Since 2011

Realistic Infotech Group

Our outcomes are over 5000 trainees.

Get IT Right from RIG

Artificial Intelligence Engineering
(Level-1)

Learn, Create, and Shine

Digital Space

Level-1

- Module 1: Introduction to AI and Machine Learning
- Module 2: Linear Algebra, Statistics and Probability for AI
- Module 3: Neural Network Architecture
- Module 4: Building Machine Learning Models
- Module 5: Deep Learning Concepts
- Module 6: Python Data Structure
- Module 7: Data Handling with Pandas and NumPy
- Module 8: Python for AI
- Module 9: Classification AI Project
- Module 10: Prediction AI Project

Artificial Intelligence Engineering (Level-1)

Module 6: Python Data Structure

Content

- Why Python Data Structure needs?
- List
- Sets
- Tuples
- Dictionary
- Linked Lists
- Binary Tree
- Graphs

Learning Outcomes

- Lists: Understand the structure, properties, and versatile operations of Python lists for dynamic data storage and manipulation.
- Sets: Learn the characteristics of Python sets, including unordered collections and operations like union, intersection, and difference.
- Tuples: Gain knowledge of Python tuples as immutable sequences and their efficient use in fixed data storage scenarios.
- Dictionaries: Master the use of Python dictionaries for key-value pair data management, including efficient lookups and updates.
- Linked Lists: Understand the concept of linked lists, their implementation in Python, and their advantages over arrays for dynamic memory allocation.

Learning Outcomes

- Binary Trees: Learn to construct and traverse binary trees and their applications in hierarchical data representation.
- Graphs: Explore graph data structures, including nodes and edges, and understand their implementation and traversal techniques in Python.
- Problem-Solving Skills: Develop skills to solve real-world problems using these data structures effectively.
- Algorithm Efficiency: Analyze the time and space complexities of operations performed on these data structures.
- Practical Applications: Implement these structures in Python to build efficient, scalable, and real-world applications.

Why Python Data Structure needs?

- In AI projects, using the right data structures in Python is crucial for efficient data processing, model building, and performance optimization.
  - Efficient data handling and processing
  - Data preprocessing and feature engineering
  - Storing and managing model parameters
  - Graph structures for neural network
  - Optimizing search and retrieval
  - Handling sequential data

List

- The most basic data structure in Python is the sequence.
- **Each element of a sequence** is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.
- Python has **six built-in types of sequences**.
- The list is a **most versatile datatype** available in Python
- Items in a list need **not be of the same type**.
- Creating a list is as simple as putting different **comma-separated values between square brackets**.

Basic List Operations

- Lists respond to the **+ and * operators** much like strings; they mean **concatenation and repetition**, except that **the result is a new list**, not a string.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

List Functions & Methods

Function	Description
cmp(list1, list2) or list1==list2	Compares elements of both lists.
len(list)	Gives the total length of the list.
max(list)	Returns item from the list with max value.
min(list)	Returns item from the list with min value
list(seq)	Converts a tuple into list.

List Functions & Methods

Method	Description
list.append(obj)	Appends object obj to list
list.count(obj)	Returns count of how many times obj occurs in list
list.extend(seq)	Appends the contents of seq to list
list.index(obj)	Returns the lowest index in list that obj appears
list.insert(index, obj)	Inserts object obj into list at offset index

List Example

```
languages = ["Python", "C", "C++", "Java", "Perl"]
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print ("list1[0]: ", list1[0])
 print ("list2[1:5]: ", list2[1:5])
for lan in languages:
print ("A programming language : %s" % lan)
#Updating List Elements
languages[1]="Android“
 print ("languages[0:4]: ", languages[0:4])
 #Delete List Elements
del list2[0]
 print ("list2[0:5]: ", list2[0:5])
 print("Length = ",len(languages))
print("list1+list2 = ",list1+list2)
print("list1 * 4 = ",list1*4)
 print ("3 in list2", 3 in list2)
 print("list1[-2] : ",list1[-2])
 print("list1[1:] : ",list1[1:])
```

Output

```
OneDrive/Desktop/python_oct/test.py
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
A programming language : Python
A programming language : C
A programming language : C++
A programming language : Java
A programming language : Perl
languages[0:4]:  ['Python', 'Android', 'C++', 'Java']
list2[0:5]:  [2, 3, 4, 5, 6]
Length =  5
list1+list2 =  ['physics', 'chemistry', 1997, 2000, 2, 3, 4, 5, 6, 7]
list1 * 4 =  ['physics', 'chemistry', 1997, 2000, 'physics', 'chemistry', 1997, 2
000, 'physics', 'chemistry', 1997, 2000, 'physics', 'chemistry', 1997, 2000]
3 in list2 True
list1[-2] :  1997
list1[1:] :  ['chemistry', 1997, 2000]
```

Sets

- A Python set data structure is a **non-duplicate data collection** that is **modifiable**.
- Sets are mainly used for **membership screening and removing redundant entries**.
- These processes use the **Hashing data structure**, a popular method for traversal, insertion, and deletion of elements that typically takes O(1) time.

Set Example

```
# Creating a Python set
Set = {"Python", "Data", "Structures", "Tutorial"}
 print("The Python Set is: ")
print(Set)
# Accessing the set elements
 for ind, i in enumerate(Set):
 print(ind, i)
# Finding the intersection of two sets
Set_ = {1, 2, "Python", "Data"}
print("Intersection: ", Set.intersection(Set_))
 # Union of two sets
print("Union: ", Set.union(Set_))
```

Output

```
The Python Set is:
{'Structures', 'Data', 'Tutorial', 'Python'}
0 Structures
1 Data
2 Tutorial
3 Python
Intersection:  {'Data', 'Python'}
Union:  {1, 2, 'Structures', 'Data', 'Tutorial', 'Python'}
```

Tuples

- A tuple is a **sequence of immutable Python objects**.
- Tuples are **sequences**, just like lists.
- The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.
- **Cannot add elements** to a tuple. Tuples have **no append or extend method**.
- **Cannot remove elements** from a tuple. Tuples have **no remove or pop method**.
- **Cannot find elements in a tuple**. Tuples have no index method.

Tuple Example1

```python
# Zero-element tuple.
a = ()
# One-element tuple.
b = ("one",)
# Two-element tuple.
c = ("one", "two")
print(a)
print(len(a))
print(b)
print(len(b))
print(c)
print(len(c))
```

Output

```
()
0
('one',)
1
('one', 'two')
2
```

Tuple Example2

Output

```
# tuple, immutable
tuple = ('cat', 'dog', 'mouse')
# This causes an error.
  #tuple[0] = 'feline’
```

```
Traceback (most recent call last):
  File "c:\Users\User\OneDrive\Desktop\python_oct\test.py", line 16, in <mod
    tuple[0] = 'feline'
    ~~~~~^^^
TypeError: 'tuple' object does not support item assignment
```

```
# Create packed tuple.
pair = ("dog", "cat")
# Unpack tuple.
(key, value) = pair
print(key)
print(value)
```

```
dog
cat
```

Tuple Example3

```
#no parenthesis
# A trailing comma indicates a tuple.
one_item = "cat",
# A tuple can be specified with no parentheses.
two_items = "cat", "dog"
print(one_item)
print(two_items)
```

Output

```
('cat',)
('cat', 'dog')
```

```
# Max and min for strings.
friends = ("sandy", "michael", "aaron", "stacy")
print(max(friends))
 print(min(friends))
# Max and min for numbers.
earnings = (1000, 2000, 500, 4000)
print(max(earnings))
print(min(earnings))
```

```
stacy
aaron
4000
500
```

Tuple Example4

```
# Search for a value.
if "cat" in pair:
    print("Cat found")
# Search for a value not present.
if "bird" not in pair:
    print("Bird not found")
```

Output

```
Cat found
Bird not found
```

```
# Three-item tuple.
items = ("cat", "dog", "bird")
# Get index of element with value "dog".
index = items.index("dog")
print(index, items[index])
```

```
1 dog
```

Differences between Lists and Tuples

Aspect	List	Tuple
Definition	Mutable, ordered collection of items.	Immutable, ordered collection of items.
Syntax	Defined with square brackets `[ ]`.	Defined with parentheses `( )`.
Mutability	Mutable: Items can be added, removed, or changed.	Immutable: Items cannot be modified after creation.
Performance	Slower because of mutability overhead.	Faster due to immutability.
Use Case	Used for collections that need frequent updates.	Used for fixed collections of data that should not change.
Methods	Many methods, such as `append()`, `extend()`, `pop()`, etc.	Limited methods like `count()` and `index()`.
Memory Usage	Consumes more memory because of mutability.	Consumes less memory due to immutability.
Hashability	Not hashable (cannot be used as dictionary keys).	Hashable (if all elements are hashable).
Iteration Speed	Slower iteration due to dynamic nature.	Faster iteration due to immutability.
Immutability Advantage	No, items can be modified, which may lead to unintended changes.	Yes, immutability ensures data integrity.

Dictionary

- Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces.
- An empty dictionary without any items is written with just two curly braces, like this: {}.
- Keys are unique within a dictionary while values may not be.
- The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.
- To access dictionary elements, square brackets can be used along with the key to obtain its value.

Dictionary

- Can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry.
  - Can either remove individual dictionary elements or clear the entire contents of a dictionary.
    - Can also delete entire dictionary in a single operation.

Dictionary Example

```python
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print ("dict['Name']: ", dict['Name'])
print ("dict['Age']: ", dict['Age'])
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry
print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```

Output

```
dict['Name']:  Zara
dict['Age']:  7
dict['Age']:  8
dict['School']:  DPS School
```

Dictionary Example

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print ("dict['Name']: ", dict['Name'])
print ("dict['Age']: ", dict['Age'])
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry
print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```

RIG
Realistic Infotech Group

dict

Key		Value
Name		Zara
Age		7
Class		First

dict

Key		Value
Name		Zara
Age		8
Class		First
School		DPS School

Properties of Dictionary

- Dictionary values have no restrictions.
- Dictionary values can be any arbitrary Python object, either standard objects or user defined objects. However, same is not true for the keys.
- There are two important points in using dictionary keys
- More than one entry per key not allowed. No duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins.
- Keys must be immutable. Can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed.

Dictionary Functions & Methods

Function	Description
cmp(dict1, dict2)	Compares elements of both dict.
len(dict)	Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
str(dict)	Produces a printable string representation of a dictionary
type(variable)	Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

Dictionary Functions & Methods

Method	Description
dict.clear()	Removes all elements of dictionary dict.
dict.copy()	Returns a shallow copy of dictionary dict
dict.fromkeys()	Create a new dictionary with keys from seq and values set to value.
dict.get(key, default=None)	For key key, returns value or default if key not in dictionary
dict.has_key(key)	Returns true if key in dictionary dict, false otherwise
dict.items()	Returns a list of dict's (key, value) tuple pairs
dict.keys()	Returns list of dictionary dict's keys
dict.setdefault (key,default=None)	Similar to get(), but will set dict[key]=default if key is not already in dict
dict.update(dict2)	Adds dictionary dict2's key-values pairs to dict .

Dictionary Example-2

```
# Create a dictionary
person = {
    "name": "Alice",
    "age": 25,
    "skills": ["Python", "Machine Learning"],
    "active": True
}

# Properties and Functions
print("Initial Dictionary:", person)
print("Number of key-value pairs:", len(person))
print("Type of the object:", type(person))
```

Output

Initial Dictionary: {'name': 'Alice', 'age': 25, 'skills': ['Python', 'Machine Learning'], 'active': True}
Number of key-value pairs: 4
Type of the object: <class 'dict'>

Dictionary Example-2

```
# Accessing keys, values, and items
print("\nKeys:", person.keys())
print("Values:", person.values())
print("Items:", person.items())


# Accessing a value using get()
print("\nName:", person.get("name"))
print("Address (default):",
person.get("address", "Not Available"))
```

Output

Keys: dict_keys(['name', 'age', 'skills', 'active'])
Values: dict_values(['Alice', 25, ['Python', 'Machine Learning'], True])
Items: dict_items([('name', 'Alice'), ('age', 25), ('skills', ['Python', 'Machine Learning']), ('active', True)])

Name: Alice
Address (default): Not Available

Dictionary Example-2

```
# Adding and updating key-value pairs
person["location"] = "New York"
print("\nAfter Adding Location:", person)
person.update({"age": 26, "job": "Engineer"})
print("After Updating Age and Adding Job:",
person)

# Removing items
removed_age = person.pop("age")  # Removes key
'age'
print("\nRemoved Age:", removed_age)
print("After Removing Age:", person)
```

Output

After Adding Location: {'name': 'Alice', 'age': 25, 'skills': ['Python', 'Machine Learning'], 'active': True, 'location': 'New York'}
After Updating Age and Adding Job: {'name': 'Alice', 'age': 26, 'skills': ['Python', 'Machine Learning'], 'active': True, 'location': 'New York', 'job': 'Engineer'}

Removed Age: 26
After Removing Age: {'name': 'Alice', 'skills': ['Python', 'Machine Learning'], 'active': True, 'location': 'New York', 'job': 'Engineer'}

Dictionary Example-2

```
last_item = person.popitem()  # Removes the
last key-value pair
print("Removed Last Item:", last_item)
print("After popitem():", person)

# Using setdefault
person.setdefault("country", "USA")
print("\nAfter Using setdefault for
'country':", person)
```

Output

Removed Last Item: ('job', 'Engineer')
After popitem(): {'name': 'Alice', 'skills': ['Python', 'Machine Learning'], 'active': True, 'location': 'New York'}

After Using setdefault for 'country': {'name': 'Alice', 'skills': ['Python', 'Machine Learning'], 'active': True, 'location': 'New York', 'country': 'USA'}

Dictionary Example-2

```
# Shallow copy
person_copy = person.copy()
print("\nShallow Copy of the Dictionary:",
person_copy)


# Iterating through the dictionary
print("\nIterating through Dictionary:")
for key, value in person.items():
    print(f"Key: {key}, Value: {value}")
```

Output

Shallow Copy of the Dictionary: {'name': 'Alice', 'skills': ['Python', 'Machine Learning'], 'active': True, 'location': 'New York', 'country': 'USA'}

Iterating through Dictionary:
Key: name, Value: Alice
Key: skills, Value: ['Python', 'Machine Learning']
Key: active, Value: True
Key: location, Value: New York
Key: country, Value: USA

Dictionary Example-2

```
# Nested Dictionary Example
nested_dict = {
    "person1": {"name": "Alice", "age": 25},
    "person2": {"name": "Bob", "age": 30}
}
print("\nNested Dictionary:", nested_dict)
print("Accessing Nested Value:",
nested_dict["person1"]["name"])


# Clearing the dictionary
person.clear()
print("\nAfter Clearing Dictionary:", person)
```

Output

Nested Dictionary: {'person1': {'name': 'Alice', 'age': 25}, 'person2': {'name': 'Bob', 'age': 30}}
Accessing Nested Value: Alice

After Clearing Dictionary: {}

Dictionary Example-2

RIG
Realistic Infotech Group

```
# Nested Dictionary Example
nested_dict = {
    "person1": {"name": "Alice", "age": 25},
    "person2": {"name": "Bob", "age": 30}
}
print("\nNested Dictionary:", nested_dict)
print("Accessing Nested Value:",
nested_dict["person1"][“age"])

# Clearing the dictionary
person.clear()
print("\nAfter Clearing Dictionary:", person)
```

nesteddict

Key		Value
person1		
person2		

Key		Value
name		Alice
age		25

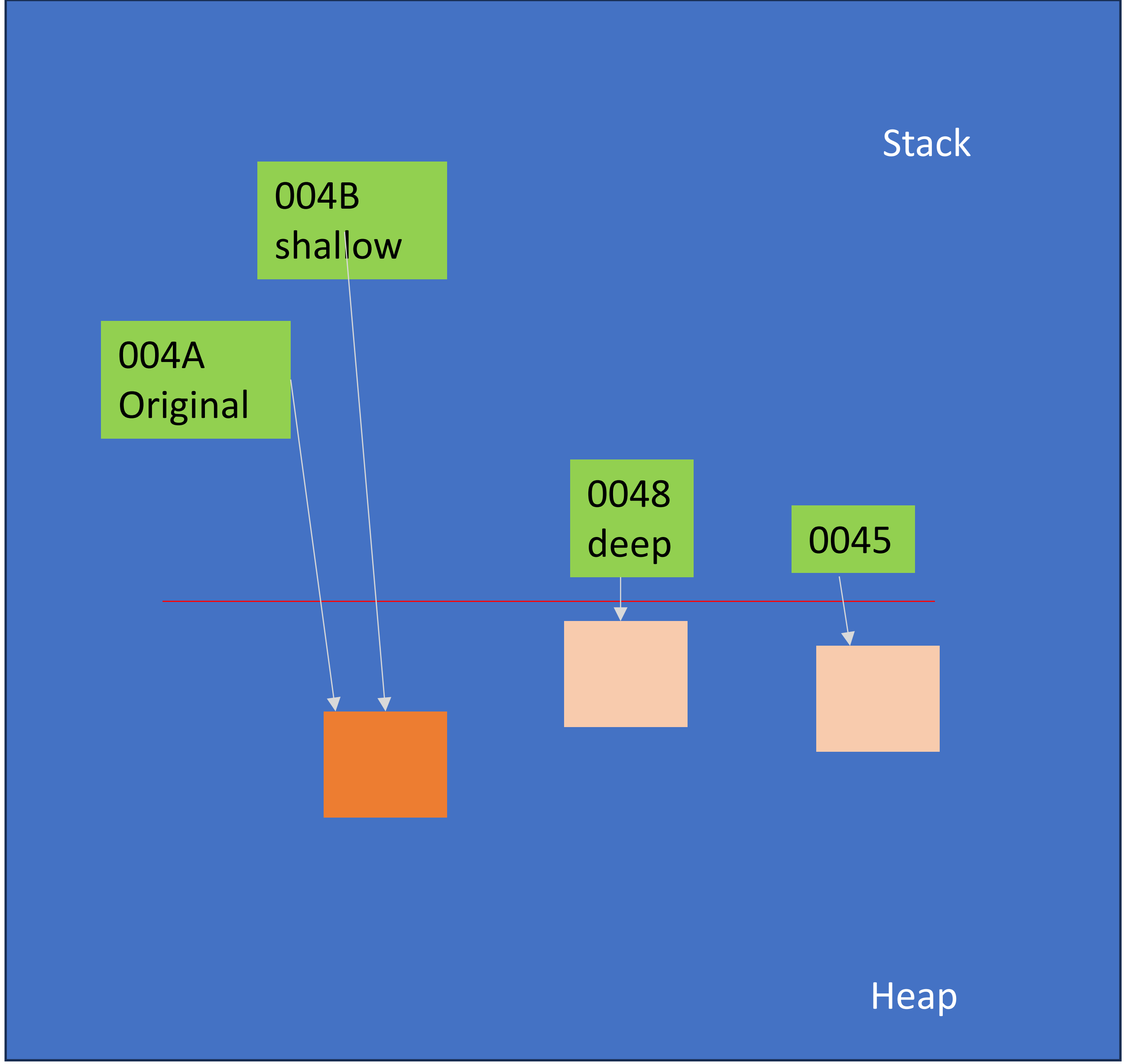
Key		Value
name		Bob
age		30

Dictionary Example-3

