FUNCTION

```
• def fibonacci(n):
•     if n == 0:
•         return 0
•     elif n == 1:
•         return 1
•     else:
•         return fibonacci(n - 1) + fibonacci(n - 2)
```

TRACE EXAMPLE: FIBONACCI(4)

- fibonacci(4)
- = fibonacci(3) + fibonacci(2)
- = (fibonacci(2) + fibonacci(1)) + (fibonacci(1) + fibonacci(0))
- = ((fibonacci(1) + fibonacci(0)) + 1) + (1 + 0)
- = ((1 + 0) + 1) + 1
- = 2 + 1 = 3

WHY RECURSIVE FIBONACCI IS INEFFICIENT

Recalculates the same values many times

- Grows exponentially: O(2^n) time complexity
- Deep recursion leads to stack overflow for large n
- Better approaches: Memoization or Iterative version

IMPROVING WITH MEMOIZATION

- Use a dictionary to store previously computed values:

```
memo = {}
def fibonacci(n):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci(n-1) + fibonacci(n-2)
    return memo[n]
```

ITERATIVE FIBONACCI VERSION

- An efficient way to compute Fibonacci numbers using a loop:

```
• def fibonacci_iterative(n):
•     if n == 0:
•         return 0
•     elif n == 1:
•         return 1
•     a, b = 0, 1
•     for _ in range(2, n + 1):
•         a, b = b, a + b
•     return b
```

RECURSIVELY DEFINED SETS

Recursive definitions of sets have two parts:

- The basis step specifies an initial collection of elements.
- The recursive step gives the rules for forming new elements in the set from those already known to be in the set.

- Sometimes the recursive definition has an **exclusion rule**, which specifies that the set contains nothing other than those elements specified in the basis step and generated by applications of the rules in the recursive step.
- We will always assume that the exclusion rule holds, even if it is not explicitly mentioned.
- We will later develop a form of induction, called structural induction, to prove results about recursively defined sets.

RECURSIVELY DEFINED SETS

Example : Subset of Integers S:

BASIS STEP: $3 \in S$.

RECURSIVE STEP: If $x \in S$ and $y \in S$, then $x + y$ is in S.

- Initially 3 is in S, then $3 + 3 = 6$, then $3 + 6 = 9$, etc.

Example: The natural numbers N.

BASIS STEP: $0 \in N$.

RECURSIVE STEP: If n is in N, then $n + 1$ is in N.

- Initially 0 is in S, then $0 + 1 = 1$, then $1 + 1 = 2$, etc.

RECURSIVELY DEFINED STRUCTURES- ROOTED TREES

The set of rooted trees, where a rooted tree consists of a set of vertices containing a distinguished vertex called the *root,* and edges connecting these vertices, can be defined recursively by these steps:

BASIS STEP: A single vertex *r* is a rooted tree.

RECURSIVE STEP: Suppose that T_1, T_2,……,T_n, are disjoint rooted trees with roots r_1, r_2,……,r_n, respectively. Then the graph formed by starting with a root *r,* which is not in any of the rooted trees T_1, T_2,……,T_n, and adding an edge from *r* to each of the vertices r_1, r_2,……,r_n, is also a rooted tree.

RECURSIVELY DEFINED STRUCTURES- ROOTED TREES

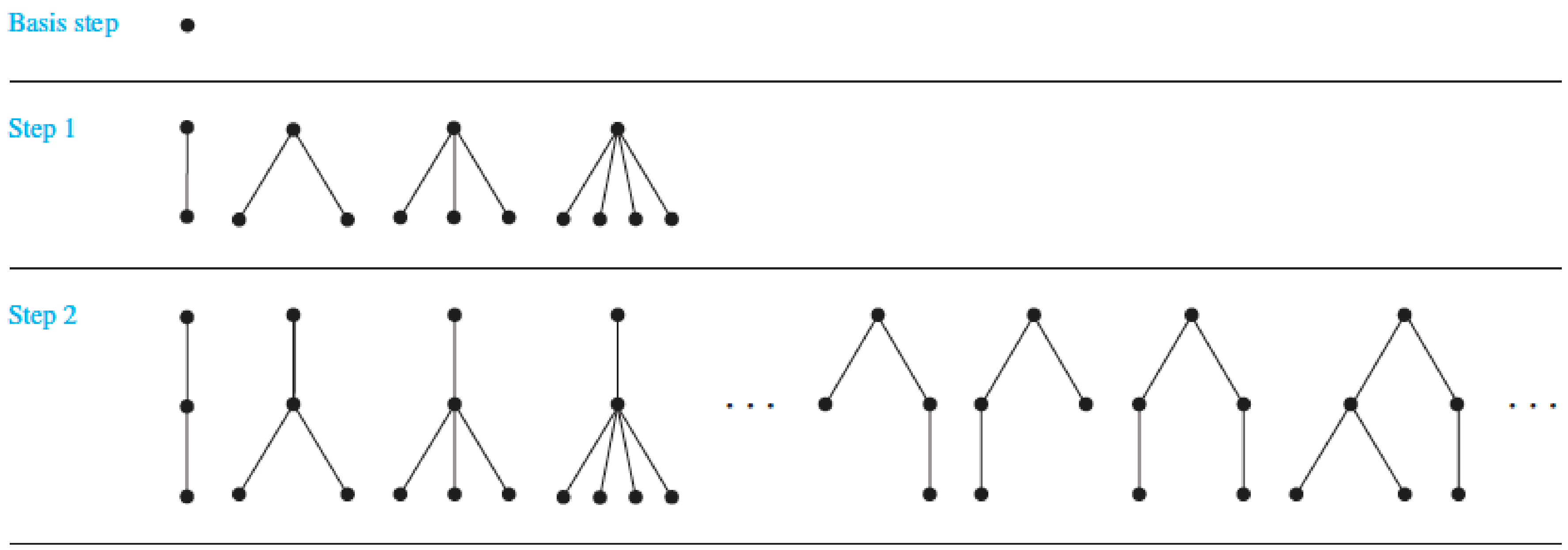

FIGURE 2 Building up rooted trees.

