Advanced Programming

Chapter 5: Data Structure I & II

Kick-off

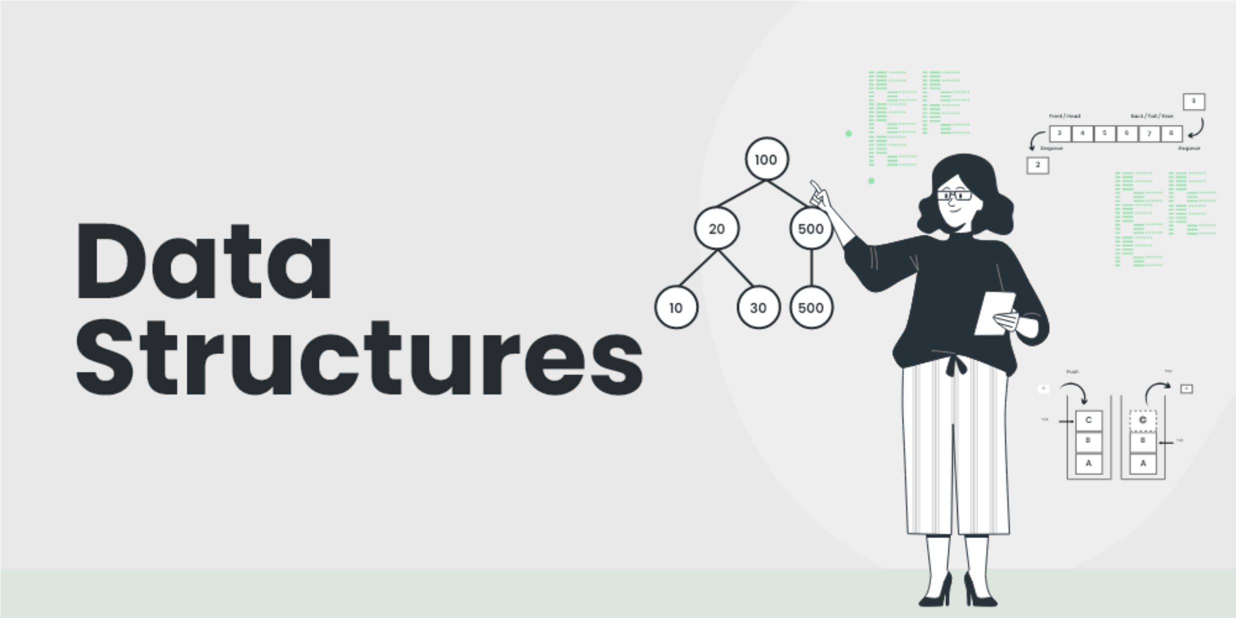
Use Anagram Solver to see a list of all possible words made from your input

10 Great Places to Find Free Datasets for Your Project

- Google Dataset Search
- Kaggle
- Data.Gov
- Datahub.io
- UCI Machine Learning Repository
- Earth Data
- CERN Open Data Portal
- Global Health Observatory Data Repository

Data Structures and algorithms

- Part of the "science" in computer science is the design and use of data structures and algorithm.
- As you go on in CS, you will learn more and more about these two areas.



Data Structures

- Data structures are particular ways of storing data to make some operation easier or more efficient. That is, they are tuned for certain tasks
- Data structures are suited to solving certain problems, and they are often associated with algorithms.

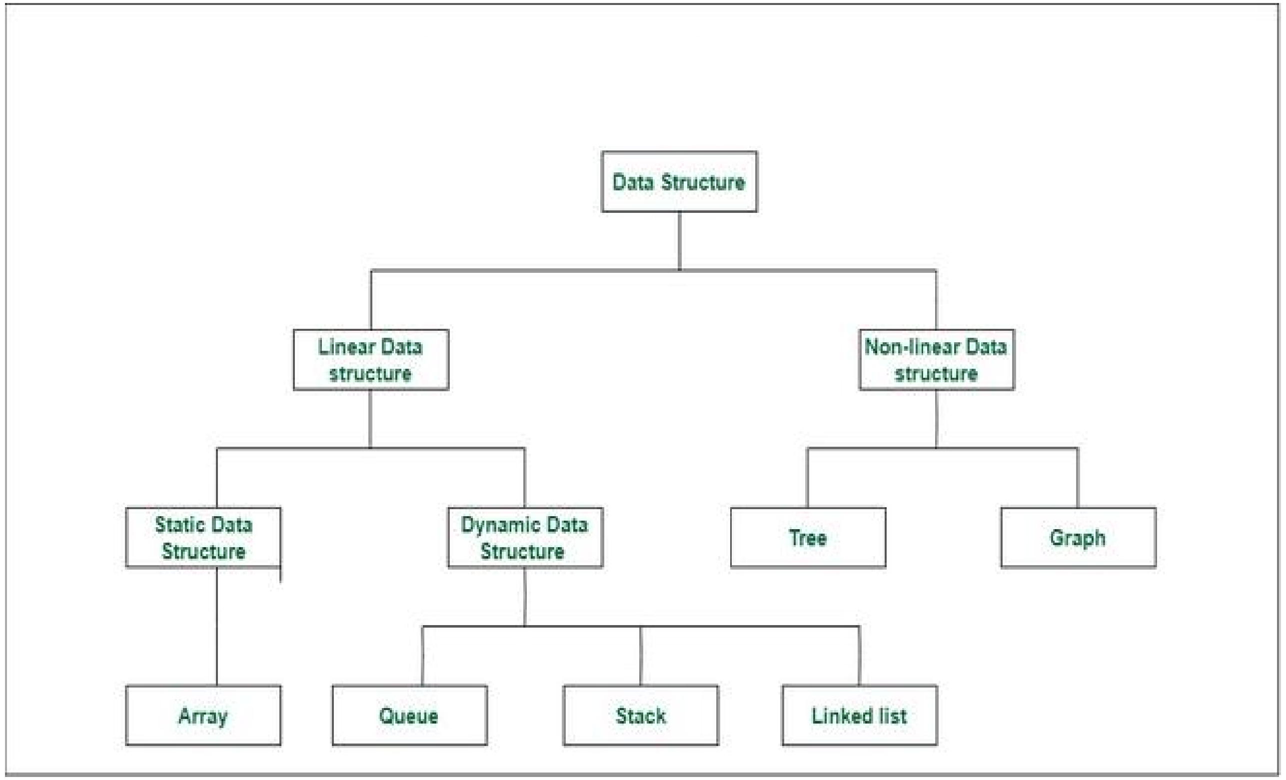
Classification of Data Structure

Linear Data Structure: Elements are arranged **sequentially**, where each element is attached to its previous and next adjacent elements.

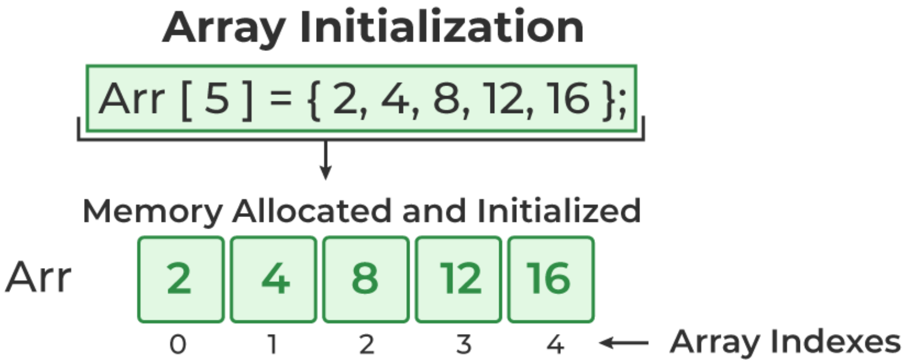
Static Data Structure: A fixed memory size. It is easier to access the elements in a static data structure.

Dynamic Data Structure: The size is **not fixed**.

Non-Linear Data Structure: Elements are not placed sequentially.