```
# Original dictionary with a mutable value
original = {
    "name": "Alice",
    "skills": ["Python", "Machine Learning"]
}
# Create a shallow copy
shallow = original.copy()
# Modify the original dictionary's list
original["skills"].append("Deep Learning-2")
# Check the shallow copy
print("Original:", original)
print("Shallow Copy:", shallow)
```

Output

Original: {'name': 'Alice', 'skills': ['Python', 'Machine Learning', 'Deep Learning-2']}
Shallow Copy: {'name': 'Alice', 'skills': ['Python', 'Machine Learning', 'Deep Learning-2']}


Stack
004B
shallow
004A
Original
0048
deep
0045
Heap

Dictionary Example-3

```
import copy


# Create a deep copy
deep = copy.deepcopy(original)


# Modify the original dictionary's list
original["skills"].append("Data Science")


print("Original:", original)
print("Deep Copy:", deep)
```

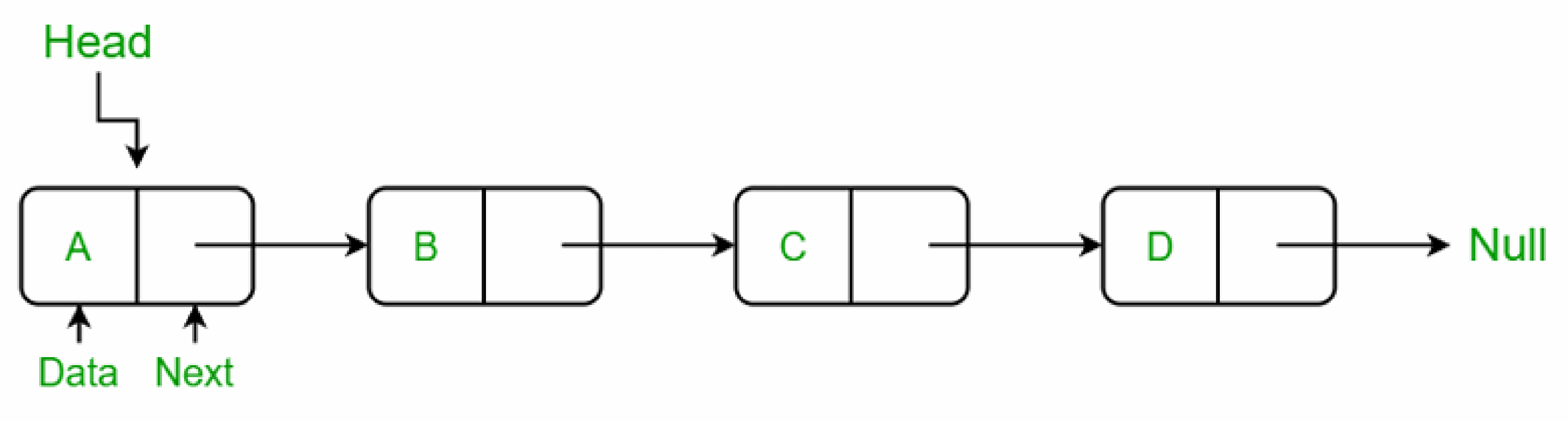
Output

Original: {'name': 'Alice', 'skills': ['Python', 'Machine Learning', 'Deep Learning-2', 'Data Science']}
Deep Copy: {'name': 'Alice', 'skills': ['Python', 'Machine Learning', 'Deep Learning-2']}

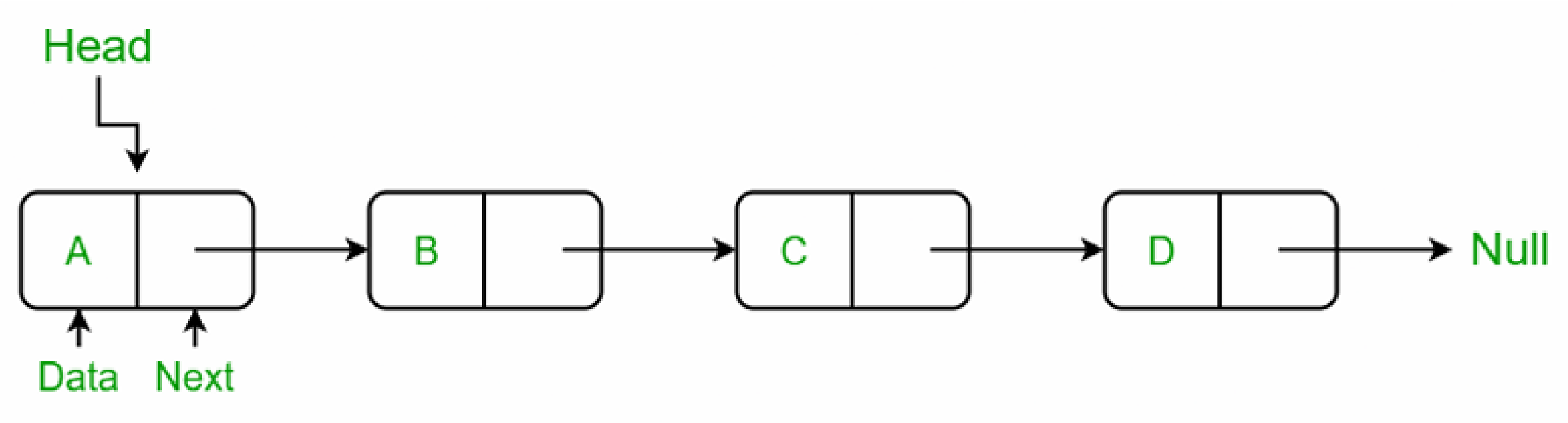
Linked Lists

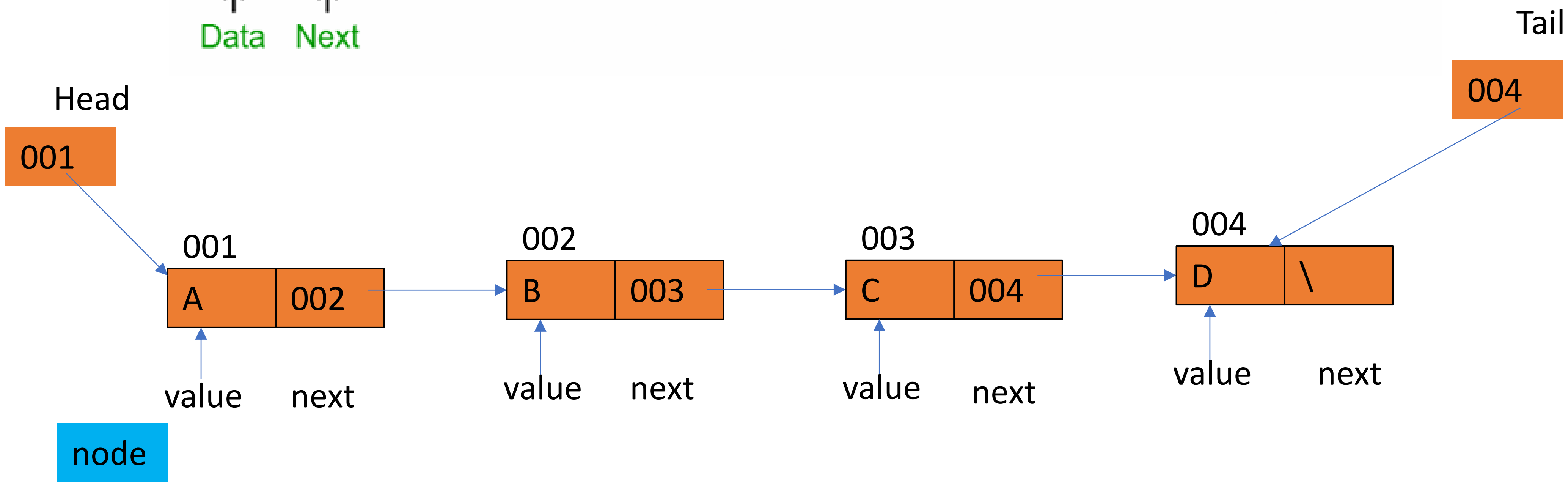
- A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations.
- The elements in a linked list are linked using pointers.
- A linked list is represented by a pointer to the first node of the linked list.
- The first node is called the head.
- If the linked list is empty, then the value of the head is NULL.
- Each node in a list consists of at least two parts:
  - Data
  - Pointer (Or Reference) to the next node

Linked Lists



Linked Lists





Node Example

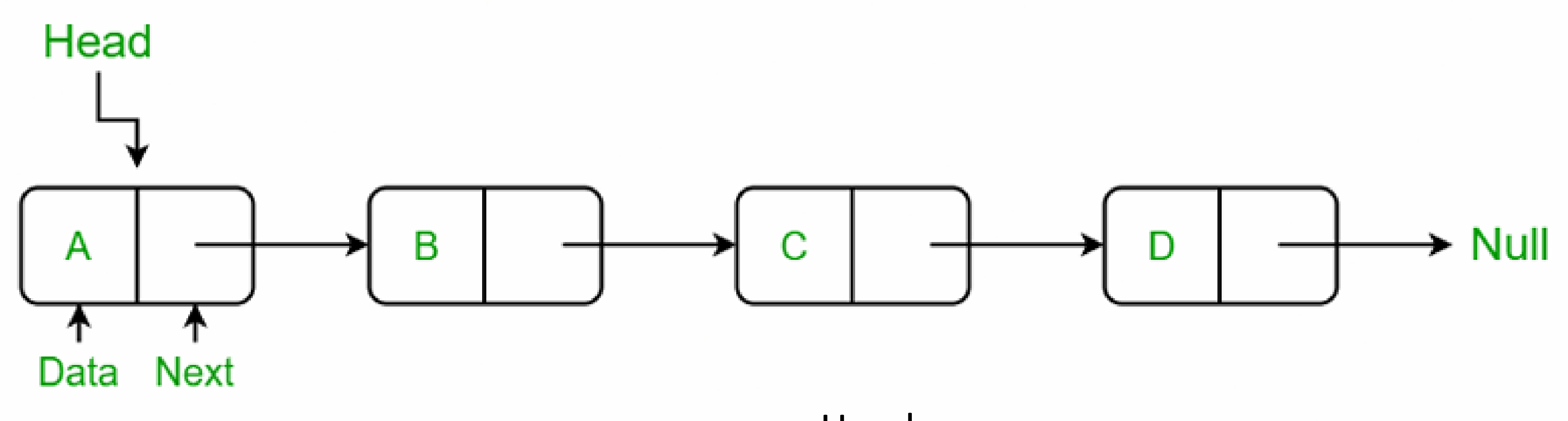
```python
# Creating a node class
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
# Creating a linked list class
class LinkedList:
    def __init__(self):
        self.head = None
# Initializing a linked list
list_ = LinkedList()
# Creating the nodes
list_.head = Node("Python")
second_node = Node("Tutorial")
third_node = Node("Data Structures")
```

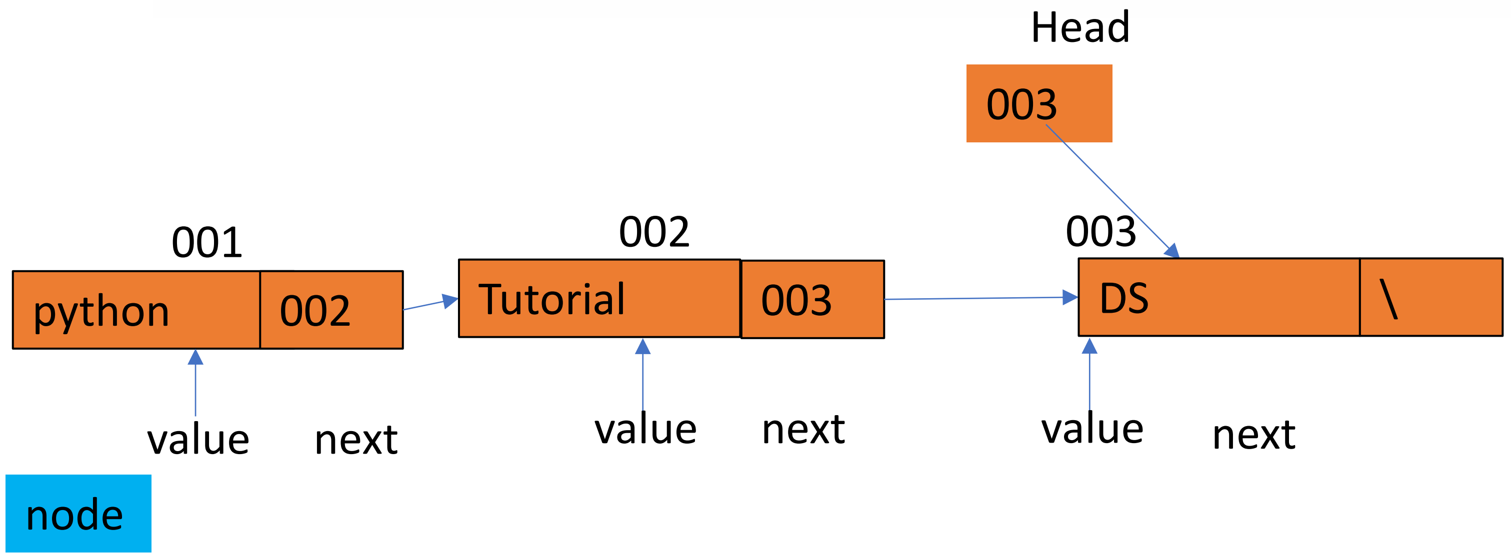
```python
# Connecting the nodes
list_.head.next = second_node
second_node.next = third_node
# Printing the linked list
while list_.head != None:
    print(list_.head.value, end = "\n")
    list_.head = list_.head.next
```

Output

```
Python
Tutorial
Data Structures
```

Linked Lists



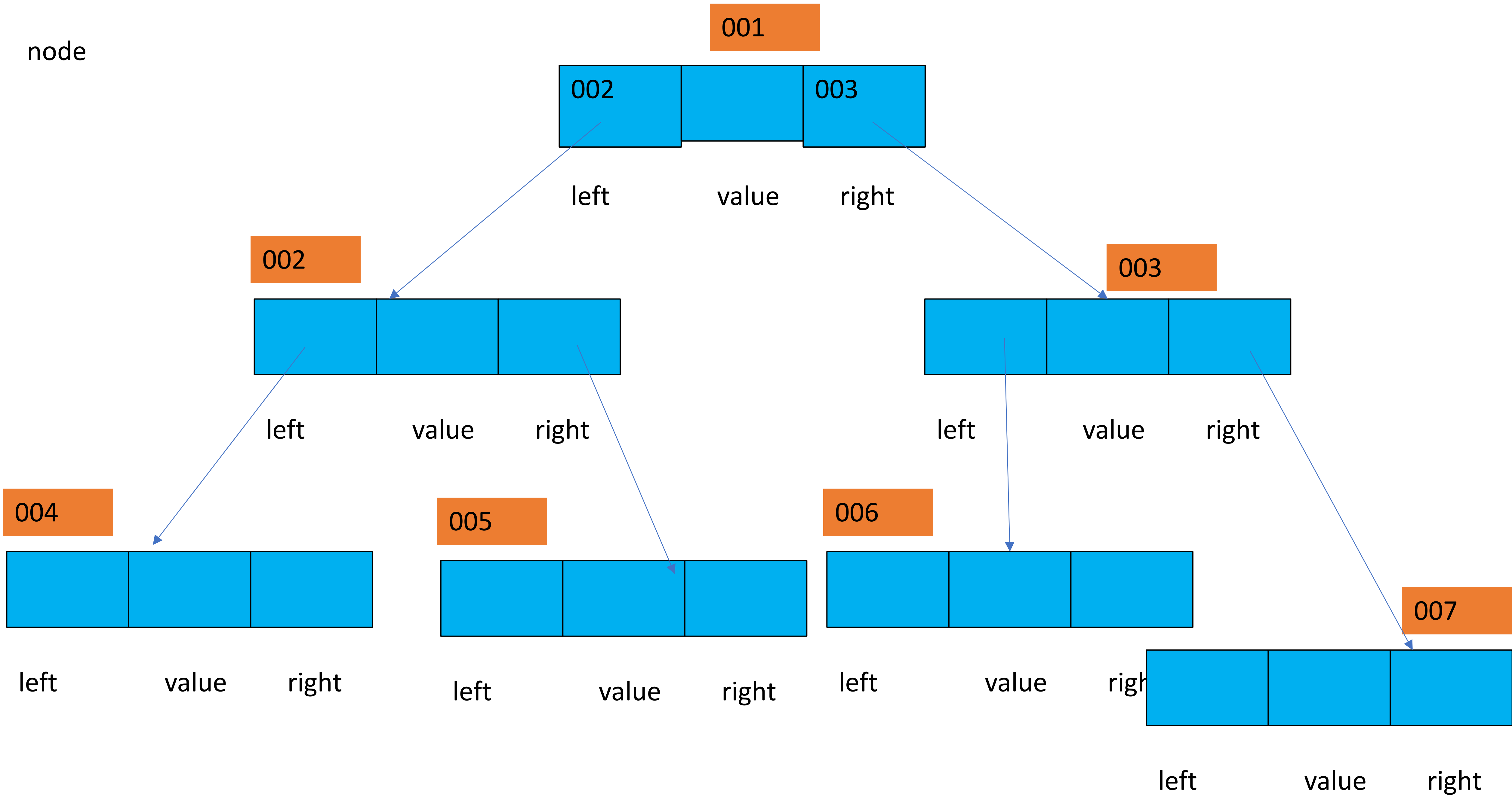


Binary Tree

- A tree is a hierarchical data structure.
- A binary tree is a tree whose elements can have almost two children.
- A Binary Tree node contains the following parts.
  - Data
  - Pointer to left child
  - Pointer to the right child

 # A Python class that represents an individual node in a Binary Tree class Node:

```
def __init__(self,key):
      self.left = None
      self.right = None
      self.val = key
```


node
001
002
003
left
value
right
002
003
left
value
right
left
value
right
004
005
006
007
left
value
right
left
value
right
left
value
righ
left
value
right

Binary Tree Example-1

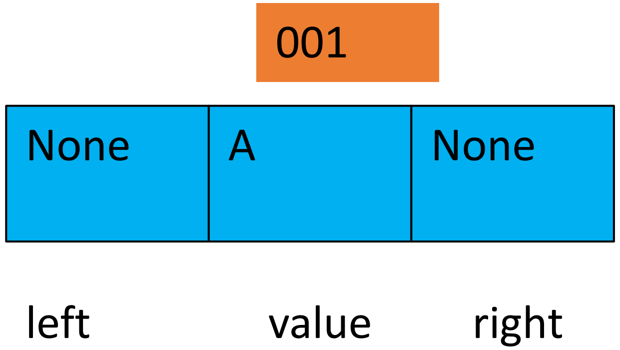
```
# Python program to introduce Binary Tree
# # A class that represents an individual node in a Binary Tree
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None  # Pointer to the left child
        self.right = None  # Pointer to the right child

class BinaryTree:
    def __init__(self):
        self.root = None  # Root of the binary tree