RECURSIVELY DEFINED STRUCTURES- FULL BINARY TREE

The set of extended binary trees, can be defined recursively by these steps:

BASIS STEP: The empty set is an extended binary tree.

RECURSIVE STEP: If T_1 *and* T_2, are disjoint extended binary trees, there is an extended binary tree denoted by $T_1 \cdot T_2$, consisting of a root *r* together with edges connecting the root to each of the root of the left subtree T_1 and the right subtree T_2 when these trees are nonempty.

RECURSIVELY DEFINED STRUCTURES- FULL BINARY TREE

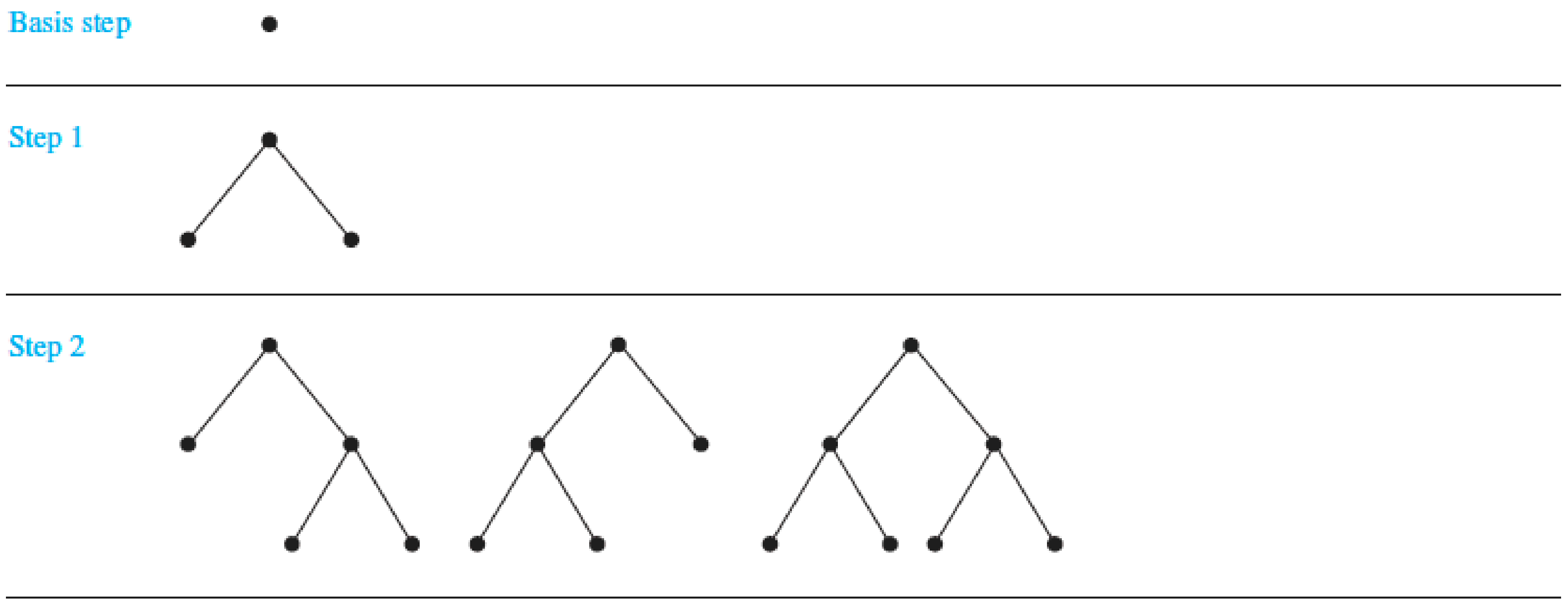

FIGURE 4 **Building up full binary trees.**

RECURSION IN CS

Linked lists

- Data structure used to store a sequence of elements
- Each element in the list is called a node
- Each node stores:
  - Data
  - A pointer to the rest of the list.
- A linked list is either:
  - < > known as null or the empty list; or
  - <*x*, *L*> where x is an arbitrary element, and L is a linked list

STRUCTURAL INDUCTION

Definition: Is a type of Induction used to prove theorems about recursively defined sets that follow the structure of the recursive definition.

To prove a property of the elements of a recursively defined set, we use structural induction.

BASIS STEP: Show that the result holds for all elements specified in the basis step of the recursive definition.

RECURSIVE STEP: Show that if the statement is true for each of the elements used to construct new elements in the recursive step of the definition, the result holds for these new elements.

- The validity of structural induction can be shown to follow from the principle of mathematical induction.