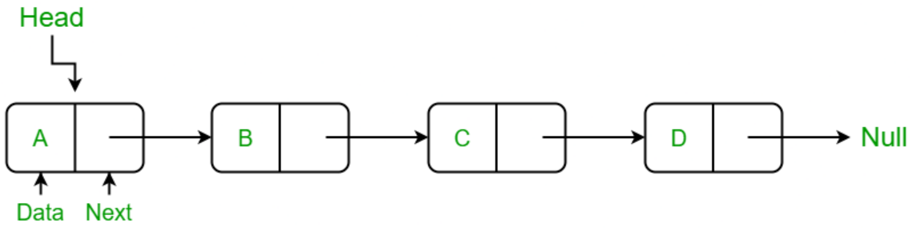


Most Popular Data Structures



1 - Array:

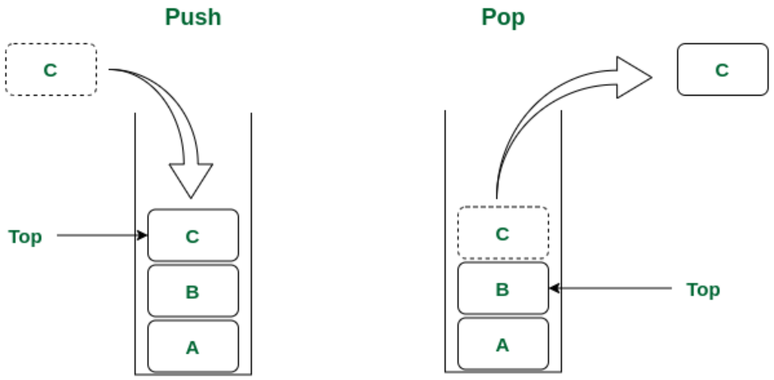
- A collection of data items stored at contiguous memory locations.
- The idea is to store multiple items of the same type together.



2 - Linked Lists:

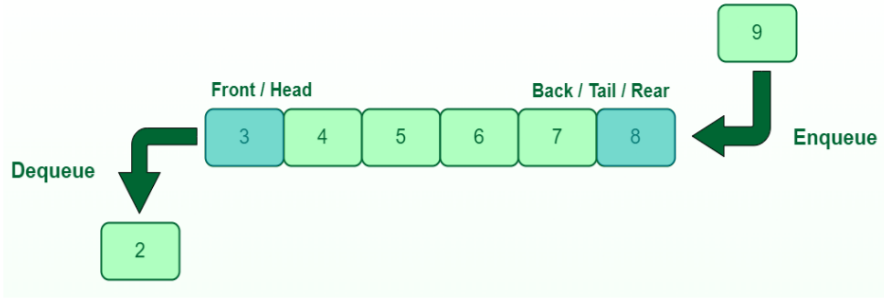
- Unlike arrays, linked list elements are not stored at a contiguous location.
- Elements are linked using pointers.

Most Popular Data Structures



3 - Stack:

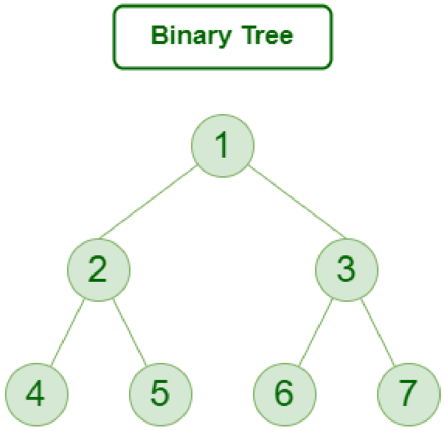
- Linear data structure.
- The order may be LIFO (Last In First Out) or FILO (First In Last Out).

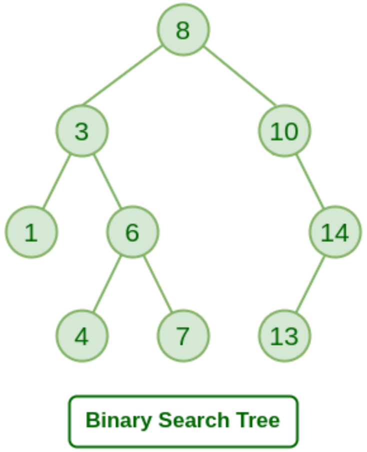


4 - Queue:

- Linear data structure
- The order is First In First Out (FIFO).
- The difference between stacks and queues is in removing. In a stack we remove the item **the most recently added**; in a queue, we remove the item the **least recently added.**

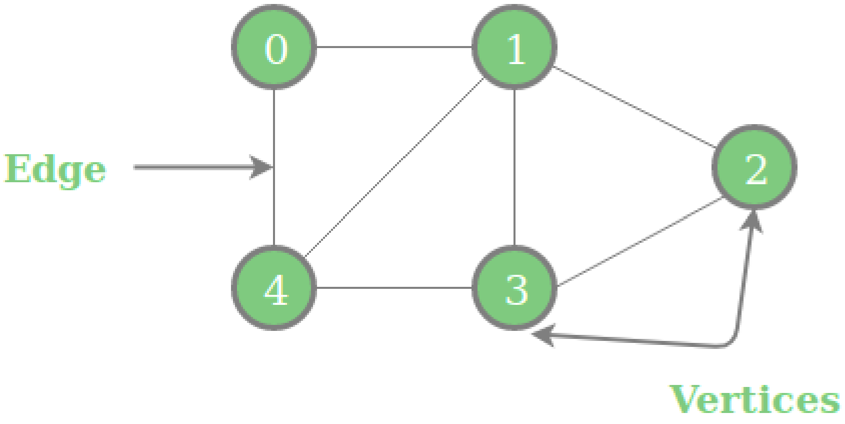
Most Popular Data Structures





5 - Binary Tree or Binary Tree Search:

- Hierarchical data structures.
- It is implemented mainly using Links.
- Binary Search Tree is a Binary Tree with additional properties.



6 - Graph:

- Collection of nodes (vertices) connected by edges.
- To represent relationships between objects.

Kinds of data structures

Roughly two kinds of data structures:

- built-in data structures, data structures that are so common as to be provided by default
- user-defined data structures (classes in object-oriented programming) that are designed for a particular task

Operations on various Data Structures

- **Traversing**: To visit the element stored in it. It visits data in a systematic manner.

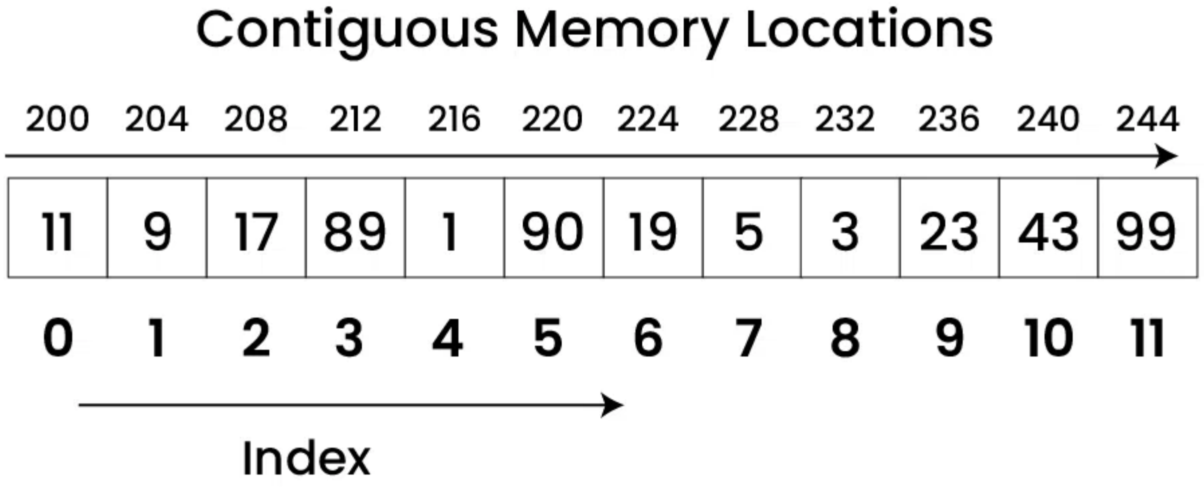
```
# Python program to traversal in an array

if __name__ == '__main__':

        # Initialise array
        arr = [ 1, 2, 3, 4 ];

        # size of array
        N = len(arr);

        # Traverse the element of arr
        for i in range(N):

                # Print element
                print(arr[i], end=" ");
```

Operations on various Data Structures

- **Searching**: To find a particular element in the given data-structure.
- **Insertion**: To add an element in the given data.
- **Deletion:** To delete an element in the given data.
- **Update**: To update any specific data by giving some condition in loop like select approach.
- **Sort**: Sorting data in a particular order (ascending or descending).
- **Merge**: Merging data of two different orders in a specific order may ascend or descend.
- **Split Data**: Dividing data into different sub-parts to make the process complete in less time.

