CSIT881
Programming and Data Structures

Introduction to Algorithm and Data Structure

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Dr. Joseph Tonien

Objectives

- Introduction to Algorithm
- Big O notation
- Introduction to Data structure

Algorithm

Encyclopedia Britannica:

An algorithm is a specific procedure for solving a well-defined computational problem.

Cambridge Dictionary:

An algorithm is a set of mathematical instructions or rules that, especially if given to a computer, will help to calculate an answer to a problem.

Oxford Languages:

An algorithm is a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

Algorithm

- **Deterministic algorithm**: always goes through the same internal states and produces the same output result.

- **Randomized algorithm**: uses a source of randomness as part of its logic and obtains the result by chance; output may change between different runs.

Algorithm

Deterministic algorithm:

Given the same input, the algorithm

- takes the same amount of time, memory, resources to run
- follows the same programming instructions at each execution
- produces the same result output

Algorithm

Randomized algorithm:

Given the same input, the algorithm

- may take different amount of time, memory, resources to run at different time
- may follow different path of programming instructions at different execution
- may produce different result output

Algorithm

Randomized algorithm:

- Monte Carlo algorithm: a randomized algorithm whose **output may be incorrect** with a certain probability.
- Las Vegas algorithm: a randomized algorithm that always gives correct results.

Algorithm

Example. (Primality Test Problem) Given an integer n>1, determine whether n is a prime number or a not?

Simple Division Test Algorithm (deterministic algorithm)

- For each integer k from 2 to (n-1), test to see if n divide k or not. If n divides **any** k then n is a composite number. Otherwise, n is a prime number.

pseudocode

```
FOR k from 2 to (n-1)
    IF n divides k
        RETURN n is NOT prime
    END IF
END FOR

RETURN n is prime
```

An improvement: only need to test for k from 2 to sqrt(n)

Algorithm

Example. (Primality Test Problem) Given an integer n>1, determine whether n is a prime number or a not?

Fermat primality test (Monte Carlo randomized algorithm) based on the following *Fermat's little theorem*:

```
if n is a prime then for any 0 < k < n,
```

$$k^{n-1} - 1 \text{ is divisible by } n.$$

- Select a random integer $0 < k < n$ and test if $k^{n-1} - 1$ is divisible by n or not. If not then n is not a prime, otherwise, n is a prime.

Algorithm

Fermat primality test

based on the following *Fermat's little theorem*:

if n is a prime then for any $0 < k < n$,

$k^{n-1} - 1$ is divisible by n.

pseudocode

```
k = random from 1 to (n-1)
r = $k^{n-1}$ modulo n
IF r != 1
    RETURN n is NOT prime
END IF

RETURN n is prime
```

Why this algorithm may output incorrect result?

Algorithm

Fermat primality test

based on the following *Fermat's little theorem*:

if n is a prime then for any $0 < k < n$,

$k^{n-1} - 1$ is divisible by n.

Why this algorithm may output incorrect result?

- if $k^{n-1} != 1$ (mod n) then n is **definitely** not a prime
- is $k^{n-1} = 1$ (mod n) then n **may be** a prime and n **may not be** a prime - we don't know for sure, for example, $n = 13 \times 17 = 221$ is not a prime, but for $k = 38$ we have $38^{220} = 1$ (mod 221)

Algorithm

Fermat primality test

An improvement: to reduce the chance that the program outputs incorrect result, we may run it many times

pseudocode

```
REPEAT t times
    k = random from 1 to (n-1)
    r = $k^{n-1}$ modulo n
    IF r != 1
        RETURN n is NOT prime
    END IF

RETURN n is prime
```

Algorithm

Algorithm analysis: determine efficiency of an algorithm by looking at computational resource used by the algorithm:

- running time: how long does the algorithm take to complete?
- memory usage: how much working memory is needed by the algorithm?

Algorithm

The computational resource (running time, memory usage) used by the algorithm depends on the input of the algorithm.

- For some input, the algorithm may run very fast; but for other input, it may run very long.
- For some input, the algorithm may use a lot of memory; but for other input, it may use less memory.

Algorithm

Algorithm analysis:

- **worst-case complexity**: measures computational resource in the worst case scenario, i.e. in the case it requires the longest running time (or largest memory usage)

- **best-case complexity**: measures computational resource in the best case scenario, i.e. in the case it requires the shortest running time (or least amount of memory)

- **average-case complexity**: measures computational resource used by the algorithm, averaged over all possible inputs

Algorithm

Big O notation:

As the input size of the algorithm grows, the computational resource (running time, memory usage) used by the algorithm also grows.

We use big O notation to express the growing value of the computational resource as a function of the input size.

- $O(1)$: constant
- $O(\log n)$: logarithmic $\longrightarrow$ measure the bit length of n
- $O(n)$: linear
- $O(n^2)$: quadratic

Algorithm

Big O notation:

Mathematical definition:

$f(n) = O(g(n))$ if there exists a constant c such that

$f(n) < c\, g(n)$ when n tends to infinity.