    # Pre-order traversal: Root -> Left -> Right
    def pre_order(self, node):
        if node:
            print(node.value, end=" ")  # Visit root
            self.pre_order(node.left)  # Traverse left
            self.pre_order(node.right)  # Traverse right
```

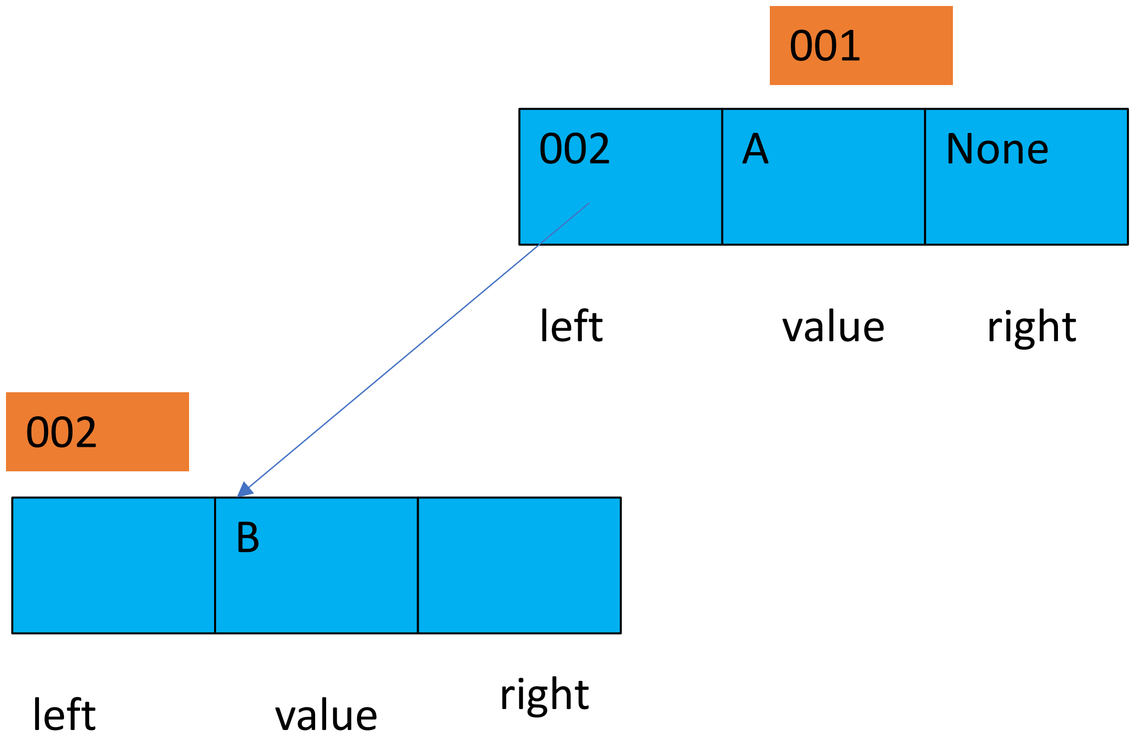
```
tree = BinaryTree()
```

node



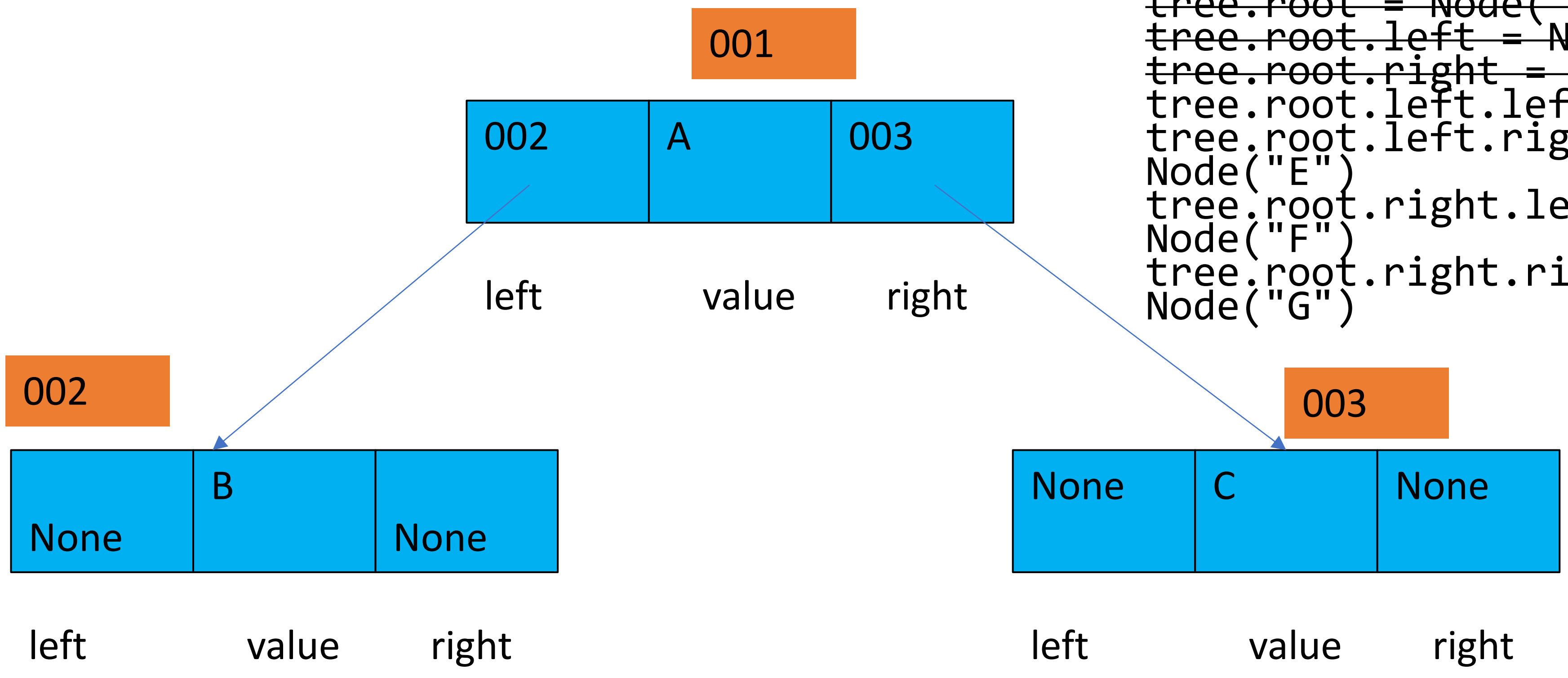
~~tree.root = Node("A")~~
tree.root.left = Node("B")
tree.root.right = Node("C")
tree.root.left.left = Node("D")
tree.root.left.right = Node("E")
tree.root.right.left = Node("F")
tree.root.right.right = Node("G")

node



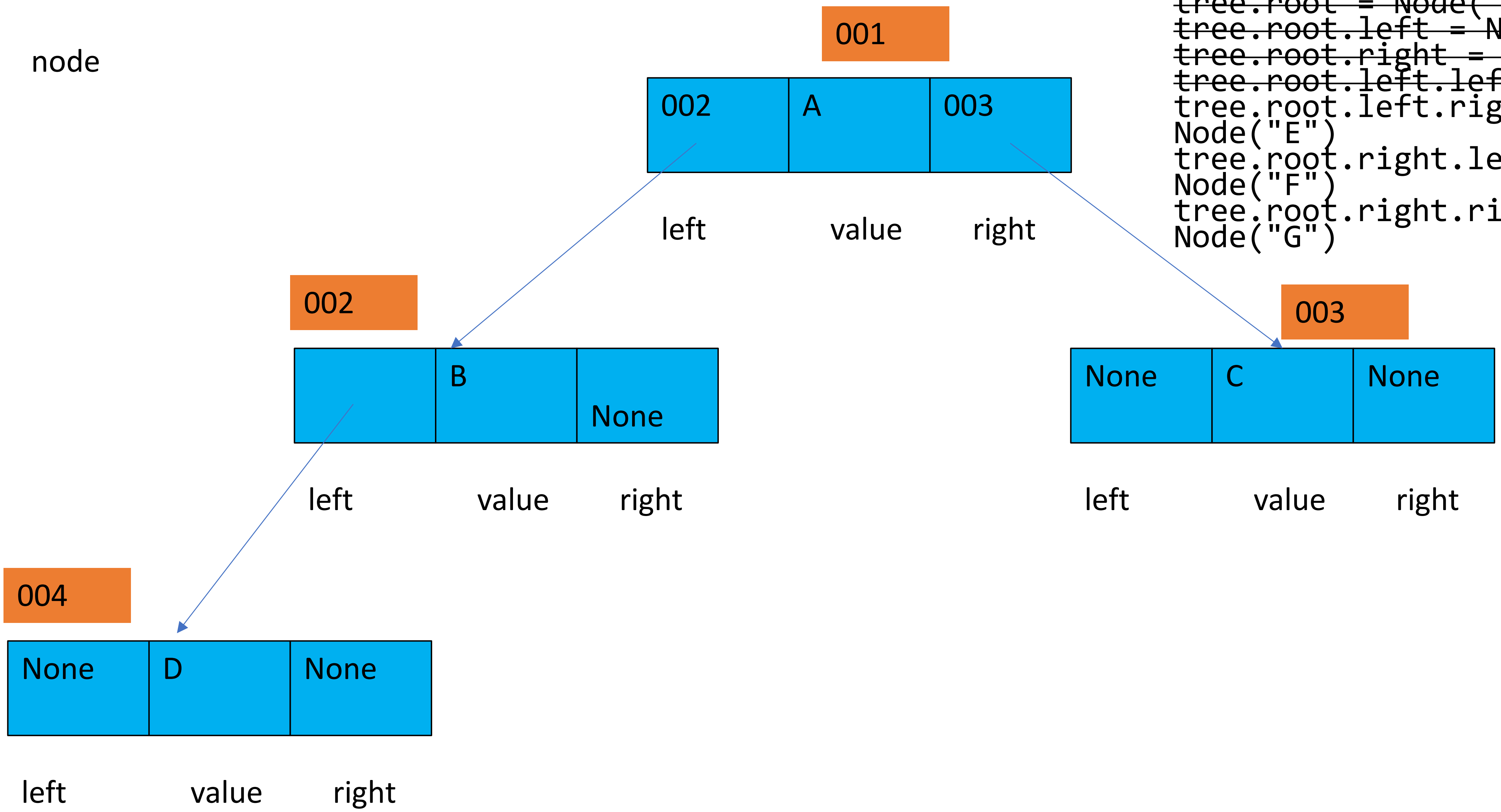
~~tree.root = Node("A")~~
~~tree.root.left = Node("B")~~
tree.root.right = Node("C")
tree.root.left.left = Node("D")
tree.root.left.right =
Node("E")
tree.root.right.left =
Node("F")
tree.root.right.right =
Node("G")

node



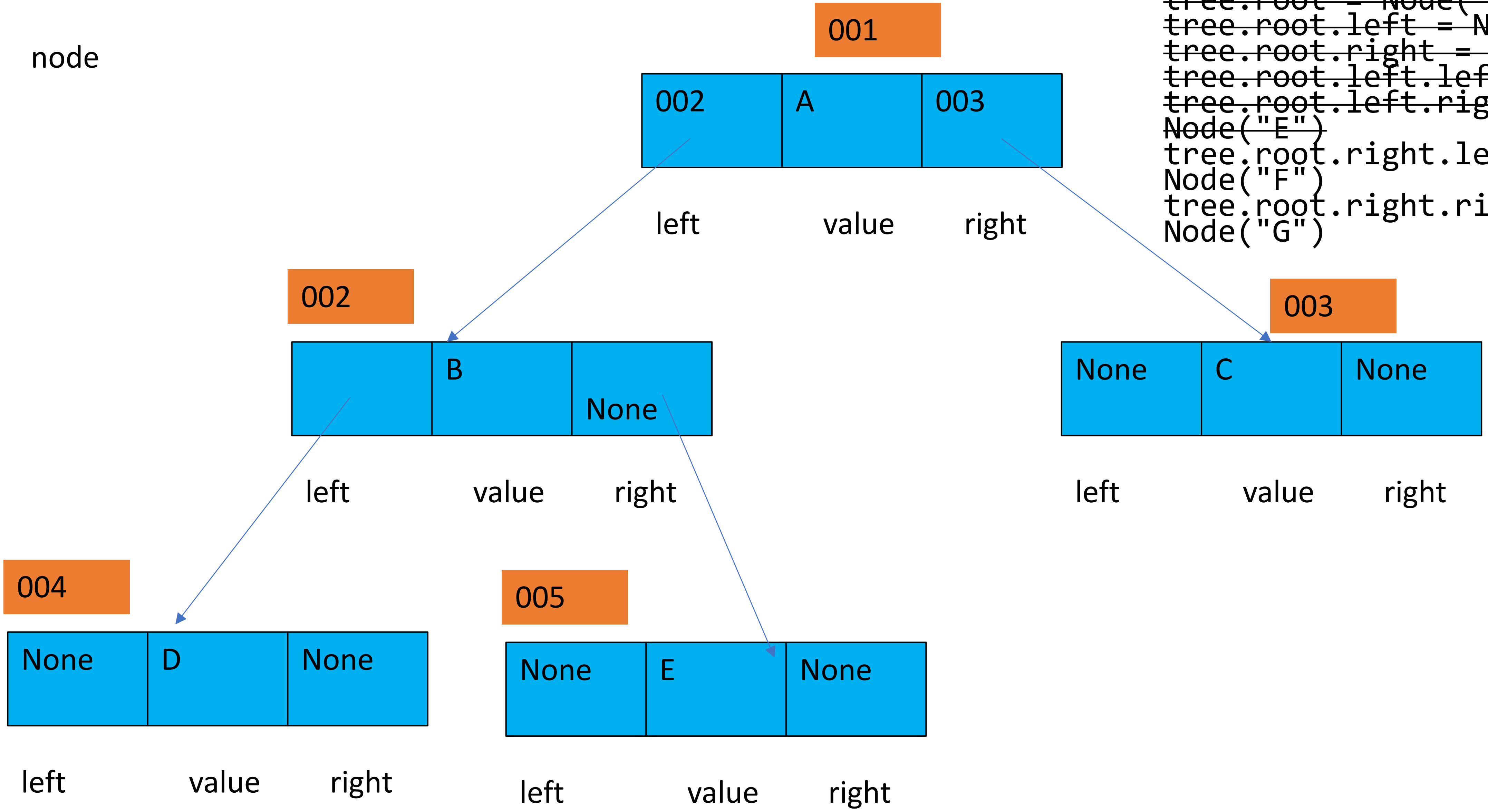
~~tree.root = Node("A")~~
~~tree.root.left = Node("B")~~
~~tree.root.right = Node("C")~~
tree.root.left.left = Node("D")
tree.root.left.right = Node("E")
tree.root.right.left = Node("F")
tree.root.right.right = Node("G")

node



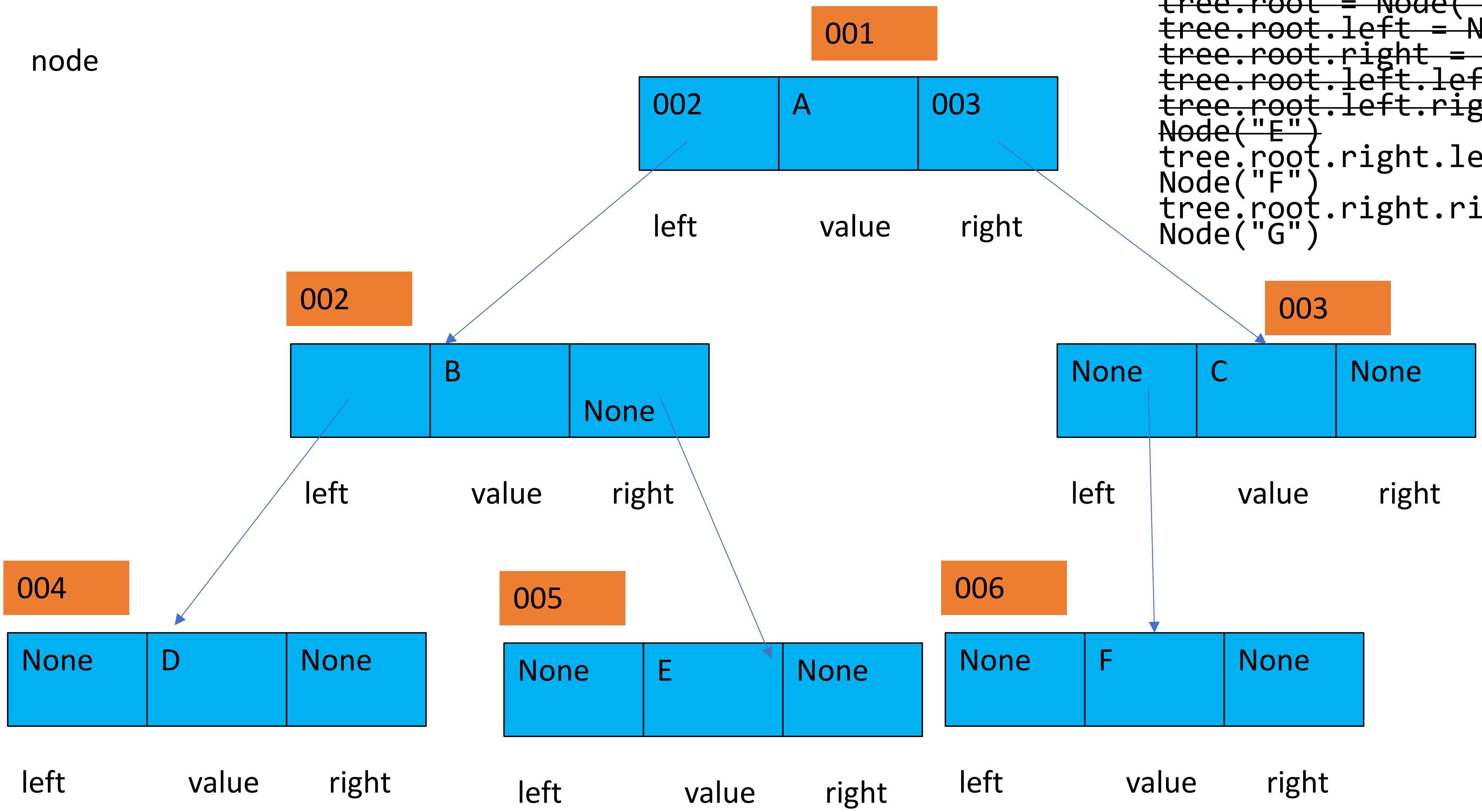
~~tree.root = Node("A")~~
~~tree.root.left = Node("B")~~
~~tree.root.right = Node("C")~~
~~tree.root.left.left = Node("D")~~
tree.root.left.right = Node("E")
tree.root.right.left = Node("F")
tree.root.right.right = Node("G")

node



~~tree.root = Node("A")~~
~~tree.root.left = Node("B")~~
~~tree.root.right = Node("C")~~
~~tree.root.left.left = Node("D")~~
~~tree.root.left.right = Node("E")~~
tree.root.right.left = Node("F")
tree.root.right.right = Node("G")

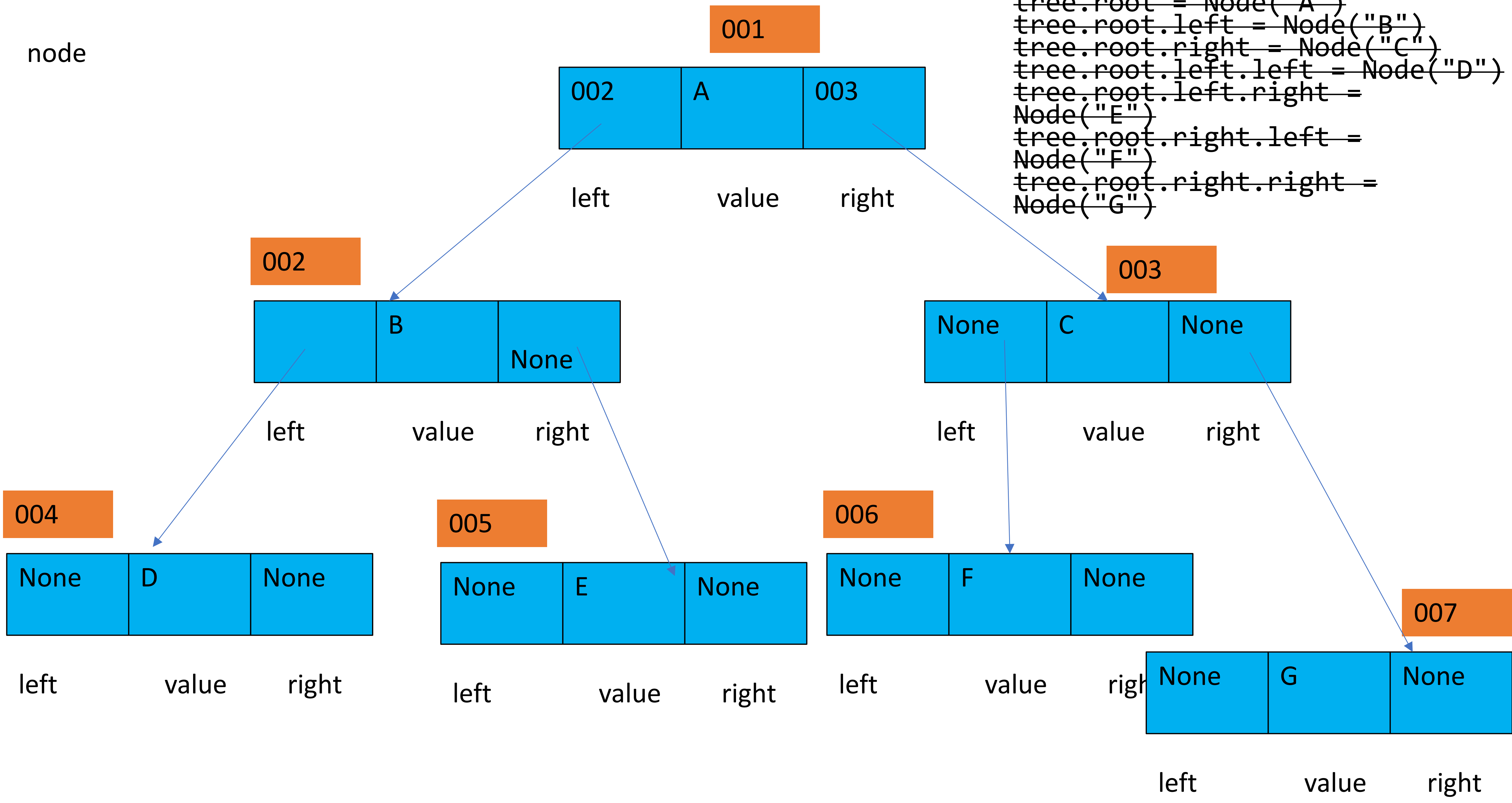
node



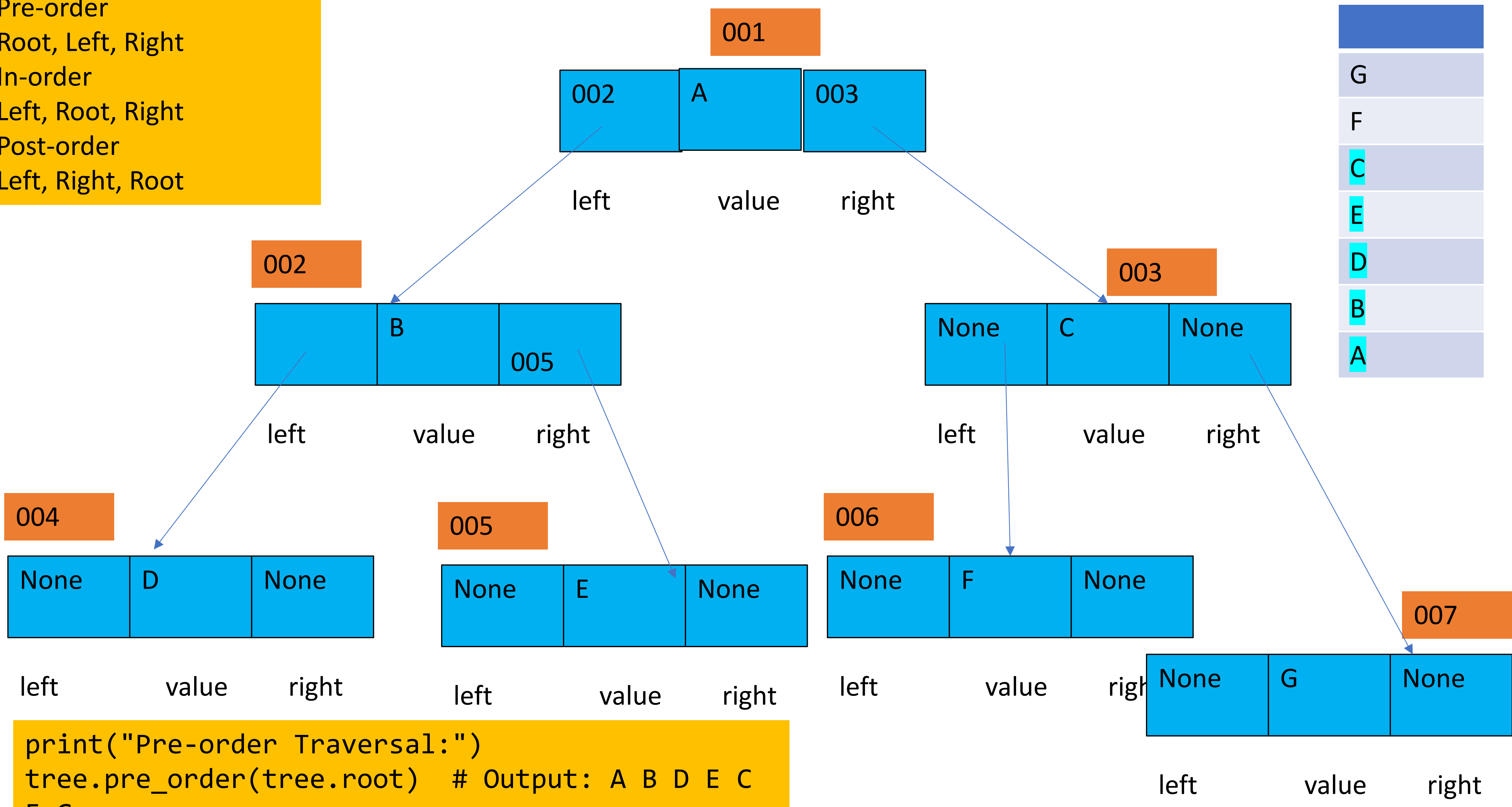
~~tree.root = Node("A")~~
~~tree.root.left = Node("B")~~
~~tree.root.right = Node("C")~~
~~tree.root.left.left = Node("D")~~
~~tree.root.left.right = Node("E")~~
tree.root.right.left = Node("F")
tree.root.right.right = Node("G")

node

~~tree.root = Node("A")~~
~~tree.root.left = Node("B")~~
~~tree.root.right = Node("C")~~
~~tree.root.left.left = Node("D")~~
~~tree.root.left.right = Node("E")~~
~~tree.root.right.left = Node("F")~~
~~tree.root.right.right = Node("G")~~

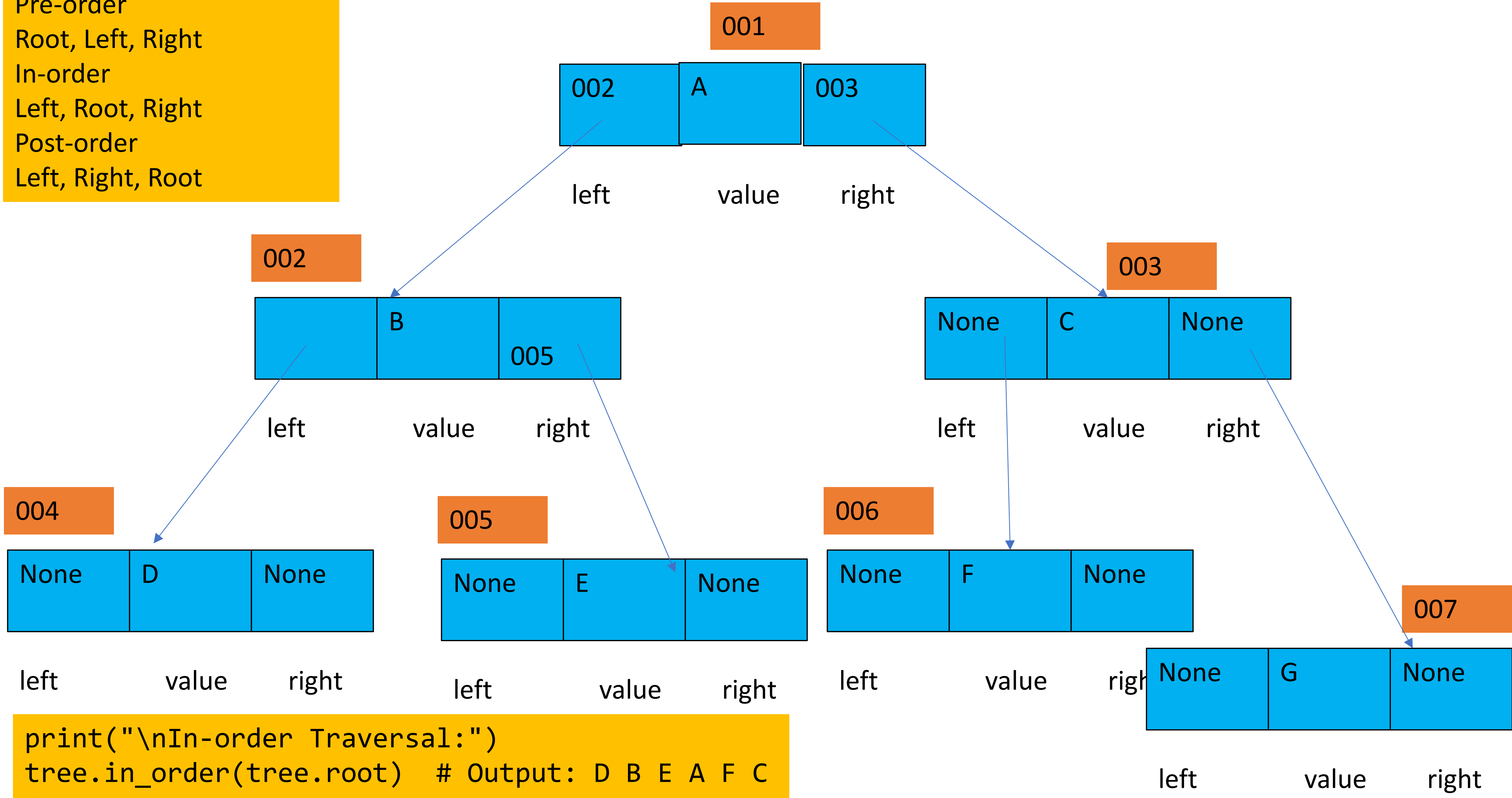


Pre-order
Root, Left, Right
In-order
Left, Root, Right
Post-order
Left, Right, Root



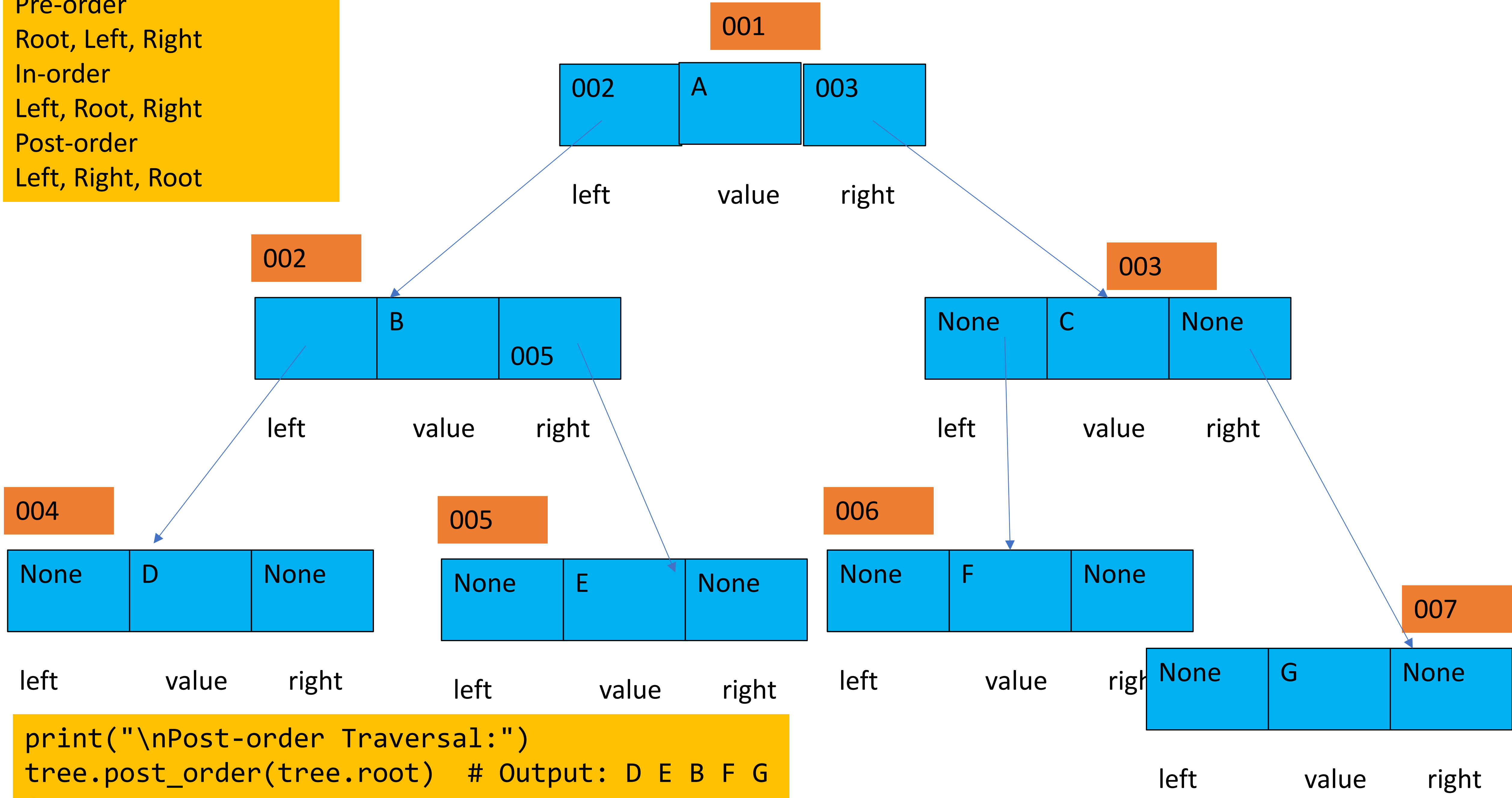