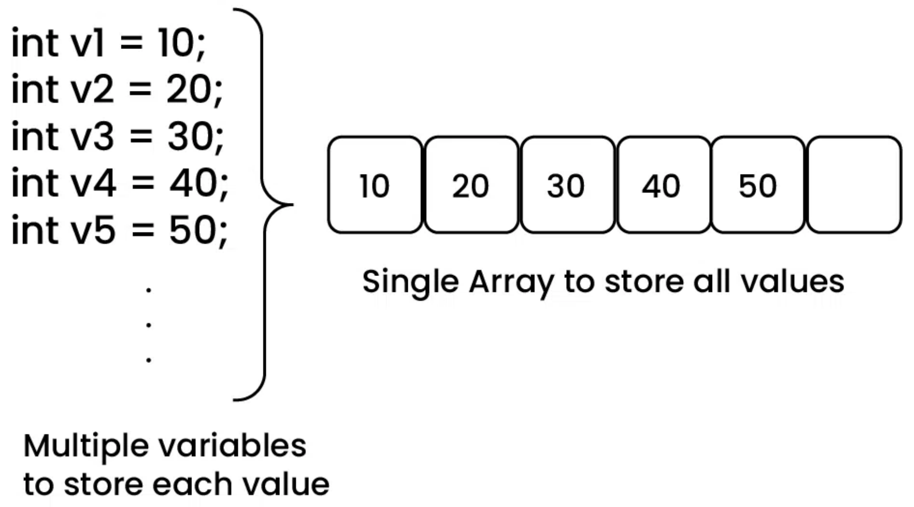
Array

Memory Representation of Array



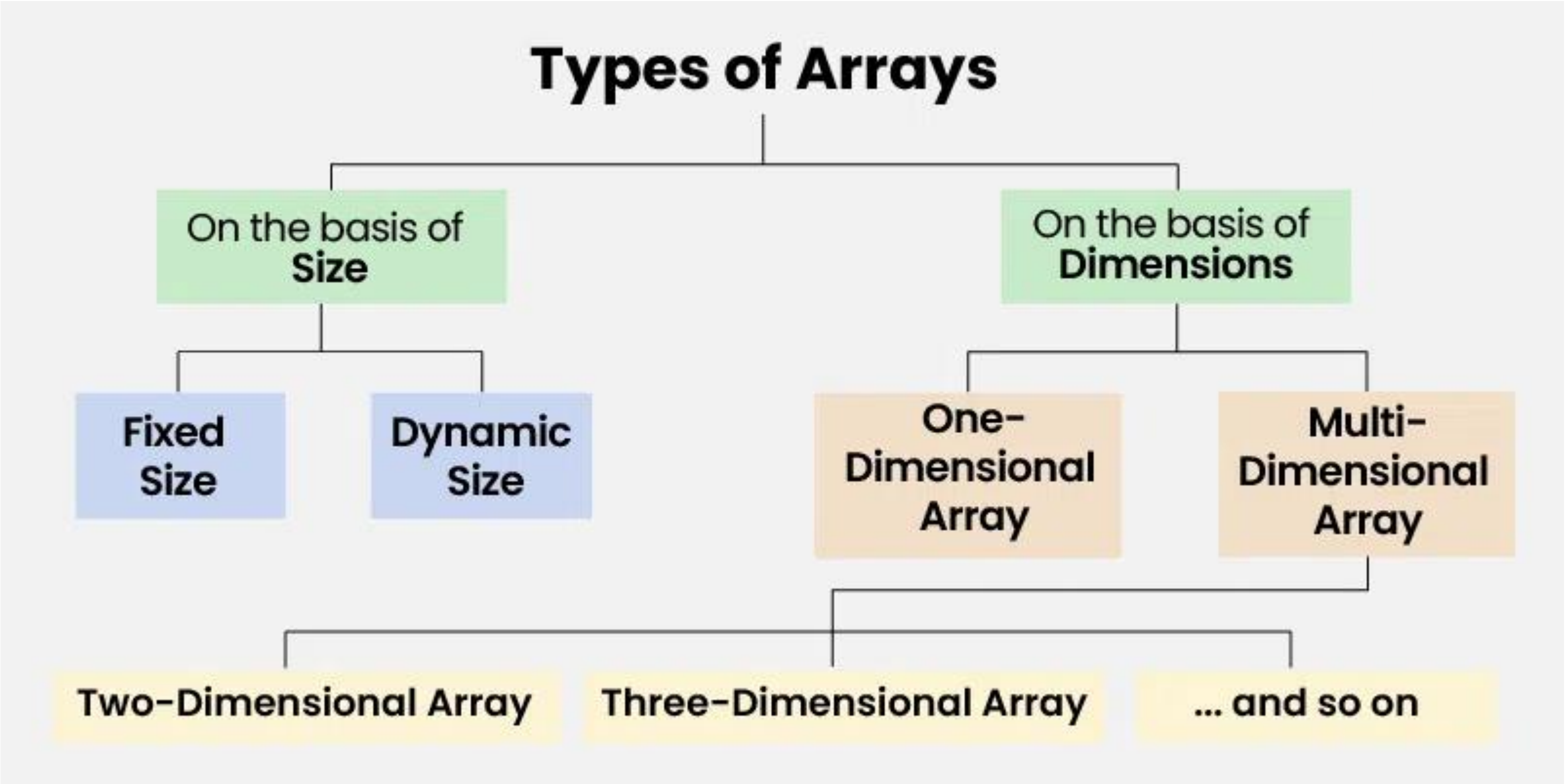
- In an array, all the elements are stored in contiguous memory locations.
- When initializing an array, the elements will be allocated sequentially in memory.

Importance of Array

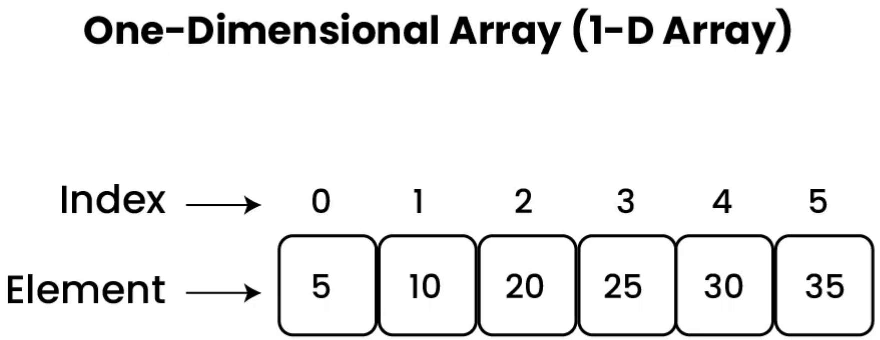


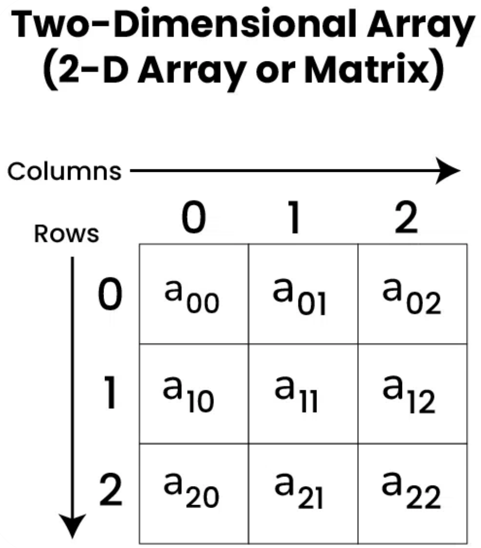
- If we want to store a large number of instances, it becomes difficult to manage them with normal variables.
- The idea of an array is to represent many instances in one variable.