Remarks:

- only consider large values of n when working with big O notation, ignore small values of n;
- ignore multiplicative constant;
- only consider the largest term in a sum.

Algorithm

Big O notation:

Example: suppose that the running time of an algorithm on input size n is ($3 n^2 + 100 n + 2000$), write this running time in big O notation.

- only consider large values of n when working with big O notation, ignore small values of n:
  - $100 n < n^2$ (when n tends to infinity)
  - $2000 < n^2$ (when n tends to infinity)

- only consider the largest term in a sum:
  - $3 n^2 + 100 n + 2000 = O(3 n^2 + n^2 + n^2) = O(5 n^2)$

- ignore multiplicative constant:
  - $3 n^2 + 100 n + 2000 = O(5 n^2) = O(n^2)$

Algorithm

Big O notation:

$1 < \log(\log(n)) < \log(n) < n < n \log(n) < n^2 < n^2 \log(n) < n^3 < n^3 \log(n) < n^4 < n^5 < n^6 < \ldots$

- $10000 = O(1)$
- $4 n + 7 = O(n)$
- $n + 3 \log(n) + 100 = O(n)$
- $5 \log(n) + 100 = O(\log(n))$
- $3 n^2 + 7 n + 8 = O(n^2)$
- $4 n \log(n) + 3 n^2 + n = O(n^2)$

Algorithm

Example: write a program to search for an index at which the two lists of integers having the same number.

Input: two lists of integers of the same length n

Output: the first index at which the two lists have the same number, return -1 if not found

Example: if list1 is [4, 6, -3, **7**, 1, 5]

and list2 is [8, -6, 8, **7**, 4, 5] then the matching index is 3

Algorithm

Example: write a program to search for an index at which the two lists of integers having the same number.

Consider the following two algorithms:

pseudocode

```
Function algorithm1(list1, list2)
{
    n = length of list1

    index = -1
    FOR i from 0 to (n-1)
        IF list1[i] = list2[i] and index = -1
            index = i
        END IF
    END FOR

    RETURN index
}
```

Algorithm

pseudocode

```
Function algorithm2(list1, list2)
{
    n = length of list1

    FOR i from 0 to (n-1)
        IF list1[i] = list2[i]
            RETURN i
        END IF
    END FOR

    RETURN -1
}
```

These two algorithms have different **best-case complexity**, but the same **worst-case complexity** and **average-case complexity**. Why?

Algorithm

```
Function algorithm1(list1, list2)
{
    n = length of list1

    index = -1
    FOR i from 0 to (n-1)
        IF list1[i] = list2[i] and index = -1
            index = i
        END IF
    END FOR

    RETURN index
}
```

For any input, the program will run the whole loop

`FOR i from 0 to (n-1)`

worst-case complexity: running time O(n)

best-case complexity: running time O(n)

average-case complexity: running time O(n)

Algorithm

```
Function algorithm2(list1, list2)
{
    n = length of list1

    FOR i from 0 to (n-1)
        IF list1[i] = list2[i]
            RETURN i
        END IF
    END FOR

    RETURN -1
}
```

worst-case complexity: running time O(n)

- when no matching found, or
- a matching found at the end of the lists

best-case complexity: running time O(1)

- when matching found at the beginning of the lists

Algorithm

```
Function algorithm2(list1, list2)
{
    n = length of list1

    FOR i from 0 to (n-1)
        IF list1[i] = list2[i]
            RETURN i
        END IF
    END FOR

    RETURN -1
}
```

average-case complexity: running time O(n)

Matching found at index 0 → running time 1

Matching found at index 1 → running time 2

Matching found at index 2 → running time 3...

Matching found at index n-1 → running time n

No matching found → running time n+1

$(1 + 2 + 3 + \ldots + n + (n+1))/(n+1) = (n+2)/2 \quad \Rightarrow \quad O(n)$

Data structure

A **data structure** is a formal structure for the organization of information:

- a collection of data values;
- the relationships among them;
- the operations that can be applied to the data.

Examples of data structure: Array, Linked List, Doubly Linked List, Stack, Queue, Binary Tree, Hash Table, ...

Data structure

For the same data structure (say Binary Tree), different programming languages may provide different implementations:

- class name may be different between programming languages;
- class name may not be the same as the original data structure name;
- functions name may be different between programming languages;
- some functions are implemented in one programming language but not in the others; etc...

With the diversity of programming language implementations, how can we provide a unified solution to a particular problem using data structure?

Data structure

With the diversity of programming language implementations, how can we provide a unified solution to a particular problem using data structure?

Strategy: using abstract data type

An **abstract data type** (ADT) is a data structure model characterised by its functionalities and behaviors from the point of view of a user.

With abstract data type, we can describe an algorithm to solve a particular problem using pseudocode.