```
print("Pre-order Traversal:")
tree.pre_order(tree.root)  # Output: A B D E C
F G
```

Pre-order
Root, Left, Right
In-order
Left, Root, Right
Post-order
Left, Right, Root




```
print("\nIn-order Traversal:")
tree.in_order(tree.root)  # Output: D B E A F C
G
```

Pre-order
Root, Left, Right
In-order
Left, Root, Right
Post-order
Left, Right, Root




```
print("\nPost-order Traversal:")
tree.post_order(tree.root)  # Output: D E B F G
C A
```

Binary Tree Example-1

```
# In-order traversal: Left -> Root -> Right
def in_order(self, node):
    if node:
        self.in_order(node.left)  # Traverse left
        print(node.value, end=" ")  # Visit root
        self.in_order(node.right)  # Traverse right

# Post-order traversal: Left -> Right -> Root
def post_order(self, node):
    if node:
        self.post_order(node.left)  # Traverse left
        self.post_order(node.right)  # Traverse right
        print(node.value, end=" ")  # Visit root
```

Binary Tree Example-1

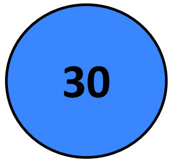
```
# Create the binary tree
tree = BinaryTree()

# Manually create nodes and link them
tree.root = Node("A")
tree.root.left = Node("B")
tree.root.right = Node("C")
tree.root.left.left = Node("D")
tree.root.left.right = Node("E")
tree.root.right.left = Node("F")
tree.root.right.right = Node("G")