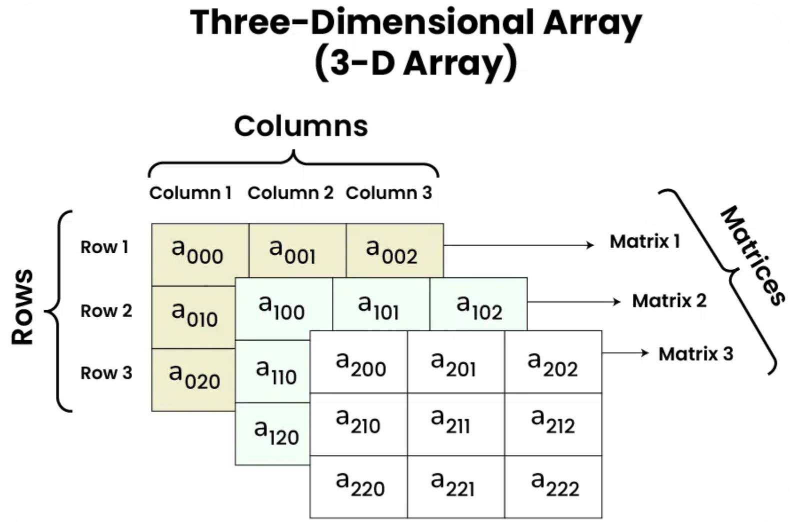
Type of Arrays



Types of Arrays on the basis of Dimensions







Operations on Array

- Array Traversal

```
import array arr = array.array('i', [1, 2, 3, 4, 5]) #
Traversing over arr[]
for x in arr:
(x, end=" ")
```

- Insertion in Array
- Deletion in Array
- Searching in Array

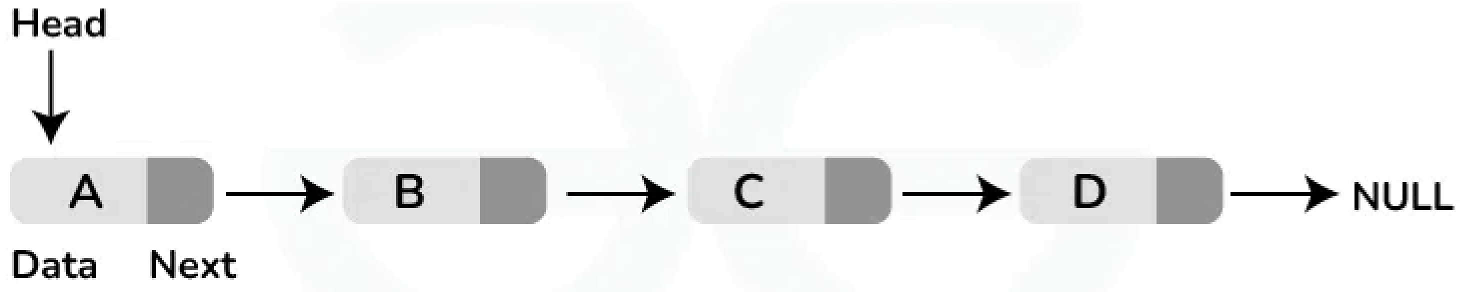
Linked List

Linked List vs Array

	Array	Linked List
• **Data Structure**	• Contiguous	• Non-contiguous
• **Memory Allocation**	• Typically allocated to the whole array	• Typically allocated one by one to individual elements
• **Insertion/Deletion**	• Inefficient	• Efficient
• **Access**	• Random	• Sequential

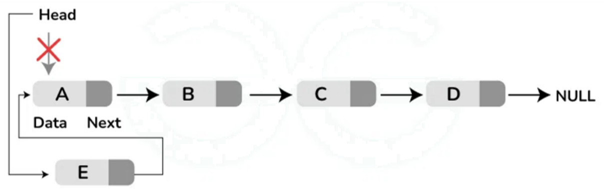
Singly Linked List

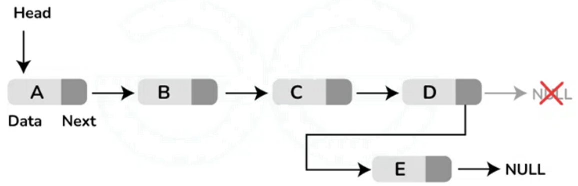
- A collection of nodes where each node contains a data field and a reference (link) to the next node in the sequence.
- An arrow indicating the link to the next node.
- Each node consists of two parts:
  - **data** - stores the actual information.
  - **pointer** (or reference) - stores the address of the next node in the sequence.

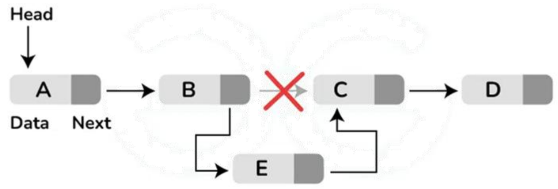


Operations on Singly Linked List

- Traversal
- Searching
- Length
- **Insertion**
  - Insert at the beginning
  - Insert at the end
  - Insert at a specific position
- Deletion
  - Delete from the beginning
  - Delete from the end
  - Delete a specific node







Python built in data structures

- Python comes with a general set of built-in data structures:
    - lists
    - tuples
    - string
    - dictionaries
    - sets
    - others...

Lists

The Python List Data Structure

- a list is an ordered sequence of items.
- you have seen such a sequence before in a string. A string is just a particular kind of list (what kind)?

Make a List

- Like all data structures, lists have a ***constructor***, named the same as the data structure. It takes an iterable data structure and ***adds each item*** to the list
- It also has a shortcut, the use of square brackets [ ] to indicate explicit items.

make a list

```
>>> a_list = [1,2,'a',3.14159]
>>> week_days_list = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> list_of_lists = [ [1,2,3], ['a','b','c']]
>>> list_from_collection = list('Hello')
>>> a_list
[1, 2, 'a', 3.1415899999999999]
>>> week_days_list
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> list_of_lists
[[1, 2, 3], ['a', 'b', 'c']]
>>> list_from_collection
['H', 'e', 'l', 'l', 'o']
>>> []
[]
>>>
```

Similarities with strings

- concatenate/+ (but only of lists)
- repeat/*
- indexing (the [ ] operator)
- slicing ([:])
- membership (the in operator)
- len (the length operator)

Operators

```
[1, 2, 3] + [4] ⇒ [1, 2, 3, 4]

[1, 2, 3] * 2 ⇒ [1, 2, 3, 1, 2, 3]

1 in [1, 2, 3] ⇒ True

[1, 2, 3] < [1, 2, 4] ⇒ True
```

compare index to index, first difference determines the result

differences between lists and strings

- lists can contain a mixture of any python object, strings can only hold characters
  - 1,"bill",1.2345, True
- lists are mutable, their values can be changed, while strings are immutable
- lists are designated with [ ], with elements separated by commas, strings use " " or ' '

```
myList = [1, 'a', 3.14159, True]
```

myList

1	'a'	3.14159	True	
0	1	2	3	Index forward
−4	−3	−2	−1	Index backward

```
myList[1]  →  'a'
myList[:3]  →  [1, 'a', 3.14159]
```

FIGURE 7.1 The structure of a list.

Indexing

- can be a little confusing, what does the [ ] mean, a list or an index?

    `[1, 2, 3][1]` $\Rightarrow$ `2`

- Context solves the problem. Index always comes at the end of an expression, and is preceded by something (a variable, a sequence)

List of Lists

```
my_list = ['a', [1, 2, 3], 'z']
```

- What is the second element (index 1) of that list? Another list.

```
my_list[1][0] # apply left to right
 my_list[1] ⇒ [1, 2, 3]
 [1, 2, 3][0] ⇒ 1
```

List Functions

- `len(lst)`: number of elements in list (top level). `len([1, [1, 2], 3])` ⇒ `3`
- `min(lst)`: smallest element. Must all be the same type!
- `max(lst)`: largest element, again all must be the same type
- `sum(lst)`: sum of the elements, numeric only

Iteration

You can iterate through the elements of a list like you did with a string:

```
>>> my_list = [1,3,4,8]
>>> for element in my_list:        # iterate through list elements
        print(element ,end=' ') # prints on one line

1 3 4 8
>>>
```

Mutable

Change an object's contents

- strings are immutable. Once created, the object's contents cannot be changed. New objects can be created to reflect a change, but the object itself cannot be changed

```
my_str = 'abc'
my_str[0] = 'z'   # cannot do!
# instead, make new str
new_str = my_str.replace('a','z')
```

Lists are mutable

Unlike strings, lists are mutable. You ***can*** change the object's contents!

```
my_list = [1, 2, 3]
my_list[0] = 127
print(my_list) ⇒ [127, 2, 3]
```

List methods

- Remember, a function is a small program (such as len) that takes some arguments, the stuff in the parenthesis, and returns some value
- a method is a function called in a special way, the ***dot call***. It is called in the context of an object (or a variable associated with an object)