Data structure

An **abstract data type** (ADT) is a data structure model characterised by its functionalities and behaviors from the point of view of a user.

Example: A stack is an abstract data type with the following operations:

- push(item): add an item onto the top of the stack;
- pop(): remove the item from the top of the stack and return it;
- top(): look at the item at the top of the stack, but do not remove it.

Data structure

Example:
Stack

Different programming languages (Java, C++, Python, etc.) may provide different implementations of this (ADT) stack.

The implementation classes may have different names, but that is NOT important.

The important thing is that they all have these basic operations: push(item), pop(), and top() with the expected behavior *Last-In-First-Out*

Data structure

Working with abstract data types:

- we are only concerned with **the behaviors** of a data structure and **what operations** we can do with the data structure to solve the problem at hand;
- not really concerned about *how it is actually implemented* under the hood;
- in software development, we need to look at the **API** to learn about the operations and behaviors of an abstract data type; we do not really need to look at the **implementation** details;
- study algorithm with abstract data types and pseudocode helps us to implement solution in any programming language.

Data structure

To decide to use the data structure or not?

Ask yourself this question:

- with the **behaviors** of the data structure and the **operations** we are allowed to use with this data structure, can we **efficiently** solve the problem that we need to solve?

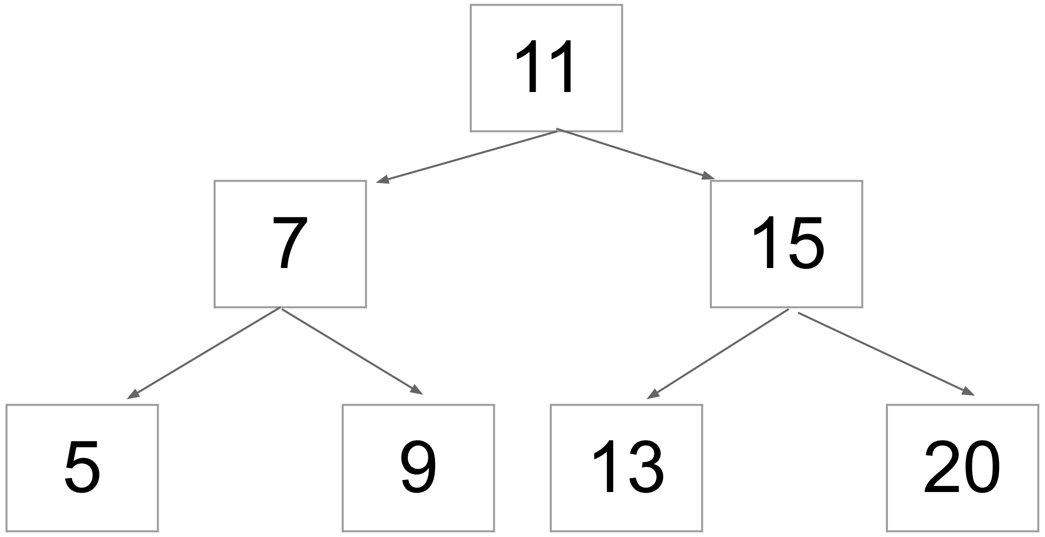
Data structure

Examples of using data structure to solve problem:

- Parenthesis checking using Stack

```
4 * {z - [(a+b) * c]}
```

- Searching using Binary search tree



Example: Parenthesis checking using Stack

In mathematics, we use different types of parenthesis, such as (,), {, }, [,], to write an expression

```
4 * {z - [(a+b) * c]}
```

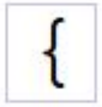
We can use Stack to check the validity of these expressions to make sure every open parentheses matches with a closed parentheses.

Example: Parenthesis checking using Stack

```
4 * {z - [(a+b) * c]}
    ^
```

encounter open symbol

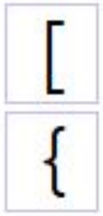
push it in Stack



```
4 * {z - [(a+b) * c]}
         ^
```

encounter open symbol

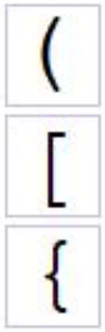
push it in Stack



```
4 * {z - [(a+b) * c]}
          ^
```