# Perform tree traversals
print("Pre-order Traversal:")
tree.pre_order(tree.root)  # Output: A B D E C F G
print("\nIn-order Traversal:")
tree.in_order(tree.root)  # Output: D B E A F C G
print("\nPost-order Traversal:")
tree.post_order(tree.root)  # Output: D E B F G C A
```

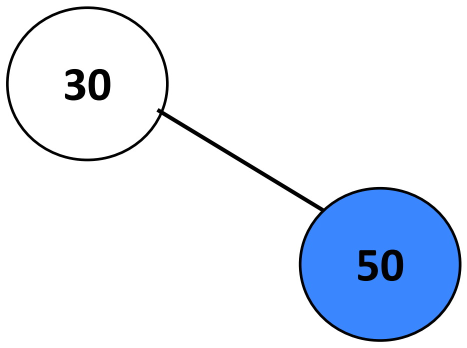
Binary Tree

30, 50, 20, 56, 40, 22,14, 6,76



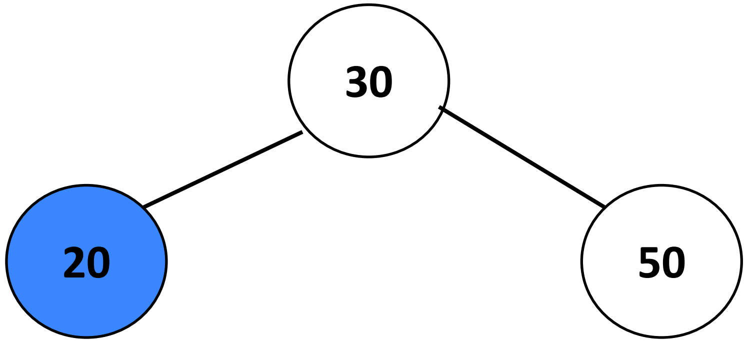
Binary Tree

30, 50, 20, 56, 40, 22,14, 6,76



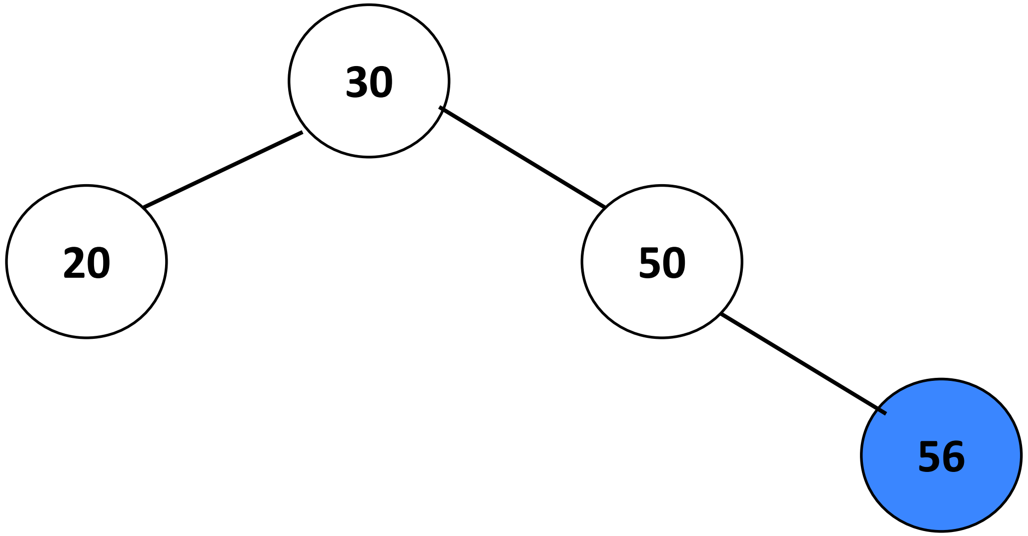
Binary Tree

30, 50, 20, 56, 40, 22,14, 6,76



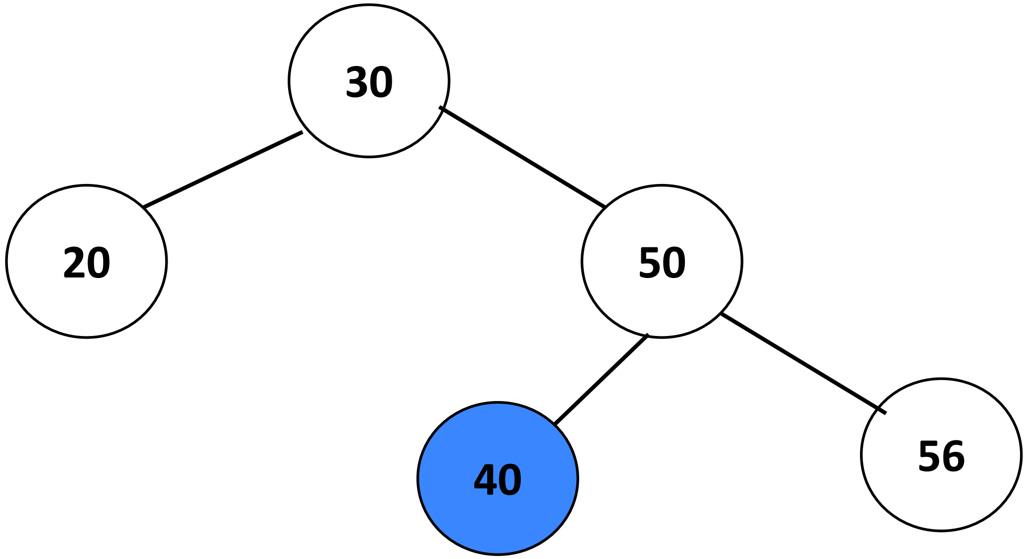
Binary Tree

30, 50, 20, 56, 40, 22,14, 6,76



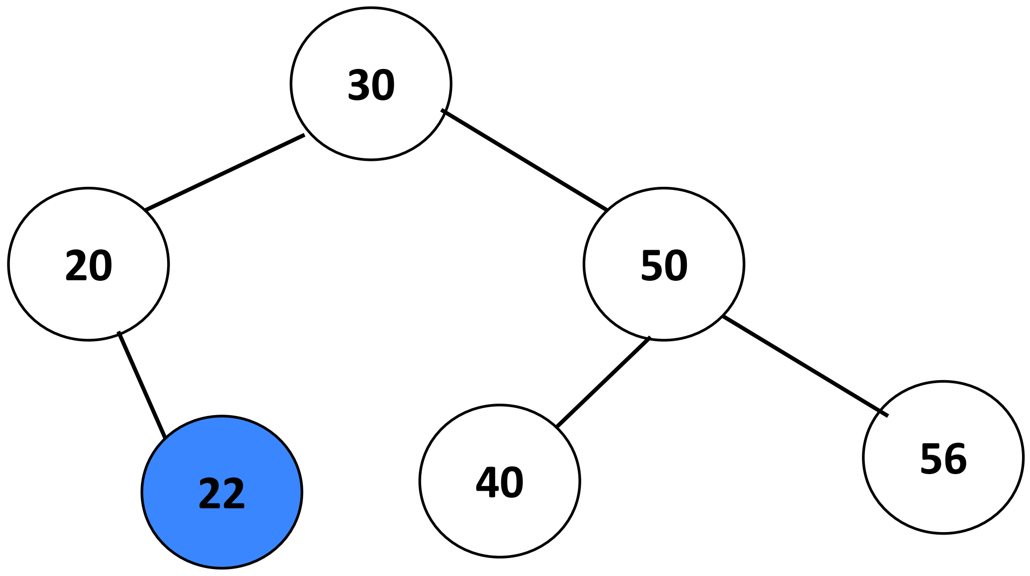
Binary Tree

30, 50, 20, 56, 40, 22,14, 6,76



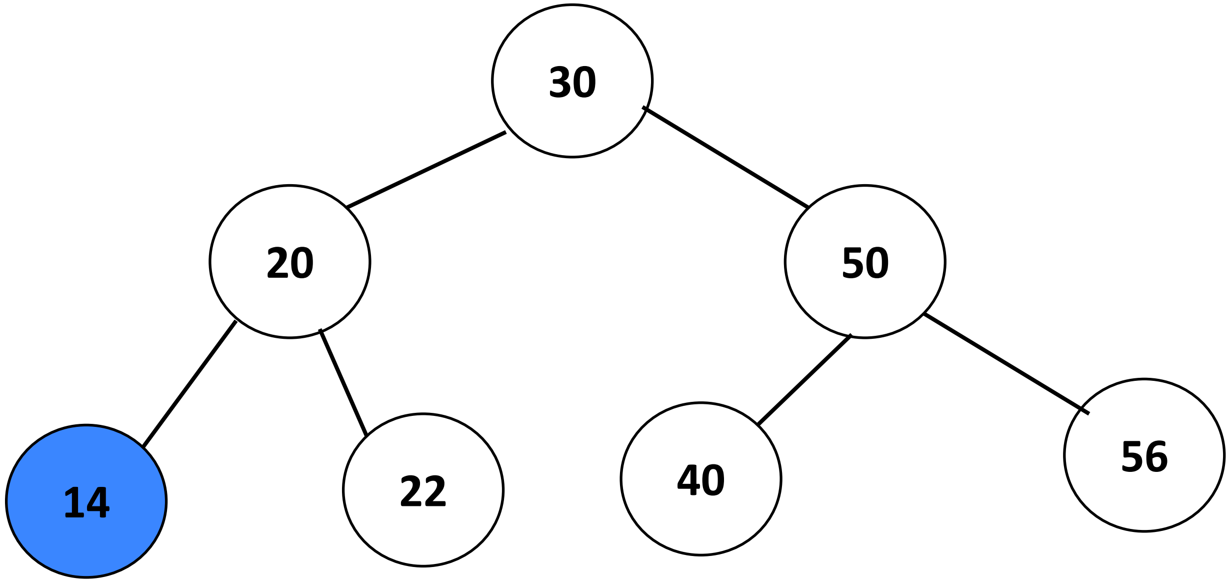
Binary Tree

30, 50, 20, 56, 40, 22,14, 6,76



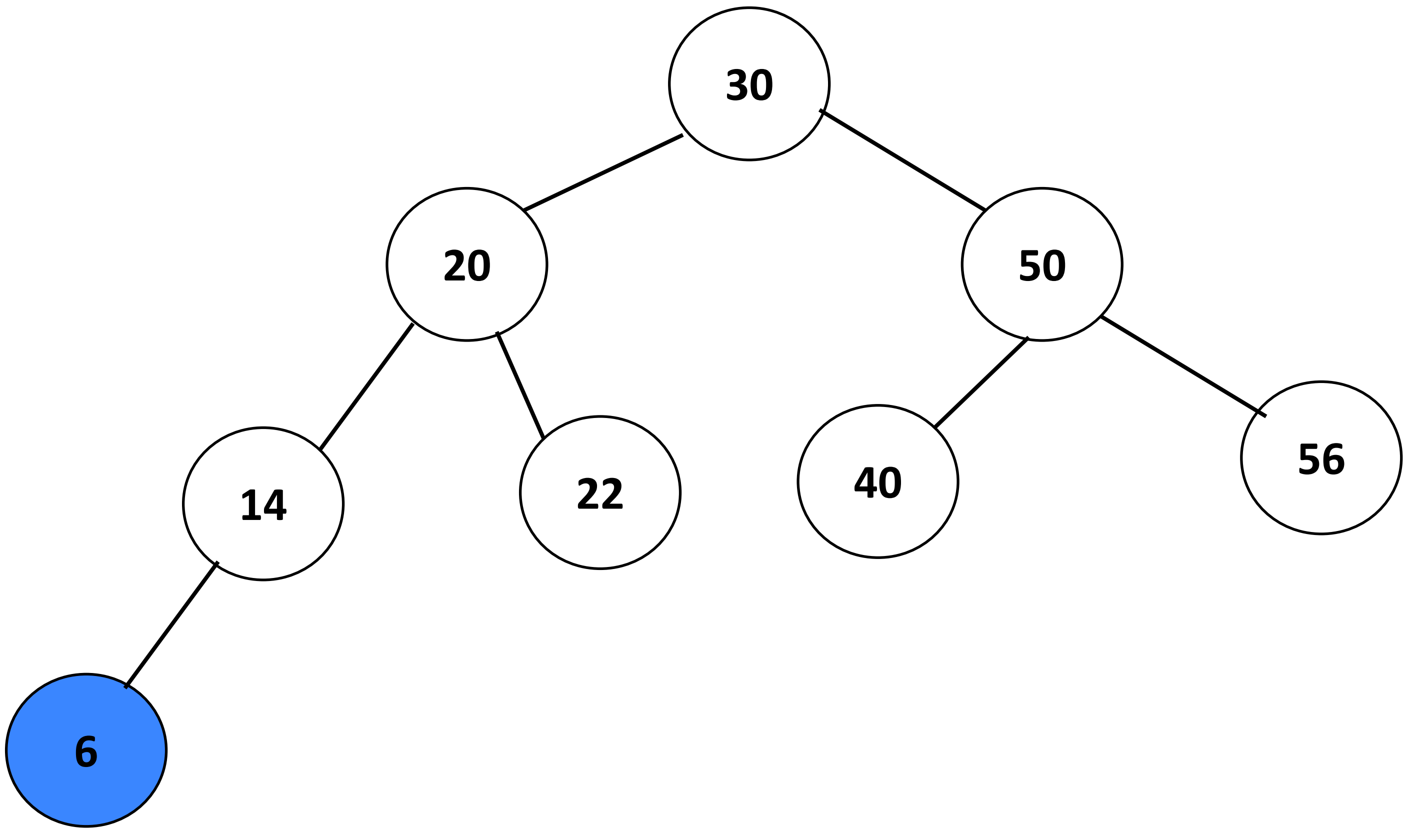
Binary Tree

30, 50, 20, 56, 40, 22,14, 6,76



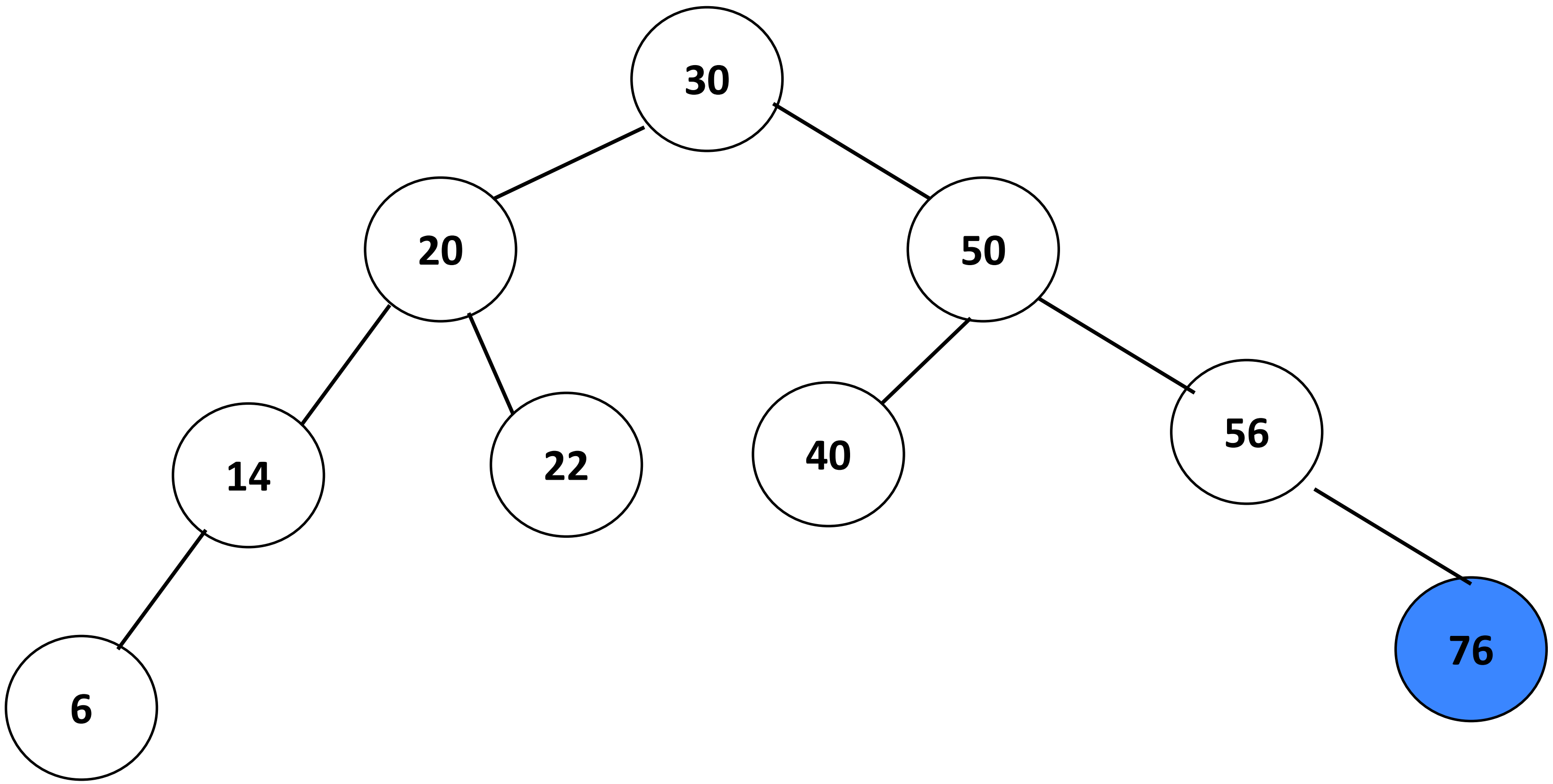
Binary Tree

30, 50, 20, 56, 40, 22,14, 6,76



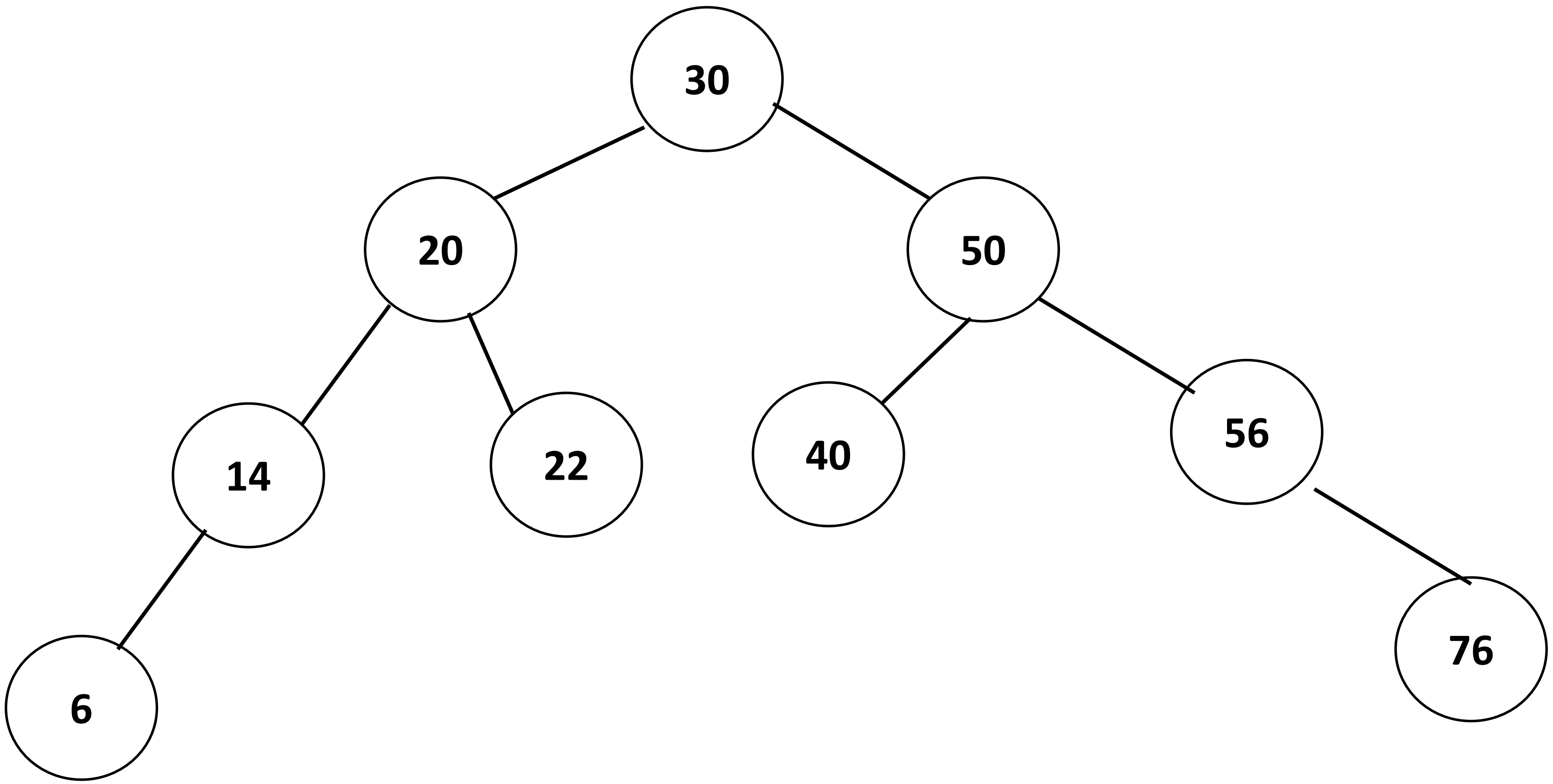
Binary Tree

30, 50, 20, 56, 40, 22,14, 6,76



Binary Tree

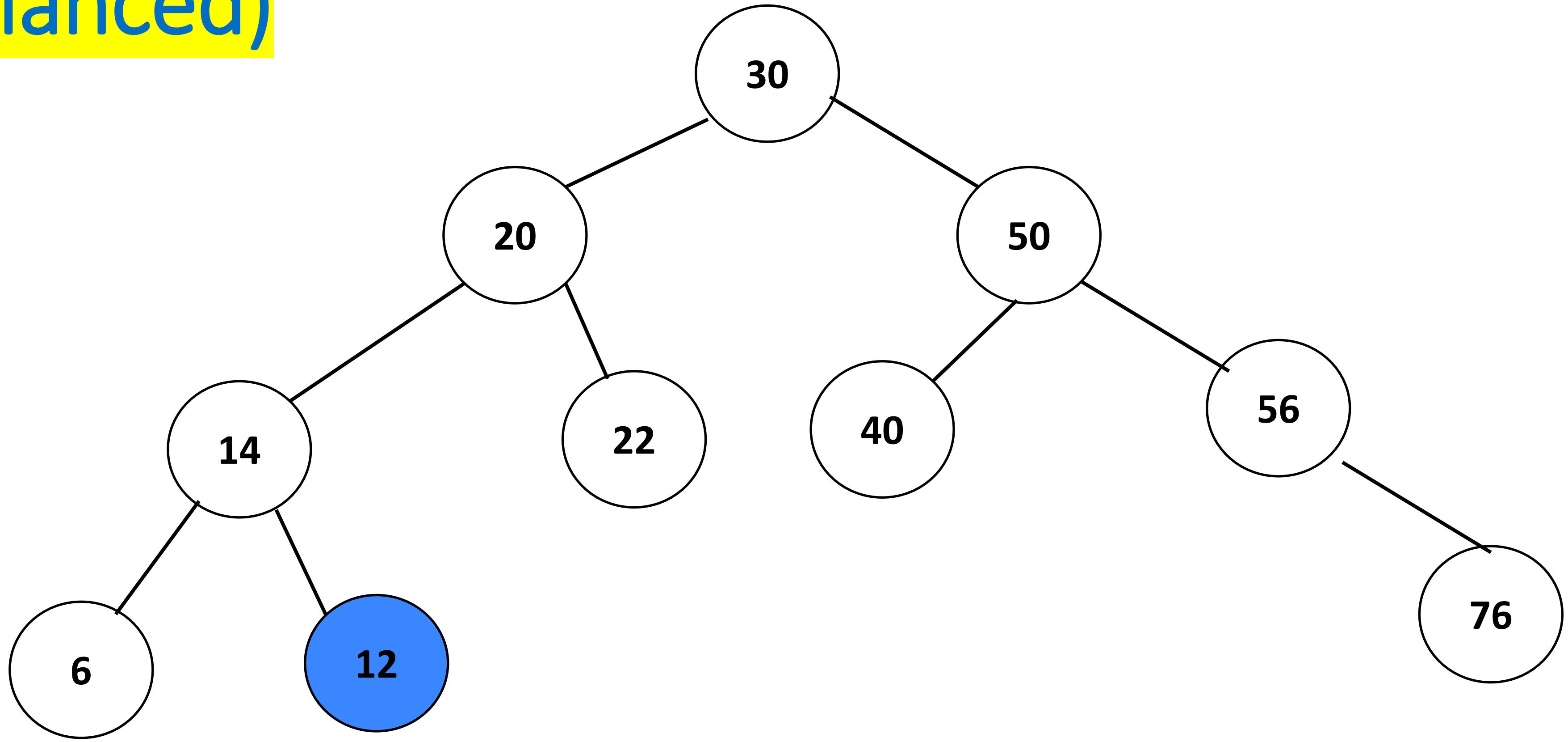
30, 50, 20, 56, 40, 22,14, 6,76



Pre-order Traversal:
30 20 14 6 22 50 40 56 76
In-order Traversal:
6 14 20 22 30 40 50 56 76
Post-order Traversal:
6 14 22 20 40 76 56 50 30

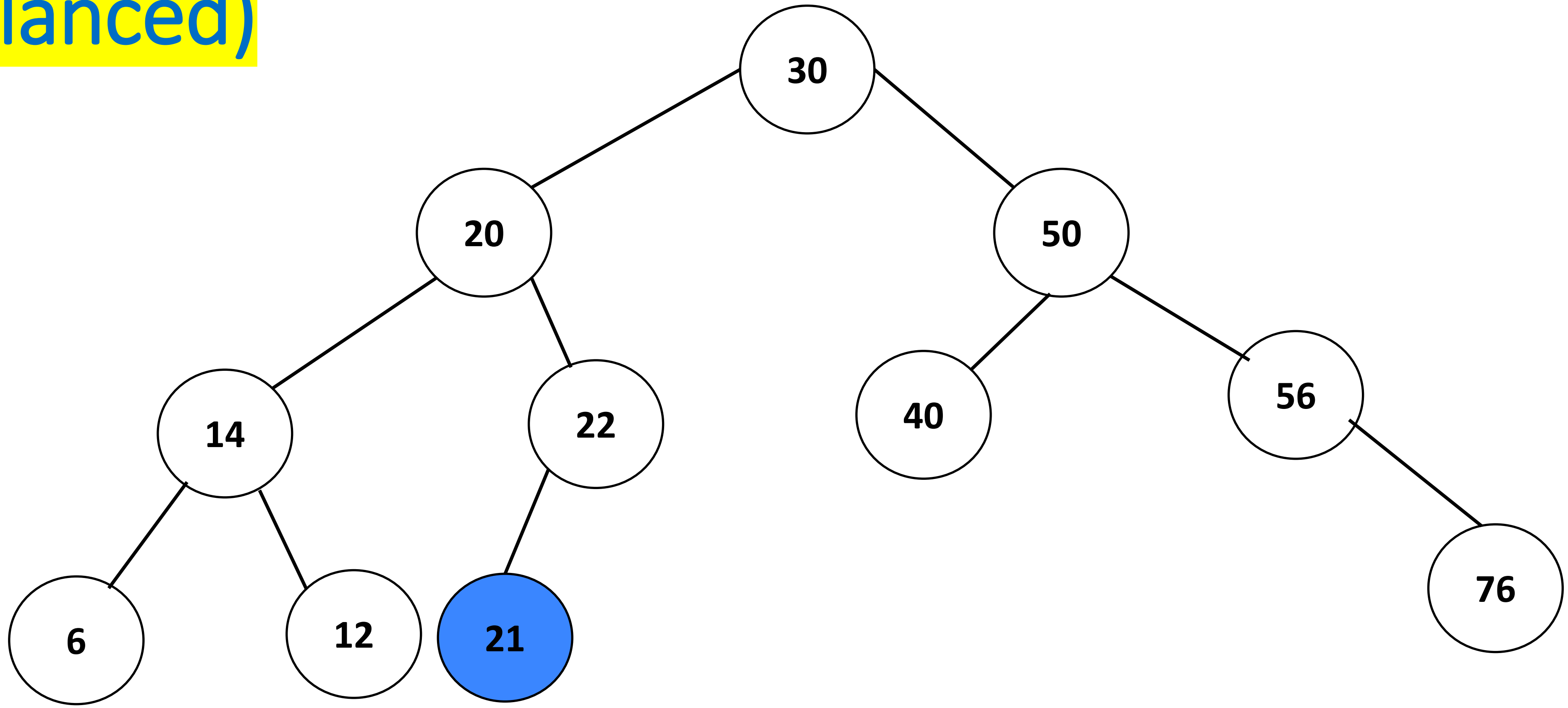
Binary Tree (Balanced)

30, 50, 20, 56, 40, 22,14, 6,76,12,21,25,32,45,53



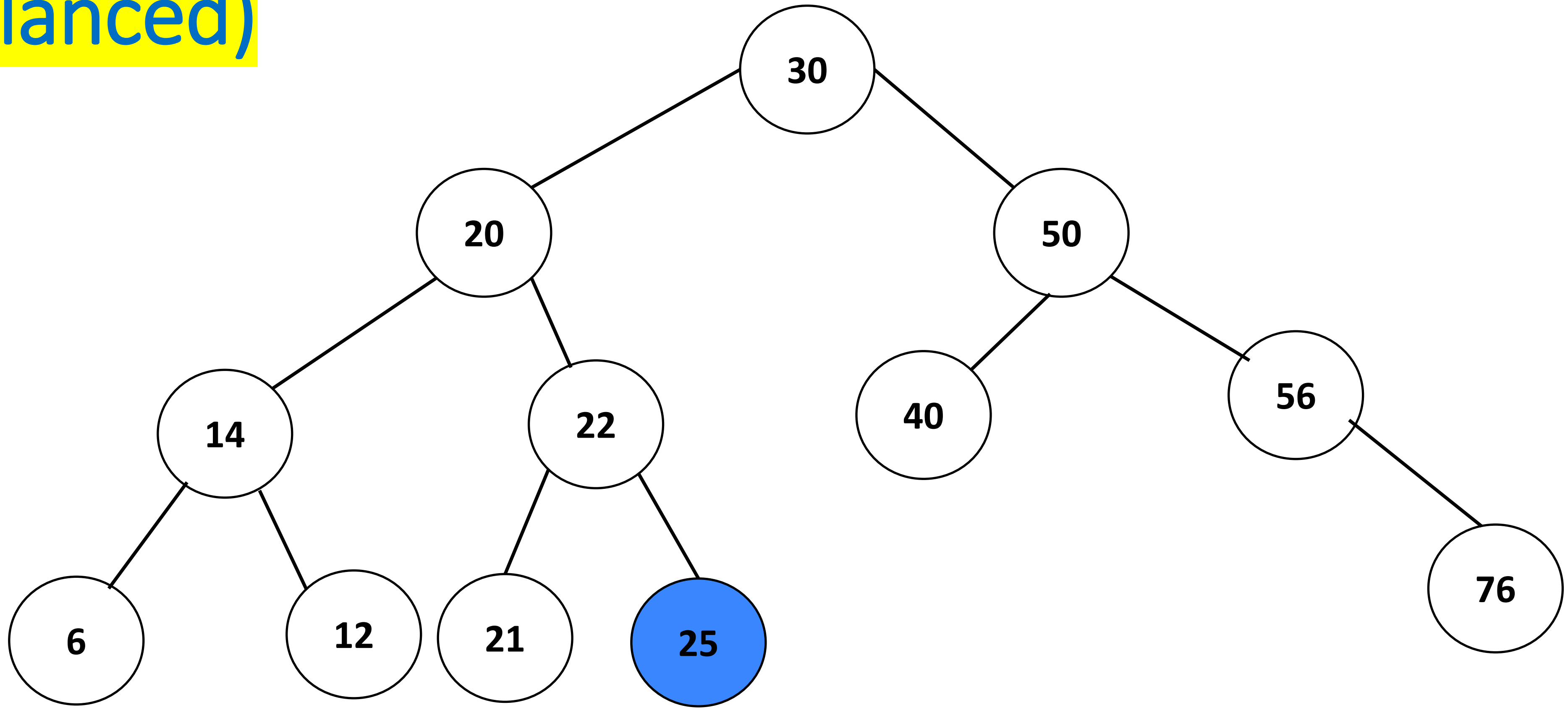
Binary Tree (Balanced)

30, 50, 20, 56, 40, 22,14, 6,76,12,21,25,32,45,53