Again, lists have methods

```
my_list = ['a',1,True]
my_list.append('z')
```

arguments to
the method

the object that
we are calling the
method with

the name of
the method

Some new methods

- A list is mutable and can change:
  - `my_list[0]='a'  #index assignment`
  - `my_list.append(), my_list.extend()`
  - `my_list.pop()`
  - `my_list.insert(), my_list.remove()`
  - `my_list.sort()`
  - `my_list.reverse()`

More about list methods

- most of these methods ***do not return a value***
- This is because lists are mutable, so the methods modify the list directly. No need to return anything.
- Can be confusing

Range

- We have seen the range function before. It generates a sequence of integers.
- In fact what it generates is a list with that sequence:

```
myList = range(1,5)
myList is [1,2,3,4]
```

Split

- The string method split generates a sequence of characters by splitting the string at certain split-characters.
- ***It returns a list*** (we didn't mention that before)

```
split_list = 'this is a test'.split()
split_list
```

$\Rightarrow$ `['this', 'is', 'a', 'test']`

Sorting

Only lists have a built in sorting method. Thus you often convert your data to a list if it needs sorting

```
my_list = list('xyzabc')
my_list →['x','y','z','a','b','c']
my_list.sort()    # no return
my_list →
     ['a', 'b', 'c', 'x', 'y', 'z']
```

reverse words in a string

`join` method of string places the calling string between every element of a list

```
>>> my_str = 'This is a test'
>>> string_elements = my_str.split()                    # list of words
>>> string_elements
['This', 'is', 'a', 'test']
>>> reversed_elements = []
>>> for element in string_elements:                     # for each word
...     reversed_elements.append(element[::-1])         # reverse , append
...
>>> reversed_elements
['sihT', 'si', 'a', 'tset']
>>> new_str = ' '.join(reversed_elements)    # join with space separator
>>> new_str
'sihT si a tset'                             # each words reversed
>>>
```

Sorted function

The `sorted` function will break a sequence into elements and sort the sequence, placing the results in a list

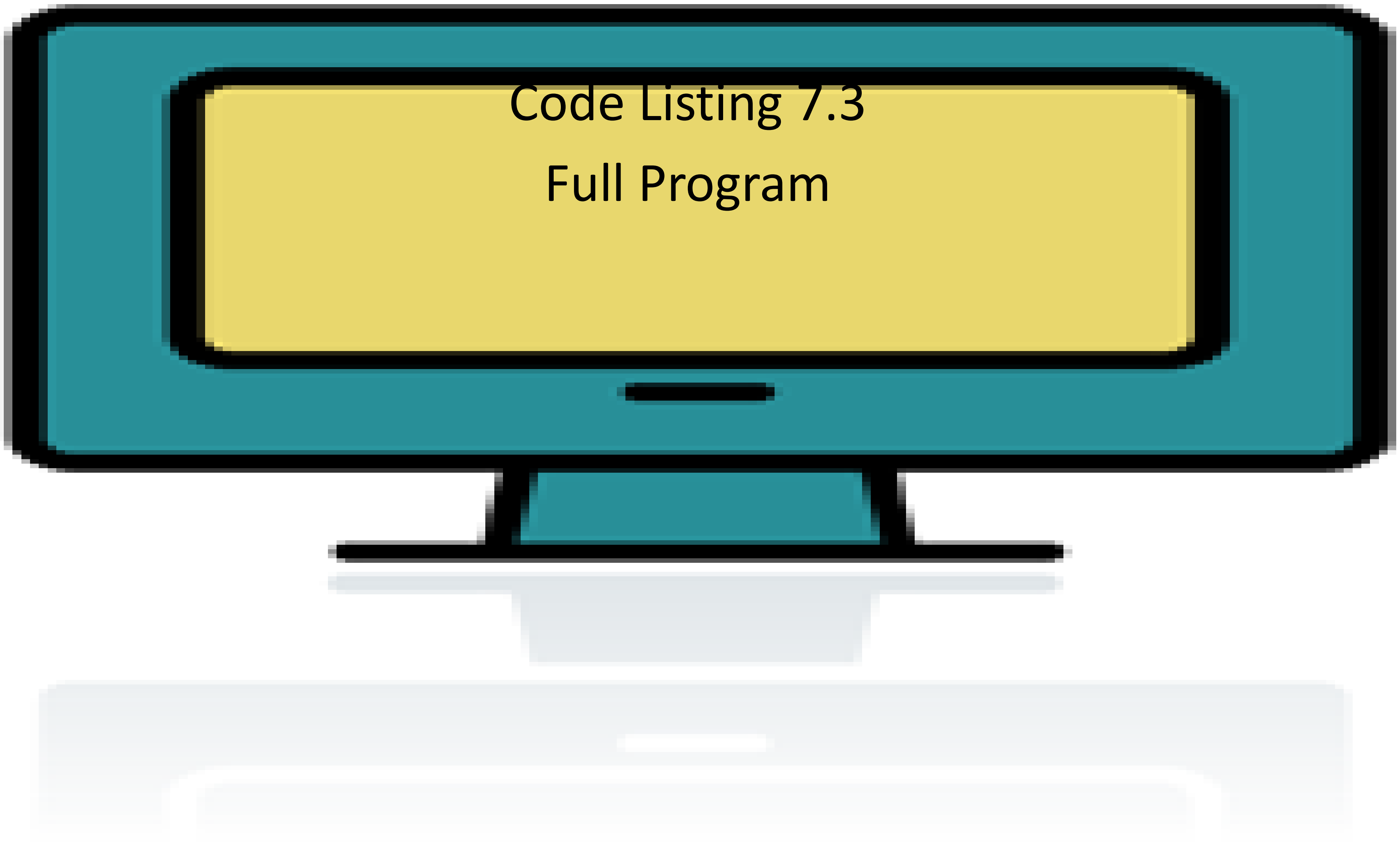
```
sort_list = sorted('hi mom')
sort_list ⇒
  [‘ ’,'h','i','m','m','o']
```

Some Examples

Anagram example

- Anagrams are words that contain the same letters arranged in a different order. For example: 'iceman' and 'cinema'
- Strategy to identify anagrams is to take the letters of a word, sort those letters, than compare the sorted sequences. Anagrams should have the same sorted sequence

```python
def are_anagrams(word1, word2):
    """Return True, if words are anagrams."""
    #2. Sort the characters in the words
    word1_sorted = sorted(word1)    # sorted returns a sorted list
    word2_sorted = sorted(word2)

    #3. Check that the sorted words are identical.
    if word1_sorted == word2_sorted:  # compare sorted lists
        return True
    else:
        return False
```


Code Listing 7.3
Full Program

```python
def are_anagrams(word1, word2):
    """Return True, if words are anagrams."""
    #2. Sort the characters of the words.
    word1_sorted = sorted(word1)    # sorted returns a sorted list
    word2_sorted = sorted(word2)

    #3. Check that the sorted words are identical.
    return word1_sorted == word2_sorted

print("Anagram Test")

# 1. Input two words.
two_words = input("Enter two space separated words: ")
word1,word2 = two_words.split()  # split into a list of words

if are_anagrams(word1, word2):    # return True or False
    print("The words are anagrams.")
else:
    print("The words are not anagrams.")
```

Code Listing 7.4

Check those errors

repeat input prompt for valid input

```
valid_input_bool = False
while not valid_input_bool:
    try:
        two_words = input("Enter two …")
        word1, word2 = two_words.split()
        valid_input_bool = True
    except ValueError:
        print("Bad Input")
```

only runs when no error, otherwise go around again

```python
def are_anagrams(word1, word2):
    """Return True, if words are anagrams."""
    #2. Sort the characters of the words.
    word1_sorted = sorted(word1)    # sorted returns a sorted list
    word2_sorted = sorted(word2)

    #3. Check that the sorted words are identical.
    return word1_sorted == word2_sorted

print("Anagram Test")

# 1. Input two words, checking for errors now
valid_input_bool = False
while not valid_input_bool:
    try:
        two_words = input("Enter two space separated words: ")
        word1,word2 = two_words.split()  # split the input string into a list
                                         of words
        valid_input_bool = True
    except ValueError:
        print("Bad Input")

if are_anagrams(word1, word2):    # function returned True or False
    print("The words {} and {} are anagrams.".format(word1, word2))
else:
    print("The words {} and {} are not anagrams.".format(word1, word2))
```