encounter open symbol

push it in Stack

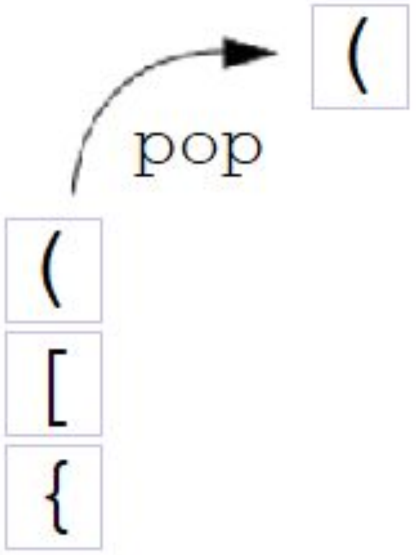


Example: Parenthesis checking using Stack

```
4  *  {z  -  [(a+b)  *  c]}
```

^
encounter closed symbol

pop the Stack and compare



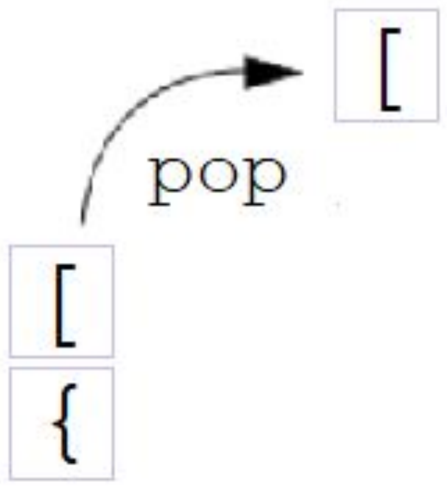
Yes it matches!

Example: Parenthesis checking using Stack

```
4 * {z - [(a+b) * c]}
```

^
encounter closed symbol

pop the Stack and compare



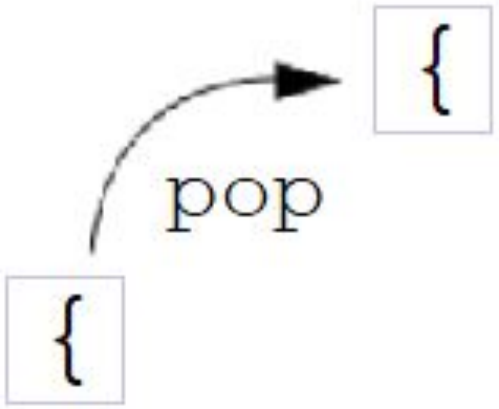
Yes it matches!

Example: Parenthesis checking using Stack

```
4 * {z - [(a+b) * c]}
                    ^
```

encounter closed symbol

pop the Stack and compare

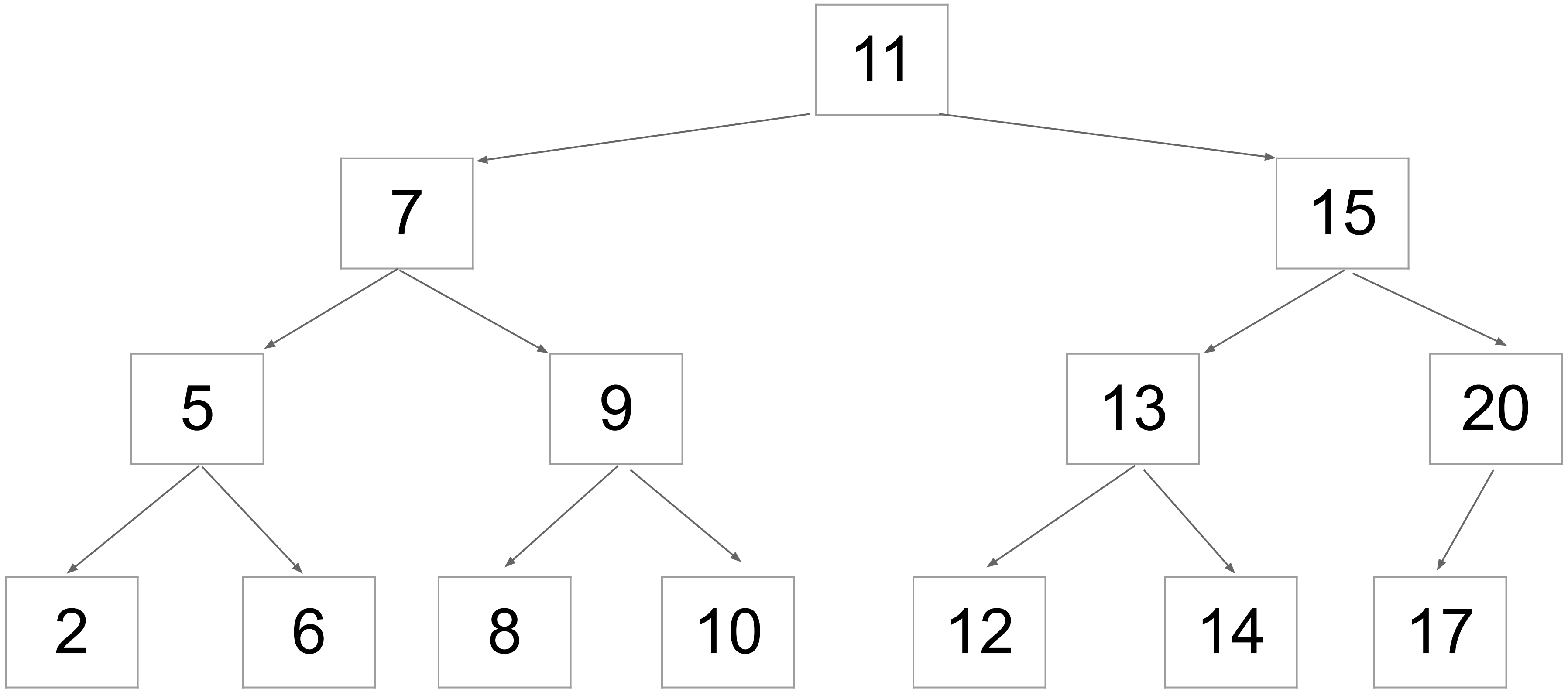


Yes it matches!