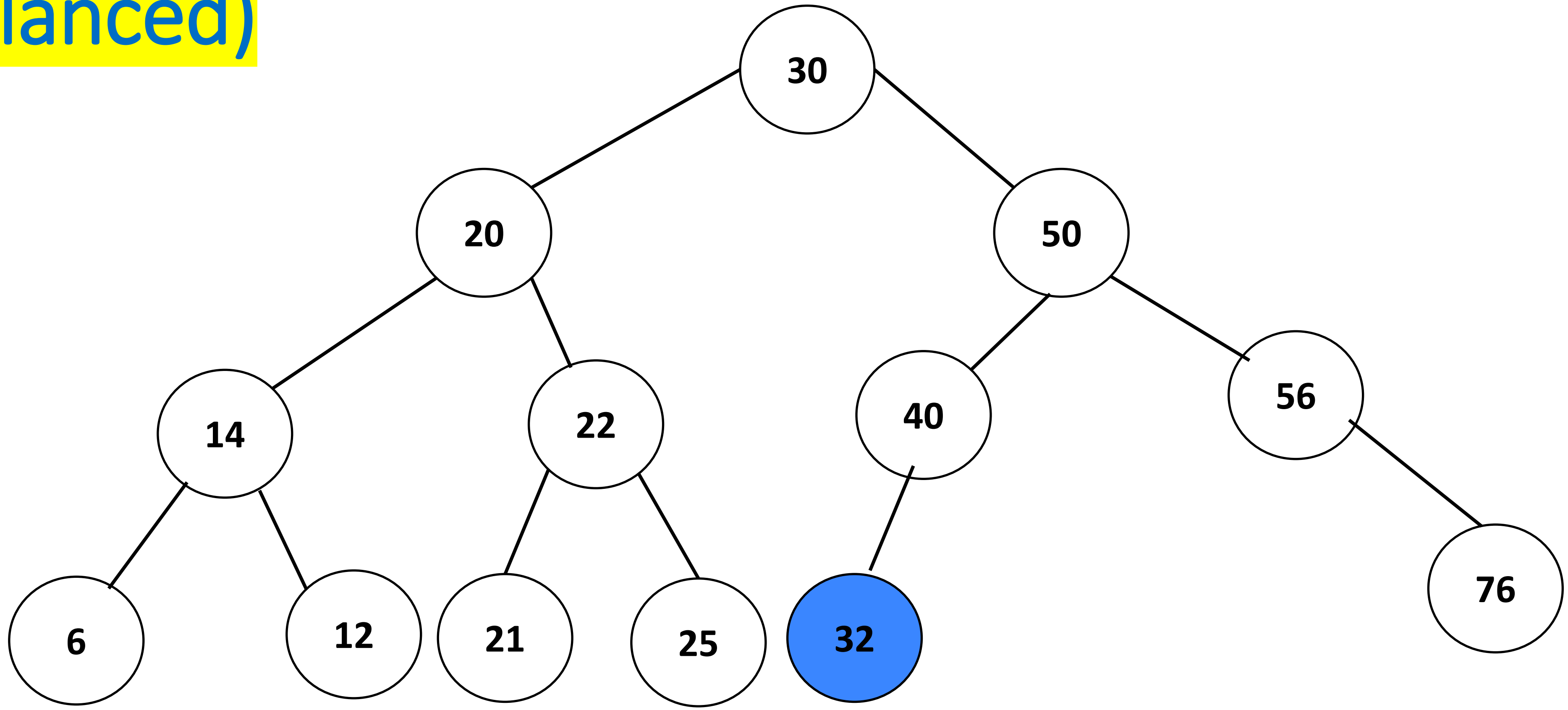
Binary Tree (Balanced)

30, 50, 20, 56, 40, 22,14, 6,76,12,21,25,32,45,53



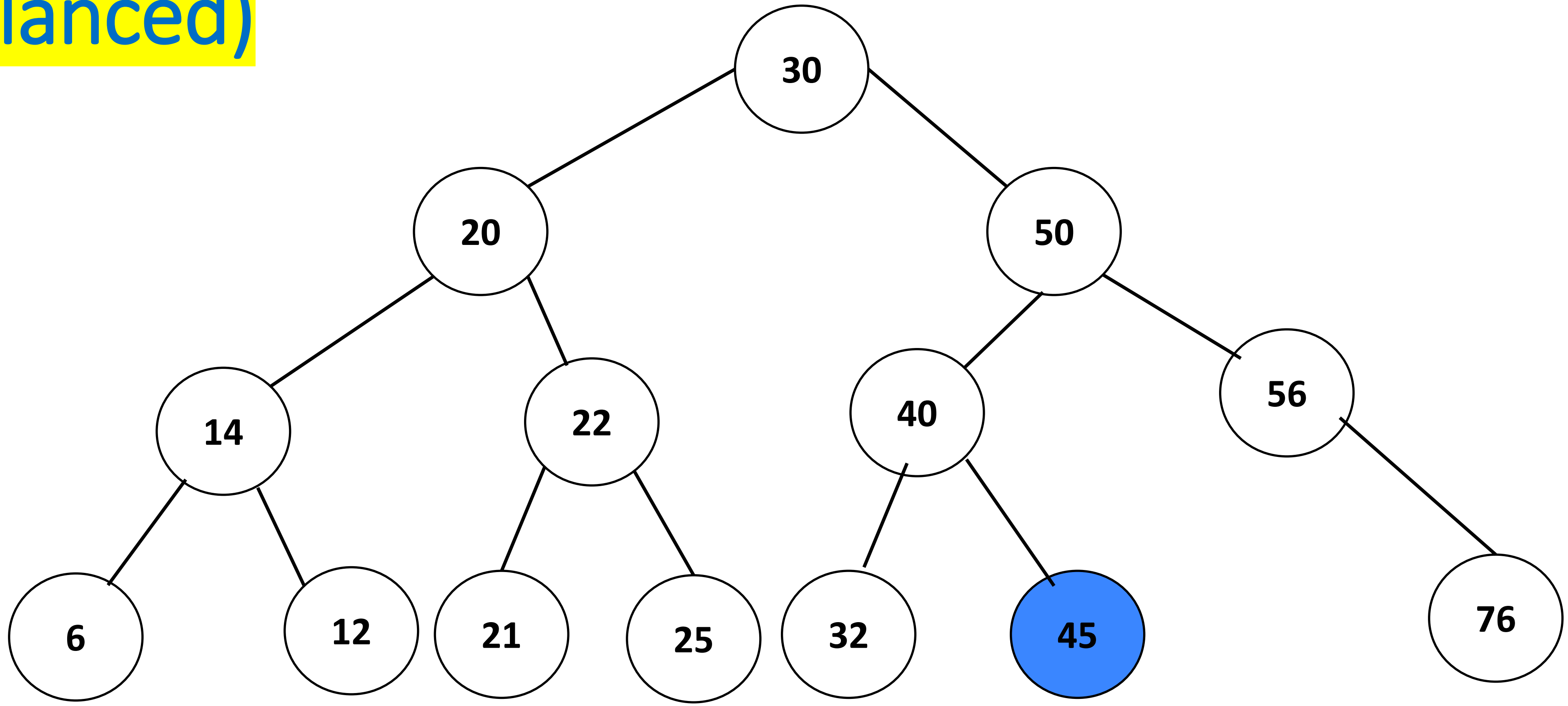
Binary Tree (Balanced)

30, 50, 20, 56, 40, 22,14, 6,76,12,21,25,32,45,53



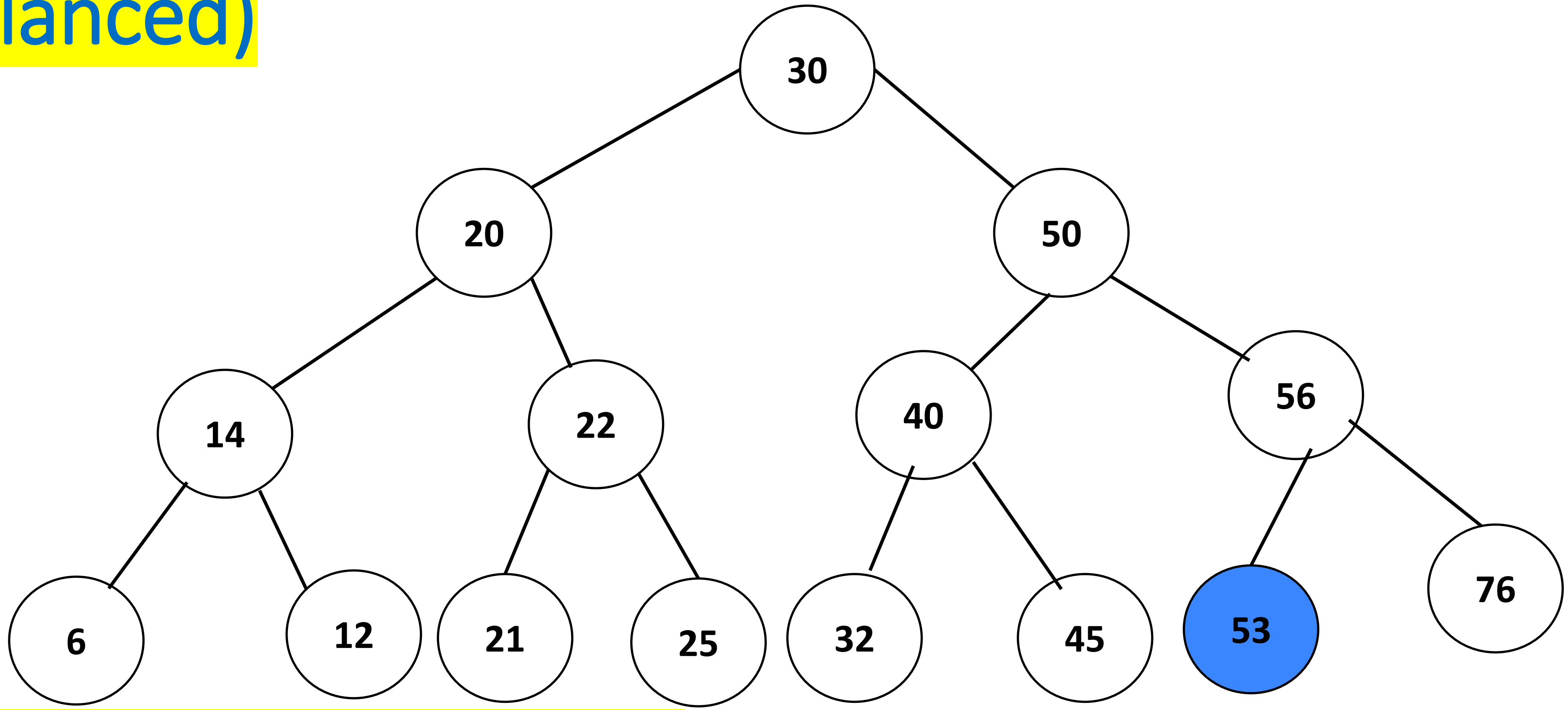
Binary Tree (Balanced)

30, 50, 20, 56, 40, 22,14, 6,76,12,21,25,32,45,53



Binary Tree (Balanced)

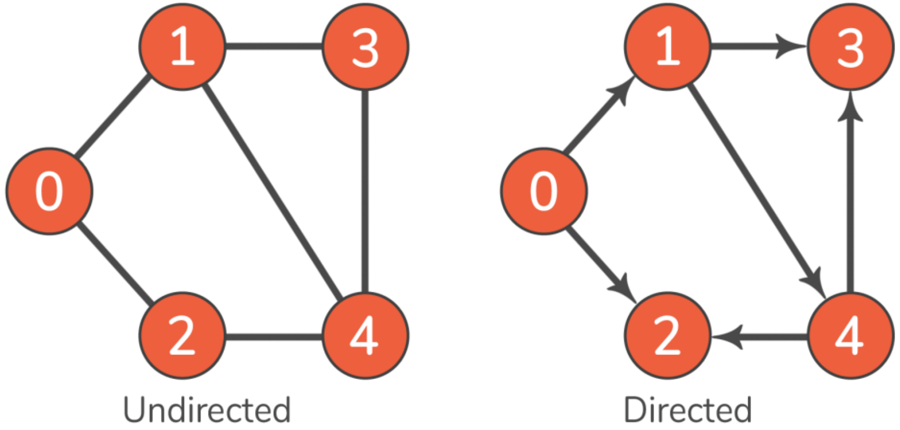
30, 50, 20, 56, 40, 22,14, 6,76,12,21,25,32,45,53



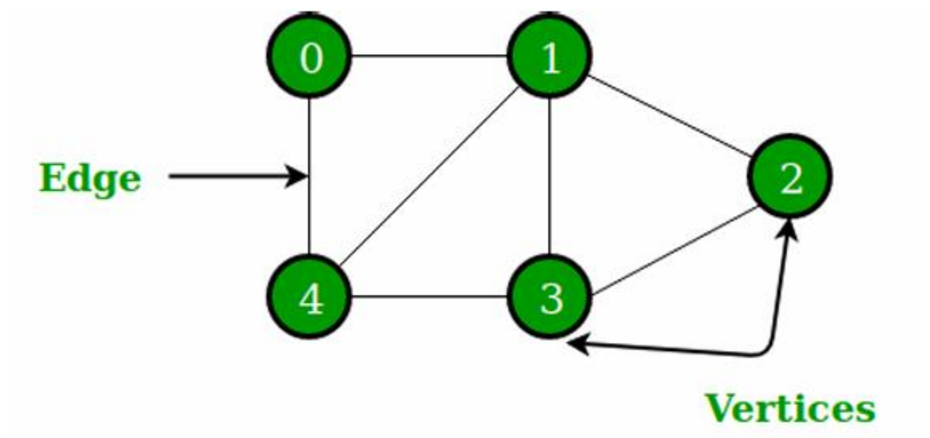
Pre-order Traversal:
30 20 14 6 12 22 21 25 50 40 32 45 56 53 76
In-order Traversal:
6 14 12 20 21 22 25 30 32 40 45 50 53 56 76
Post-order Traversal:
6 12 14 21 25 22 20 32 45 40 53 76 56 50 30

Graphs

- A graph is a nonlinear data structure consisting of nodes and edges.
- The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.
- More formally a Graph can be defined as a Graph consisting of a finite set of vertices (or nodes) and a set of edges that connect a pair of nodes.



Graphs



- In the above Graph, the set of vertices V = {0, 1, 2, 3, 4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.
- The following two are the most commonly used representations of a graph.
  - Adjacency Matrix
  - Adjacency List

Adjacency Matrix

- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.
- Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- The adjacency matrix for an undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs.
- If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

Adjacency Matrix