Code Listing 7.5

Words from text file

```python
def make_word_list(a_file):
    """Create a list of words from the file."""
    word_list = []      # list of speech words: initialized to be empty

    for line_str in a_file:                 # read file line by line
        line_list = line_str.split()   # split each line into a list of words
        for word in line_list:              # get words one at a time from list
            if word != "--":                # if the word is not "——"
                word_list.append(word)   # add the word to the speech list
    return word_list
```

Code Listing 7.7

Unique Words, Gettysburg Address

```python
# Gettysburg address analysis
# count words, unique words

def make_word_list(a_file):
    """Create a list of words from the file."""
    word_list = []       # list of speech words: initialized to be empty

    for line_str in a_file:                  # read file line by line
        line_list = line_str.split()         # split each line into a list of words
        for word in line_list:               # get words one at a time from list
            if word != "--":                 # if the word is not "--"
                word_list.append(word)       # add the word to the speech list
    return word_list

def make_unique(word_list):
    """Create a list of unique words."""
    unique_list = []   # list of unique words: initialized to be empty

    for word in word_list:           # get words one at a time from speech
        if word not in unique_list:  # if word is not already in unique list,
            unique_list.append(word) # add word to unique list

    return unique_list

################################

gba_file = open("gettysburg.txt", "r")
speech_list = make_word_list(gba_file)

# print the speech and its lengths
print(speech_list)
print("Speech Length: ", len(speech_list))
print("Unique Length: ", len(make_unique(speech_list)))
```

More about mutables

Reminder, assignment

- Assignment takes an object (the final object after all operations) from the RHS and associates it with a variable on the left-hand side
- When you assign one variable to another, you ***share the association*** with the same object

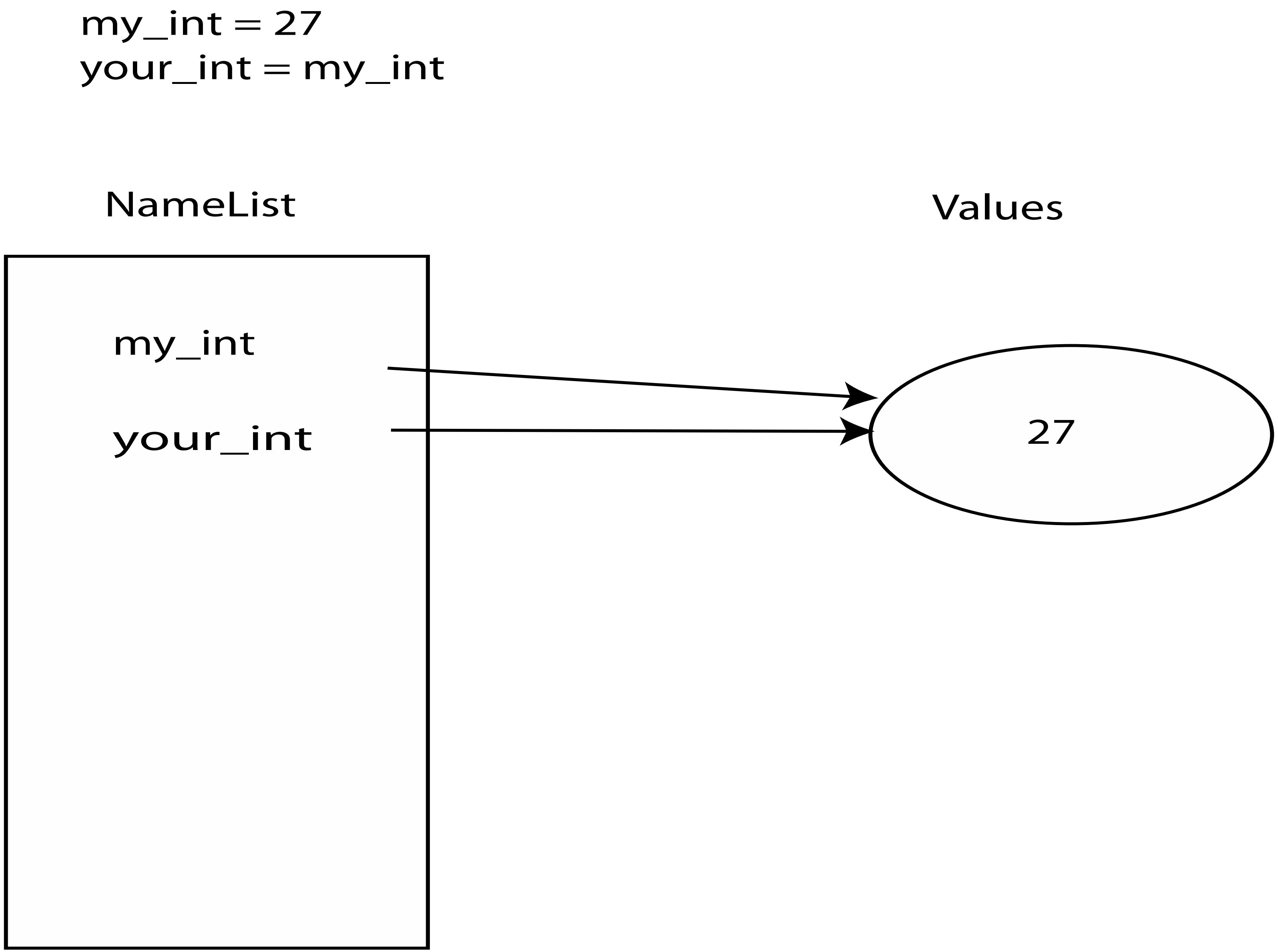


FIGURE 7.2 Namespace snapshot #1.

immutables

- Object sharing, two variables associated with the same object, is not a problem since the object cannot be changed
- Any changes that occur generate a ***new*** object.

```
my_int = 27
your_int = my_int
your_int = your_int + 1
```

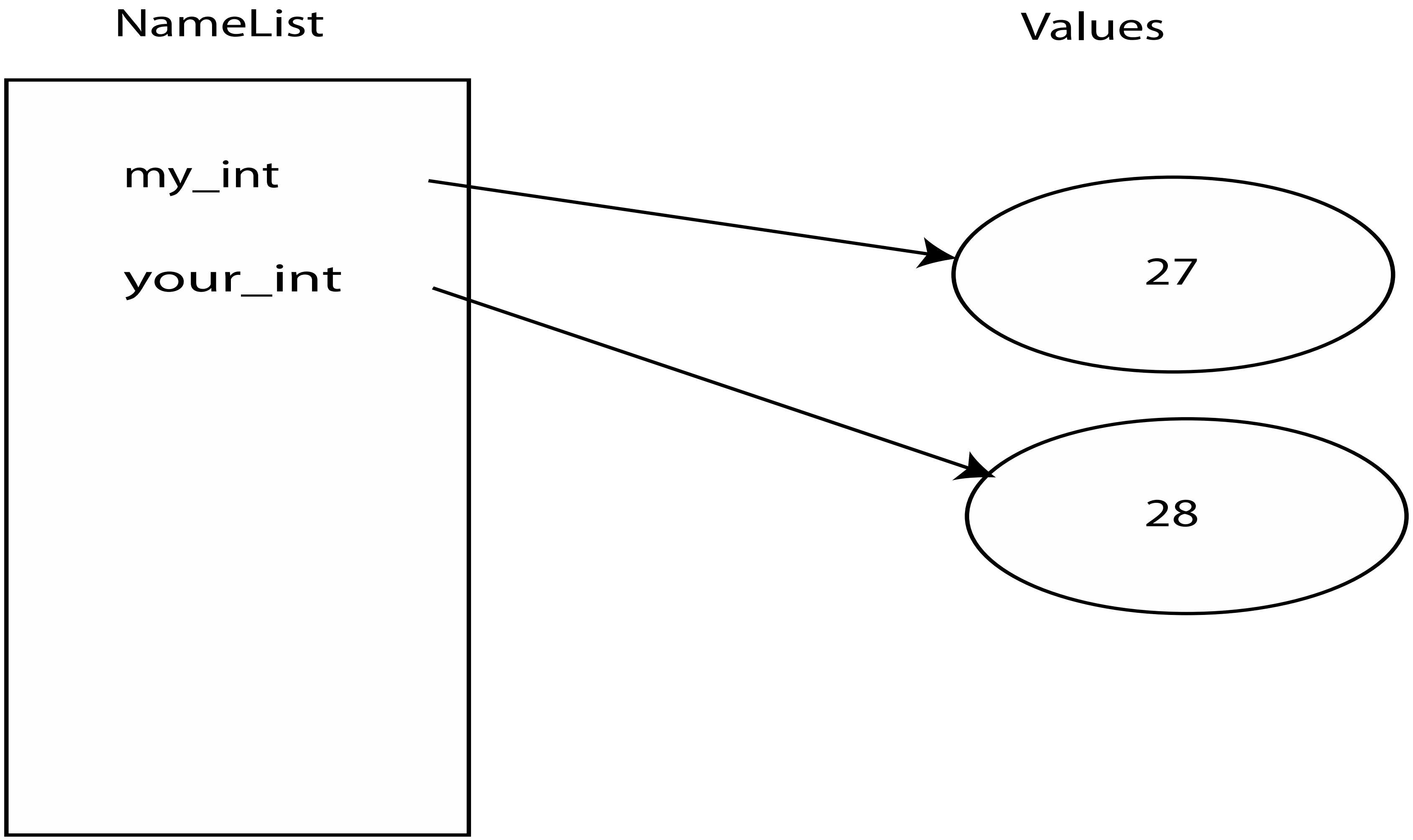


FIGURE 7.3 Modification of a reference to an immutable object.