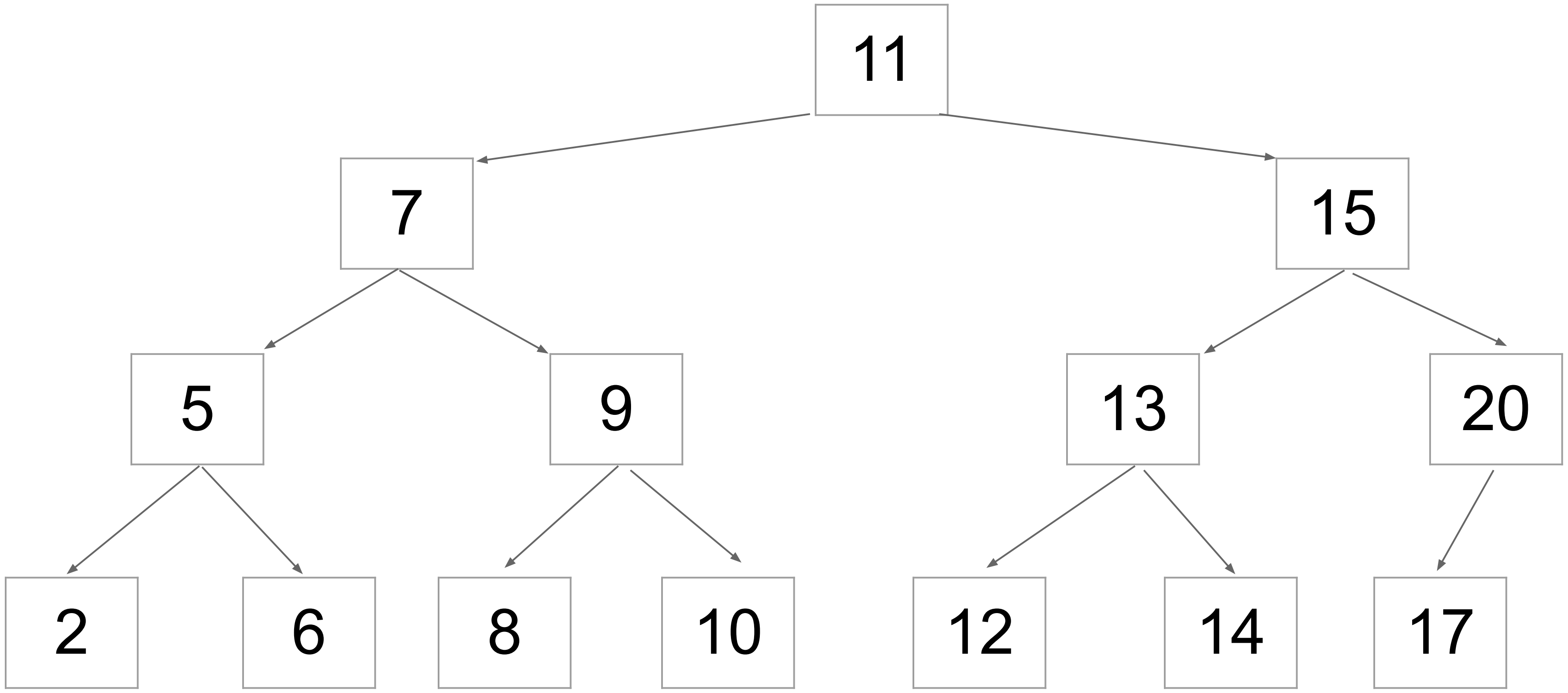
Checking parenthesis DONE!

Example: Searching using Binary search tree

In a binary search tree, each node stores a key **greater than** all the keys in the node’s **left subtree** and **less than** those in its **right subtree**.