```
class Graph:
 def __init__(self,numvertex):
self.adjMatrix = [[-1]*numvertex for x in
range(numvertex)]
self.numvertex = numvertex
self.vertices = {}
self.verticeslist =[0]*numvertex
 def set_vertex(self,vtx,id):
 if 0<=vtx<=self.numvertex:
self.vertices[id] = vtx
self.verticeslist[vtx] = id

 def set_edge(self,frm,to,cost=0):
 frm = self.vertices[frm]
 to = self.vertices[to]
self.adjMatrix[frm][to] = cost
```

```
# for directed graph do not add this
self.adjMatrix[to][frm] = cost
 def get_vertex(self):
return self.verticeslist

def get_edges(self):
 edges=[]
 for i in range (self.numvertex):
for j in range (self.numvertex):
if (self.adjMatrix[i][j]!=-1):
edges.append((self.verticeslist[i],self.vert
iceslist[j],self.adjMatrix[i][j])) return
edges
 def get_matrix(self):
 return self.adjMatrix
```

```
G =Graph(6)
G.set_vertex(0,'a')
G.set_vertex(1,'b')
G.set_vertex(2,'c')
G.set_vertex(3,'d')
G.set_vertex(4,'e')
G.set_vertex(5,'f')
G.set_edge('a','e',10)
G.set_edge('a','c',20)
G.set_edge('c','b',30)
G.set_edge('b','e',40)
G.set_edge('e','d',50)
G.set_edge('f','e',60)
print("Vertices of Graph")
print(G.get_vertex())
print("Edges of Graph")
print(G.get_edges())
print("Adjacency Matrix of
Graph")
print(G.get_matrix())
```

Graph Example Class

__init__ Method
- constructor of the class, called when an object of the class is created.

```
class Graph:
```

1

```
    def __init__(self,numvertex):
        self.adjMatrix = [[-1]*numvertex for x in range(numvertex)]
        self.numvertex = numvertex
        self.vertices = {}
        self.verticeslist =[0]*numvertex
```

2

```
    def set_vertex(self,vtx,id):
        if 0<=vtx<=self.numvertex:
            self.vertices[id] = vtx
            self.verticeslist[vtx] = id
```

set_vertex Method
- Set up a mapping between a vertex ID and its position (index) in the adjacency matrix.
- Adds a mapping of id -> vtx in self.vertices.Updates self.verticeslist to store the id at the index vtx.

3

```
    def set_edge(self,frm,to,cost=0):
        frm = self.vertices[frm]
        to = self.vertices[to]
        self.adjMatrix[frm][to] = cost
```

set_edge Method
- Add an edge between two vertices in the graph and assigns a cost (weight) to the edge.
- Uses the vertex IDs (frm and to) to get their indices in the adjacency matrix from self.vertices.Updates the adjacency matrix to set the cost (or weight) of the edge between these indices.

Graph Example Class

4

```
def get_vertex(self):
        return self.verticeslist
```

5

```
    def get_edges(self):
        edges=[]
        for i in range (self.numvertex):
            for j in range (self.numvertex):
                if (self.adjMatrix[i][j]!=-1):
                    edges.append((self.verticeslist[i],self.verticeslist[j],self.adjMa
trix[i][j]))
        return edges
```

6

```
    def get_matrix(self):
        return self.adjMatrix
```

Create Graph Object

```
G =Graph(6)
G.set_vertex(0,'a')
G.set_vertex(1,'b')
G.set_vertex(2,'c')
G.set_vertex(3,'d')
G.set_vertex(4,'e')
G.set_vertex(5,'f')
G.set_edge('a','e',10)
G.set_edge('a','c',20)
G.set_edge('c','b',30)
G.set_edge('b','e',40)
G.set_edge('e','d',50)
G.set_edge('f','e',60)
```

```
print("Vertices of Graph")
print(G.get_vertex())
print("Edges of Graph")
print(G.get_edges())
print("Adjacency Matrix of Graph")
print(G.get_matrix())
```

#output

Vertices of Graph

[‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]

Edges of Graph

[(‘a’, ‘c’, 20), (‘a’, ‘e’, 10), (‘b’, ‘c’, 30), (‘b’, ‘e’, 40), (‘c’, ‘a’, 20), (‘c’, ‘b’, 30), (‘d’, ‘e’, 50), (‘e’, ‘a’, 10), (‘e’, ‘b’, 40), (‘e’, ‘d’, 50), (‘e’, ‘f’, 60), (‘f’, ‘e’, 60)]

Adjacency Matrix of Graph

[[-1, -1, 20, -1, 10, -1], [-1, -1, 30, -1, 40, -1], [20, 30, -1, -1, -1, -1], [-1, -1, -1, -1, 50, -1], [10, 40, -1, 50, -1, 60], [-1, -1, -1, -1, 60, -1]]

Graph Trace

```
G =Graph(6) #1
G.set_vertex(0,'a')
G.set_vertex(1,'b')
G.set_vertex(2,'c')
G.set_vertex(3,'d')
G.set_vertex(4,'e')
G.set_vertex(5,'f')
```

	0 (a)	1 (b)	2 (c)	3 (d)	4 (e)	5 (f)
0 (a)	-1	-1	20	-1	10	-1
1 (b)	-1	-1	-1	-1	40	-1
2 (c)	-1	30	-1	-1	-1	-1
3 (d)	-1	-1	-1	-1	-1	-1
4 (e)	-1	-1	-1	50	-1	-1
5 (f)	-1	-1	-1	-1	60	-1

Vertices of Graph
['a', 'b', 'c', 'd', 'e', 'f']
Edges of Graph
[('a', 'c', 20), ('a', 'e', 10), ('b', 'e', 40), ('c', 'b', 30), ('e', 'd', 50), ('f', 'e', 60)]
Adjacency Matrix of Graph
[[-1, -1, 20, -1, 10, -1], [-1, -1, -1, -1, 40, -1], [-1, 30, -1, -1, -1, -1], [-1, -1, -1, -1, -1, -1], [-1, -1, -1, 50, -1, -1], [-1, -1, -1, -1, 60, -1]]

Adjacency List

- An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[].
- An entry array[i] represents the list of vertices adjacent to the i^{th} vertex.
- This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.

Adjacency List

```
class AdjNode:
def __init__(self, data):
 self.vertex = data
 self.next = None
# A class to represent a graph. A graph
 # is the list of the adjacency lists.
 # Size of the array will be the no. of the
 # vertices "V"
class Graph:
 def __init__(self, ver ces):
self.V = ver ces
 self.graph = [None] * self.V
# Function to add an edge in an
undirected graph
def add_edge(self, src, dest):
```

```
# Adding the node to the source node
node = AdjNode(dest)
 node.next = self.graph[src]
self.graph[src] = node
 # Adding the source node to the
 destination as
# it is the undirected graph
node = AdjNode(src)
node.next = self.graph[dest]
self.graph[dest] = node
 # Function to print the graph
def print_graph(self):
 for i in range(self.V):
 print("Adjacency list of vertex {}\n
head".format(i), end="")
 temp = self.graph[i] while temp:
 print(" -> {}".format(temp.vertex),
end="")
temp = temp.next print(" \n")
```

```
# Driver program to the
above graph class
 if __name__ ==
"__main__":
 V = 5
graph = Graph(V)
graph.add_edge(0, 1)
graph.add_edge(0, 4)
graph.add_edge(1, 2)
graph.add_edge(1, 3)
graph.add_edge(1, 4)
graph.add_edge(2, 3)
graph.add_edge(3, 4)
graph.print_graph()
```

Adjacency List

1

```
class AdjNode:
    def __init__(self, data):
        self.vertex = data
        self.next = None

# A class to represent a graph.
class Graph:
```

2

```
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [None] * self.V
    # Function to add an edge in an undirected graph
```

3

```
    def add_edge(self, src, dest):
        # Adding the node to the source node
        node = AdjNode(dest)
        node.next = self.graph[src]
        self.graph[src] = node
```

1

```
        # Adding the source node to the destination node as it is an
undirected graph
        node = AdjNode(src)
        node.next = self.graph[dest]
        self.graph[dest] = node
```

1

Adjacency List

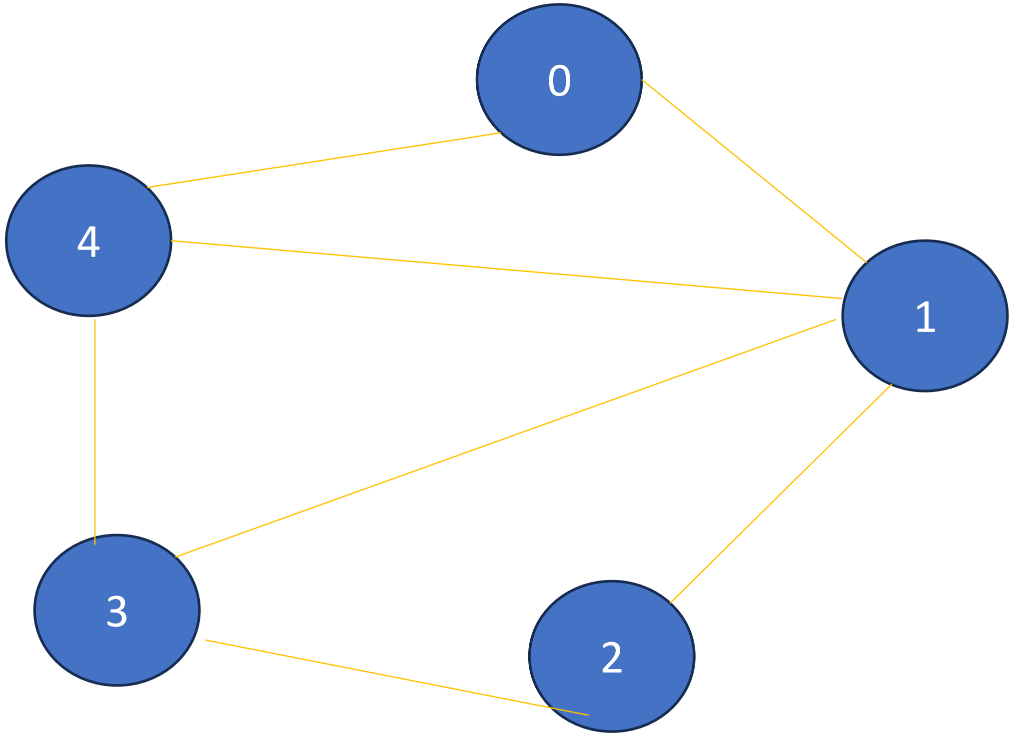
4

```
# Function to print the graph
    def print_graph(self):
        for i in range(self.V):
            print("Adjacency list of vertex {}\n head".format(i), end="")
            temp = self.graph[i]
            while temp:
                print(" -> {}".format(temp.vertex), end="")
                temp = temp.next
            print("\n")