Mutability

- If two variables associate with the same object, then ***both reflect*** any change to that object

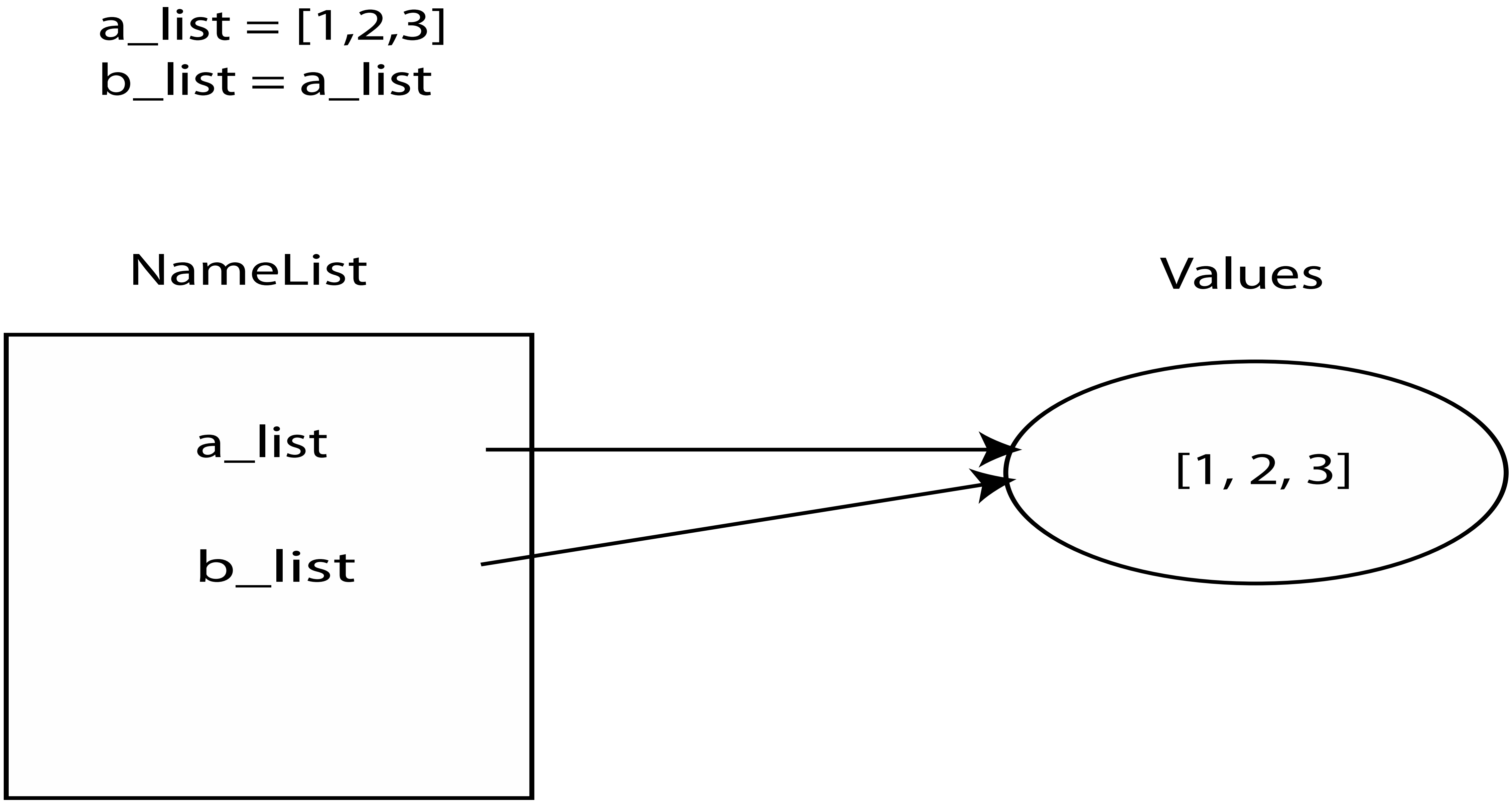


FIGURE 7.4 Namespace snapshot after assigning mutable objects.

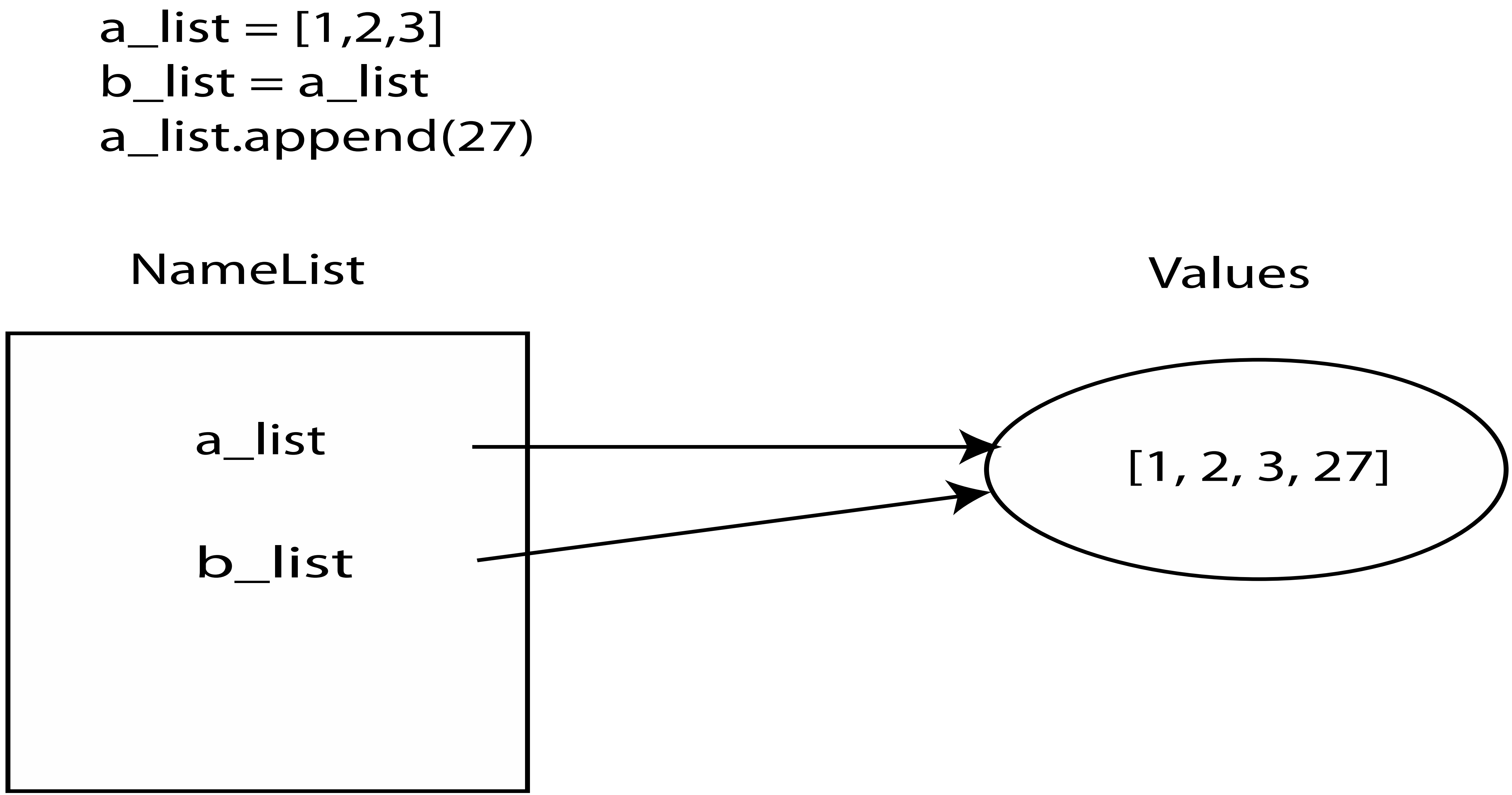


FIGURE 7.5 Modification of shared, mutable objects.