Example: Searching using Binary search tree

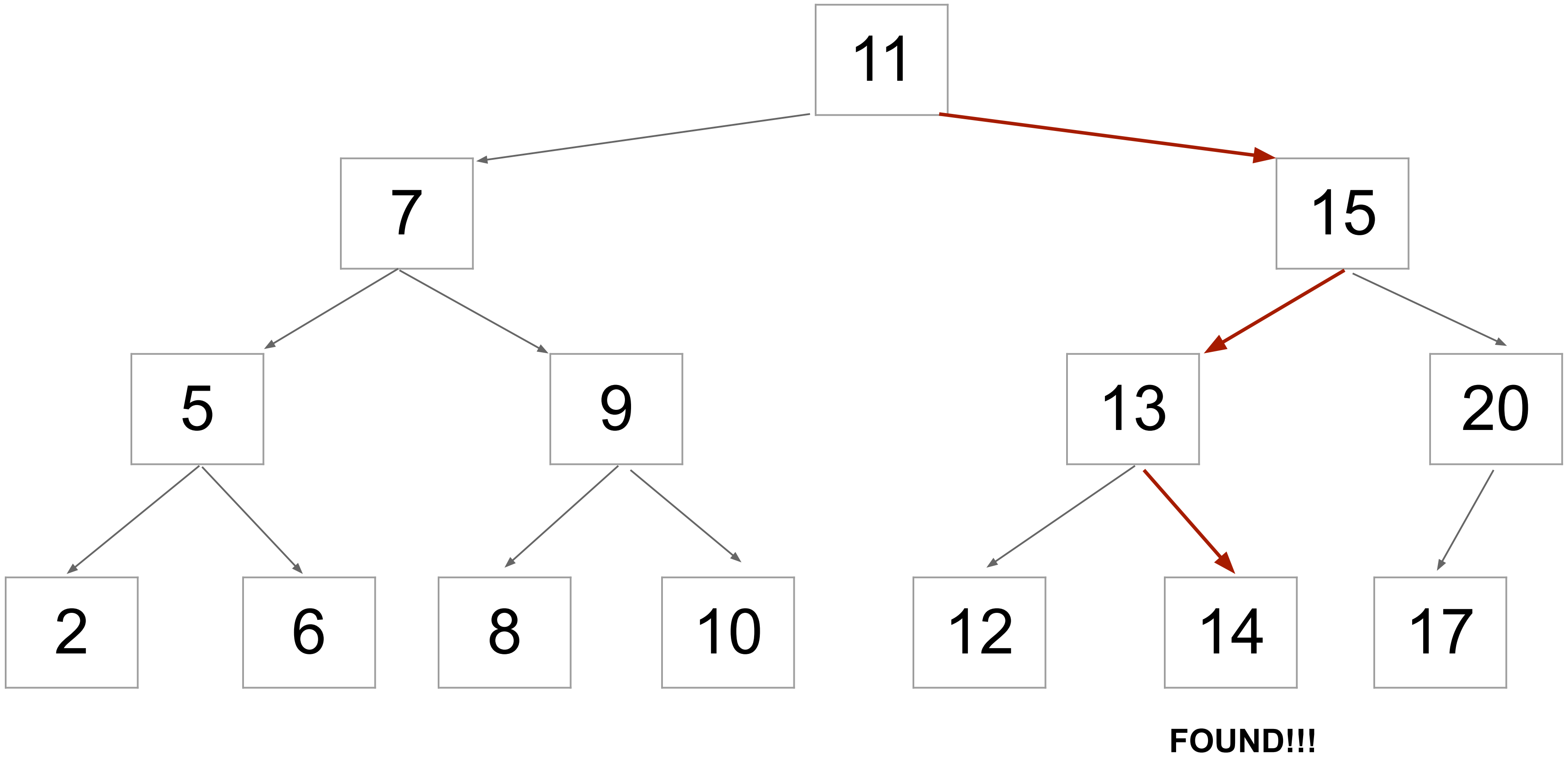


Notice that:

- All nodes on the **left** of 7 are **less than** 7;
- All nodes on the **right** of 15 are **greater than** 15; etc...