# Driver program to the above graph class
if __name__ == "__main__":
    V = 5
    graph = Graph(V)
    graph.add_edge(0, 1)
    graph.add_edge(0, 4)
    graph.add_edge(1, 2)
    graph.add_edge(1, 3)
    graph.add_edge(1, 4)
    graph.add_edge(2, 3)
    graph.add_edge(3, 4)
    graph.print_graph()
```

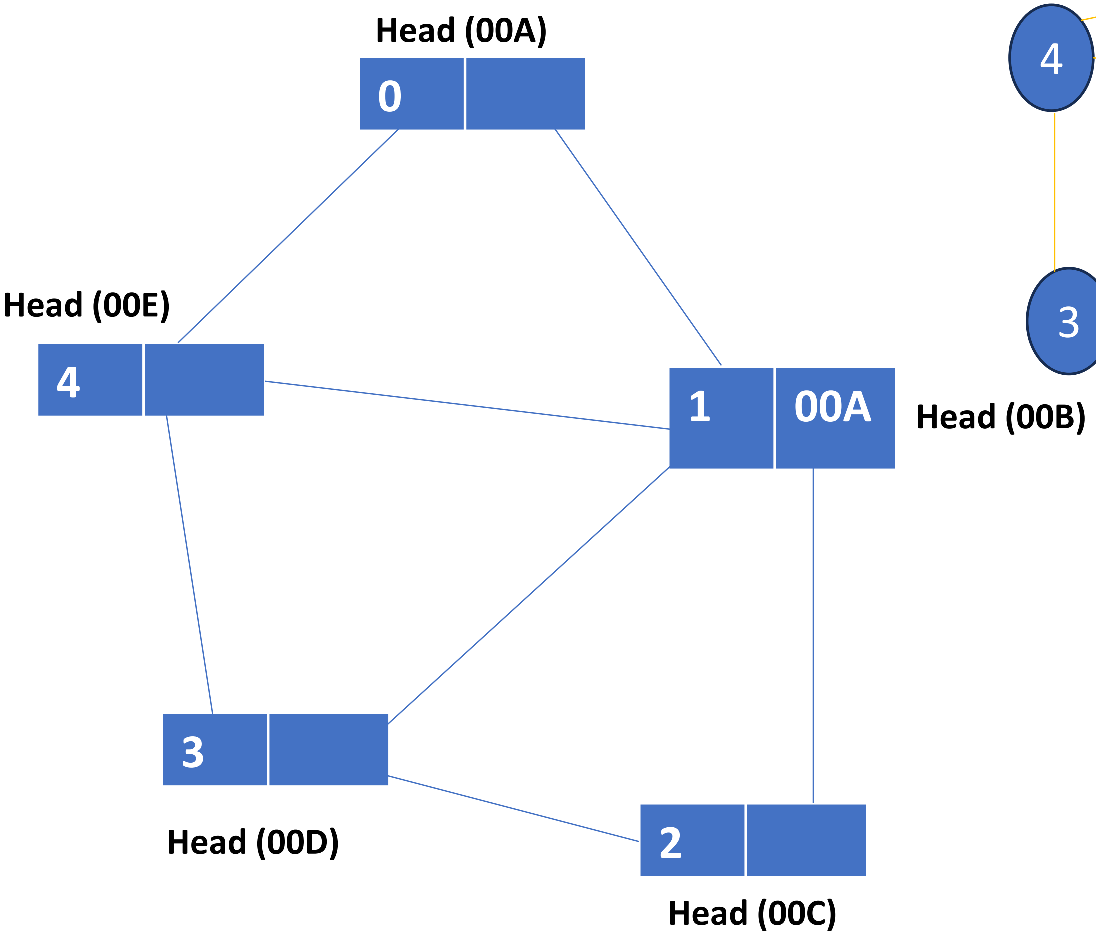
Graph Trace

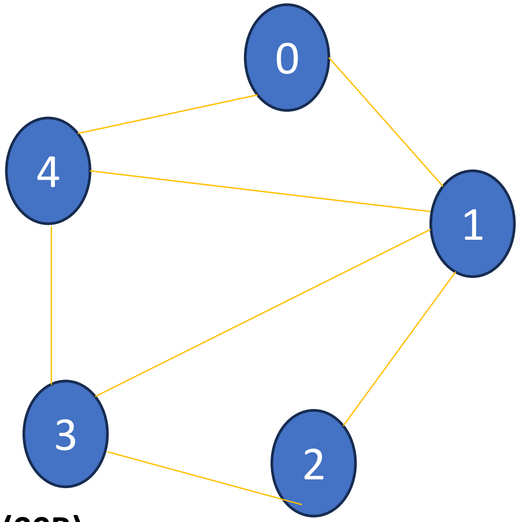
```
V = 5
    graph = Graph(V) #2
    graph.add_edge(0, 1)
    graph.add_edge(0, 4)
    graph.add_edge(1, 2)
    graph.add_edge(1, 3)
    graph.add_edge(1, 4)
    graph.add_edge(2, 3)
    graph.add_edge(3, 4)
    graph.print_graph()
```



Graph Trace

```
V = 5
    graph = Graph(V) #2
    graph.add_edge(0, 1)
    graph.add_edge(0, 4)
    graph.add_edge(1, 2)
    graph.add_edge(1, 3)
    graph.add_edge(1, 4)
    graph.add_edge(2, 3)
    graph.add_edge(3, 4)
    graph.print_graph()
```

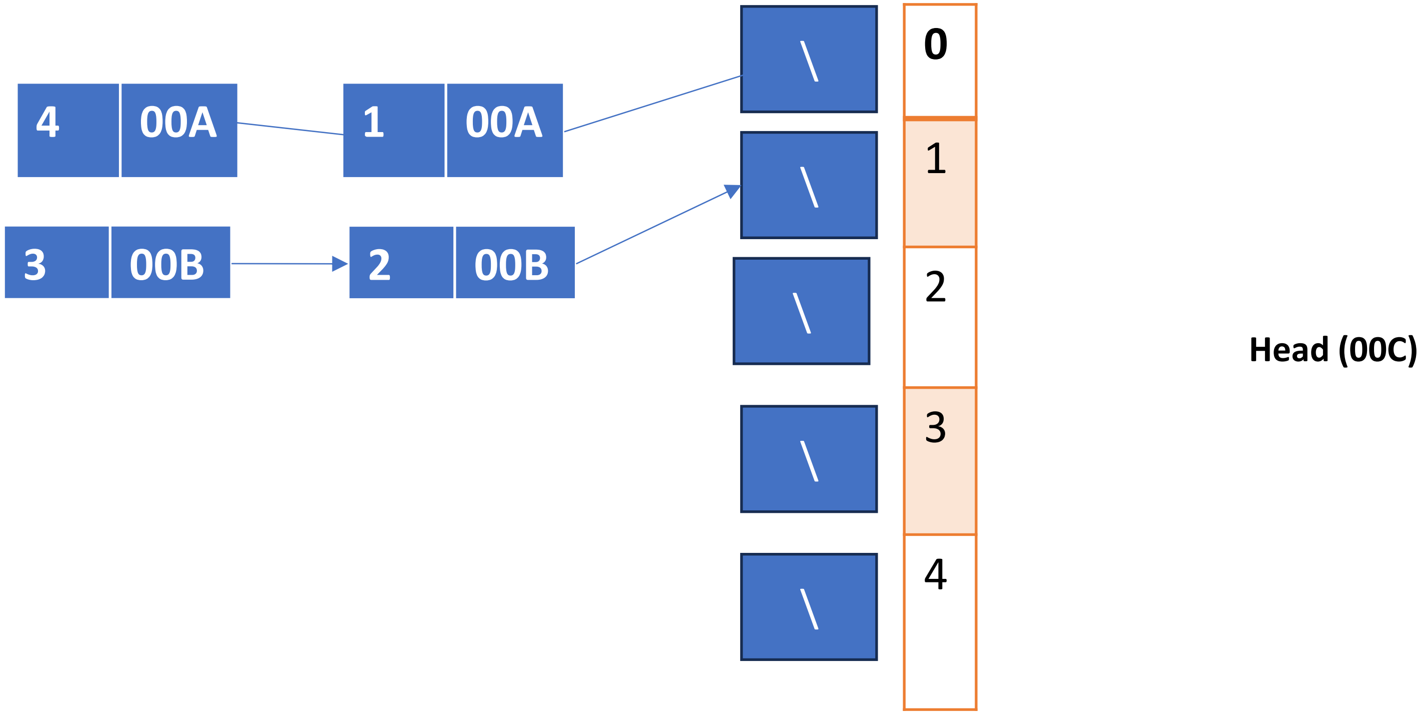




Adjacency list of vertex 0
head -> 4 -> 1
Adjacency list of vertex 1
head -> 4 -> 3 -> 2 -> 0
Adjacency list of vertex 2
head -> 3 -> 1
Adjacency list of vertex 3
head -> 4 -> 2 -> 1
Adjacency list of vertex 4
head -> 3 -> 1 -> 0

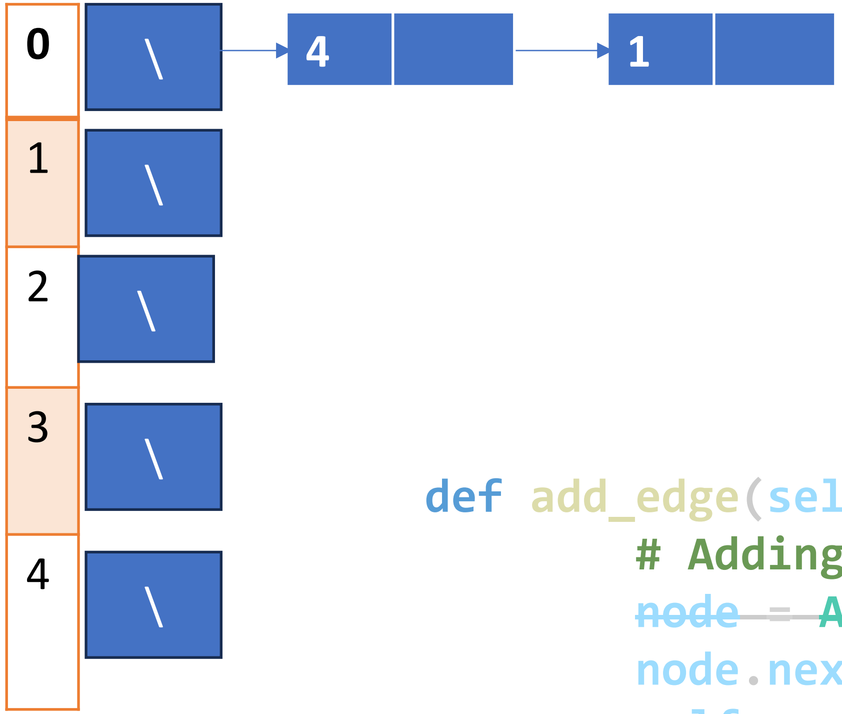
Graph Trace

```
V = 5
    graph = Graph(V) #2
    graph.add_edge(0, 1) #3
    graph.add_edge(0, 4)
    graph.add_edge(1, 2)
    graph.add_edge(1, 3)
    graph.add_edge(1, 4)
    graph.add_edge(2, 3)
    graph.add_edge(3, 4)
    graph.print_graph()
```



Graph Trace

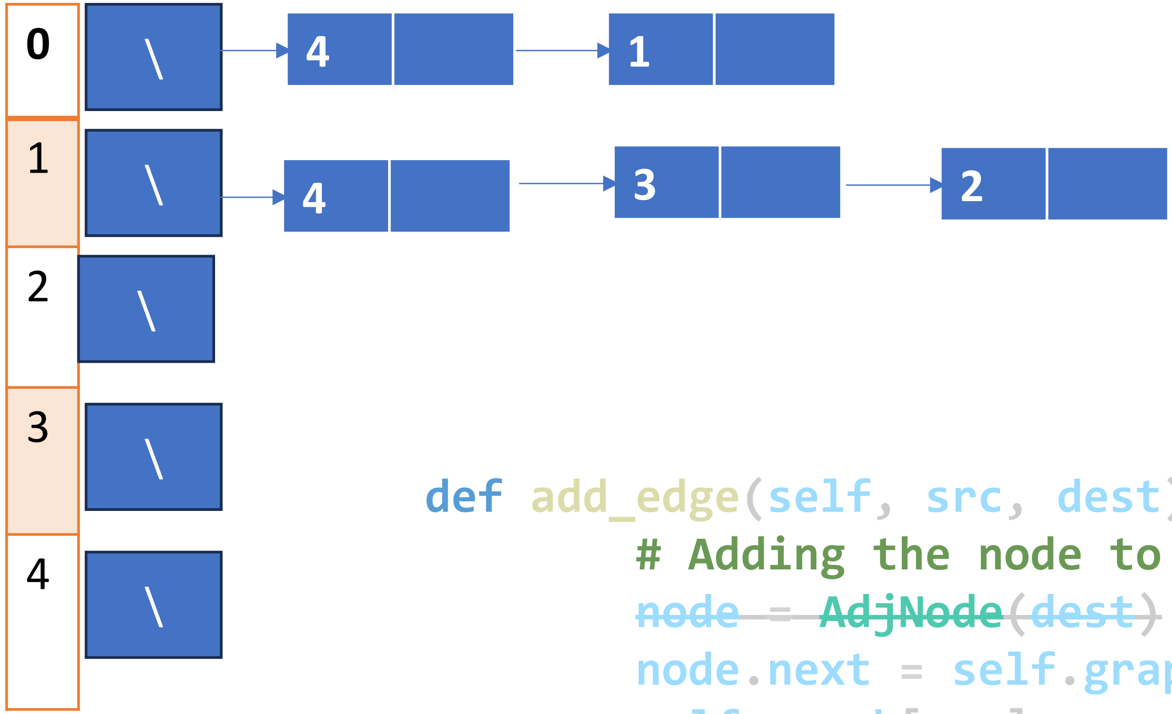
```
V = 5
    graph = Graph(V) #2
    graph.add_edge(0, 1) #3
    graph.add_edge(0, 4) #3
    graph.add_edge(1, 2)
    graph.add_edge(1, 3)
    graph.add_edge(1, 4)
    graph.add_edge(2, 3)
    graph.add_edge(3, 4)
    graph.print_graph()
```




```
def add_edge(self, src, dest):
        # Adding the node to the source node
        node = AdjNode(dest)
        node.next = self.graph[src]
        self.graph[src] = node
        # Adding the source node to the
destination node as it is an undirected graph
        node = AdjNode(src)
        node.next = self.graph[dest]
        self.graph[dest] = node
```

Graph Trace

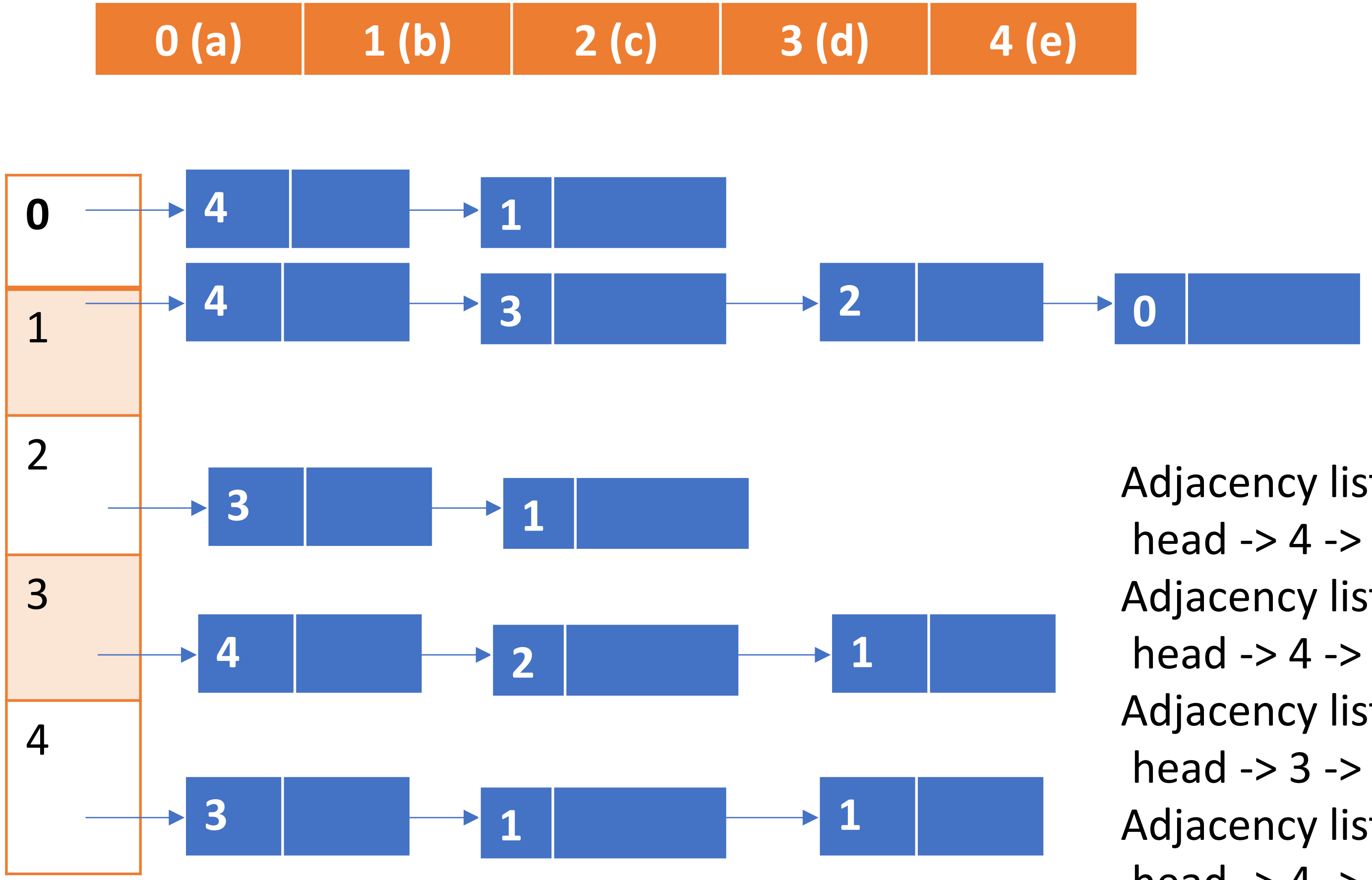
```
V = 5
    graph = Graph(V) #2
    graph.add_edge(0, 1) #3
    graph.add_edge(0, 4) #3
    graph.add_edge(1, 2)
    graph.add_edge(1, 3)
    graph.add_edge(1, 4)
    graph.add_edge(2, 3)
    graph.add_edge(3, 4)
    graph.print_graph()
```




```
def add_edge(self, src, dest):
        # Adding the node to the source node
        node = AdjNode(dest)
        node.next = self.graph[src]
        self.graph[src] = node
        # Adding the source node to the
destination node as it is an undirected graph
        node = AdjNode(src)
        node.next = self.graph[dest]
        self.graph[dest] = node
```

Graph Trace

```
V = 5
    graph = Graph(V)
    graph.add_edge(0, 1)
    graph.add_edge(0, 4)
    graph.add_edge(1, 2)
    graph.add_edge(1, 3)
    graph.add_edge(1, 4)
    graph.add_edge(2, 3)
    graph.add_edge(3, 4)
    graph.print_graph()
```



Adjacency list of vertex 0
head -> 4 -> 1
Adjacency list of vertex 1
head -> 4 -> 3 -> 2 -> 0
Adjacency list of vertex 2
head -> 3 -> 1
Adjacency list of vertex 3
head -> 4 -> 2 -> 1
Adjacency list of vertex 4
head -> 3 -> 1 -> 0

Adjacency List

Output

Adjacency list of vertex 0

head -> 4 -> 1

Adjacency list of vertex 1

head -> 4 -> 3 -> 2 -> 0

Adjacency list of vertex 2

head -> 3 -> 1

Adjacency list of vertex 3

head -> 4 -> 2 -> 1

Adjacency list of vertex 4

head -> 3 -> 1 -> 0

Realistic Infotech Group
IT Training & Services
No.79/A, First Floor
Corner of Insein Road and
Damaryon Street
Quarter (9), Hlaing Township
Near Thukha Bus Station
09256675642, 09953933826
http://www.rig-info.com