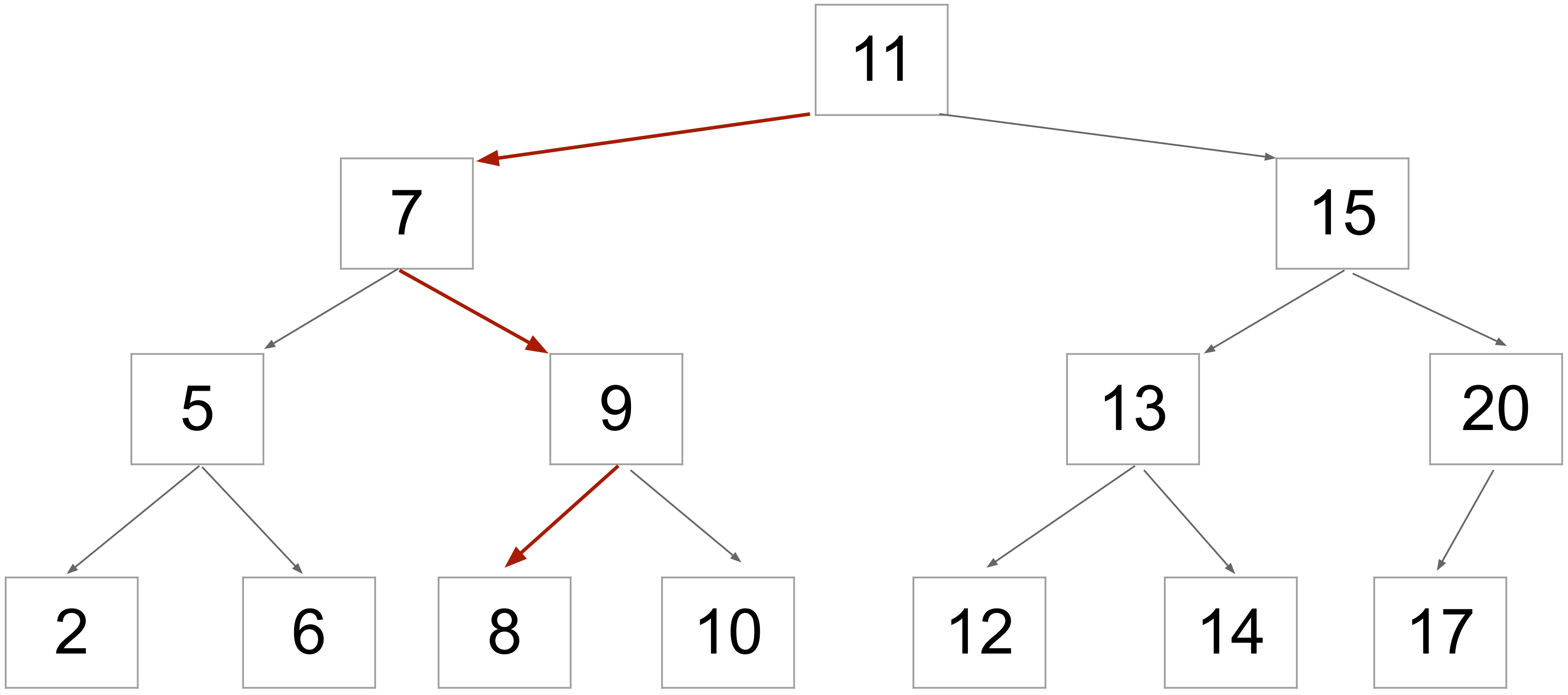
Example: Searching using Binary search tree

Searching for 14:



Example: Searching using Binary search tree

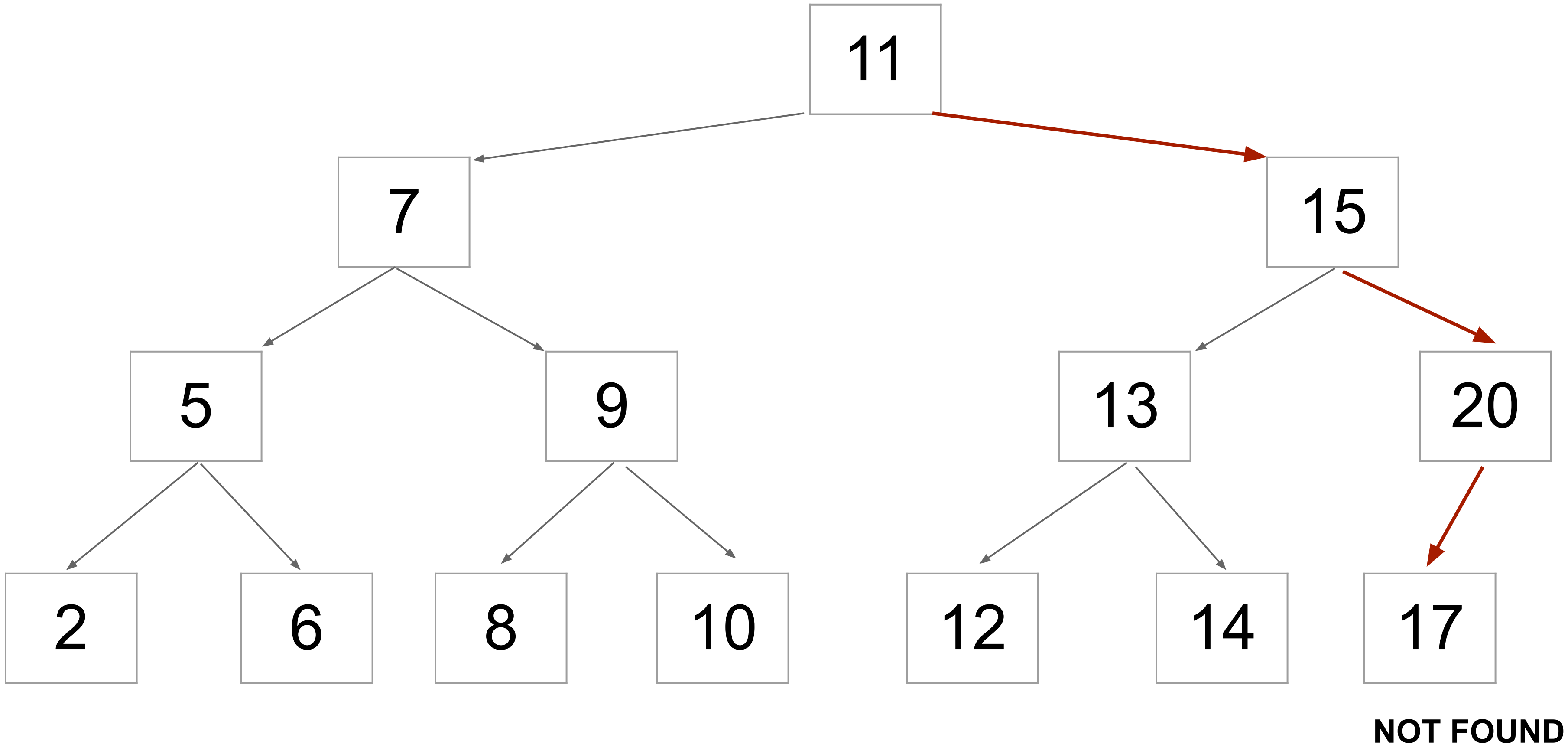
Searching for 8:



FOUND!!!

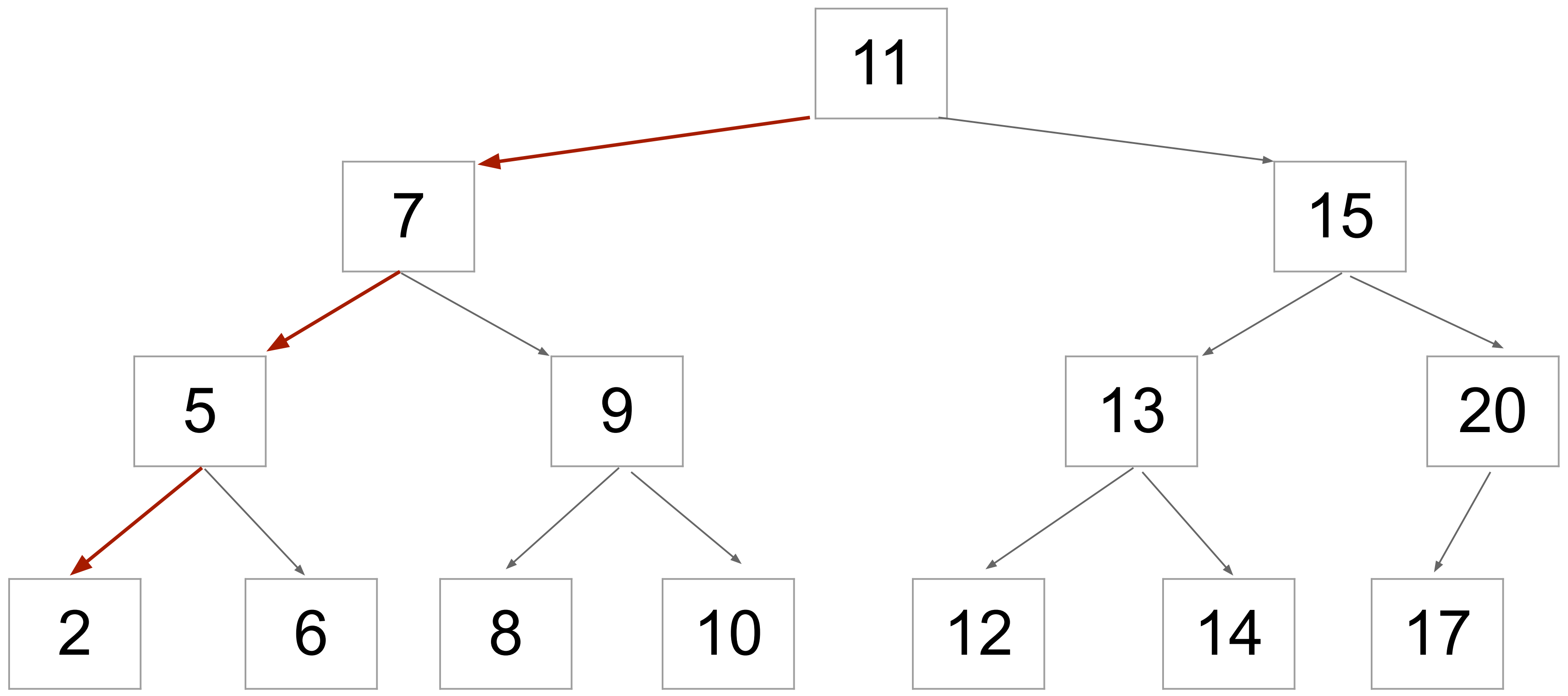
Example: Searching using Binary search tree

Searching for 16:



Example: Searching using Binary search tree

Searching for 3:



NOT FOUND

References

- Python 3 documentation
  https://docs.python.org/3/
- NumPy Reference
  https://numpy.org/doc/stable/reference/