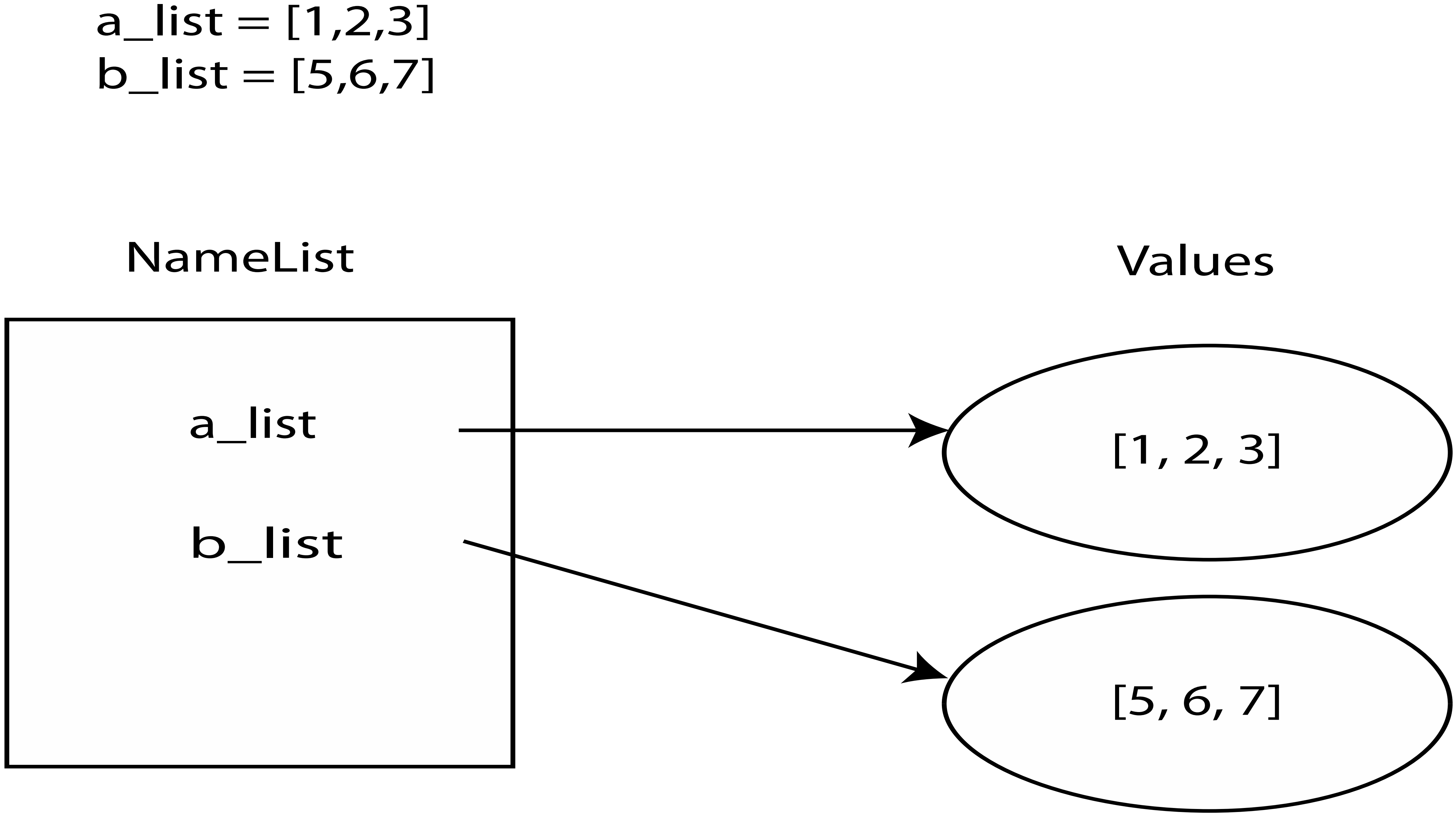


FIGURE 7.8 Simple lists before append.

```
a_list = [1,2,3]
b_list = [5,6,7]
a_list.append(b_list)
```

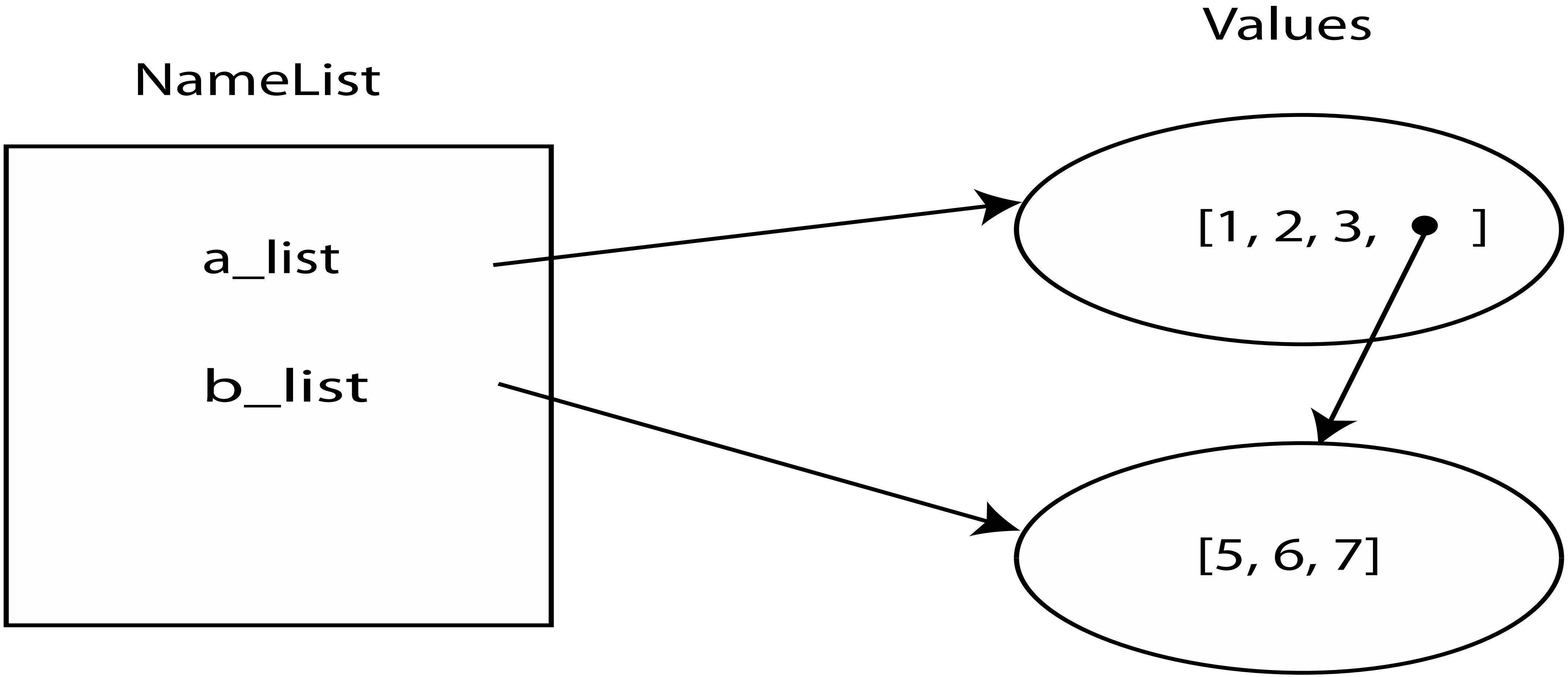


FIGURE 7.9 Lists after append.

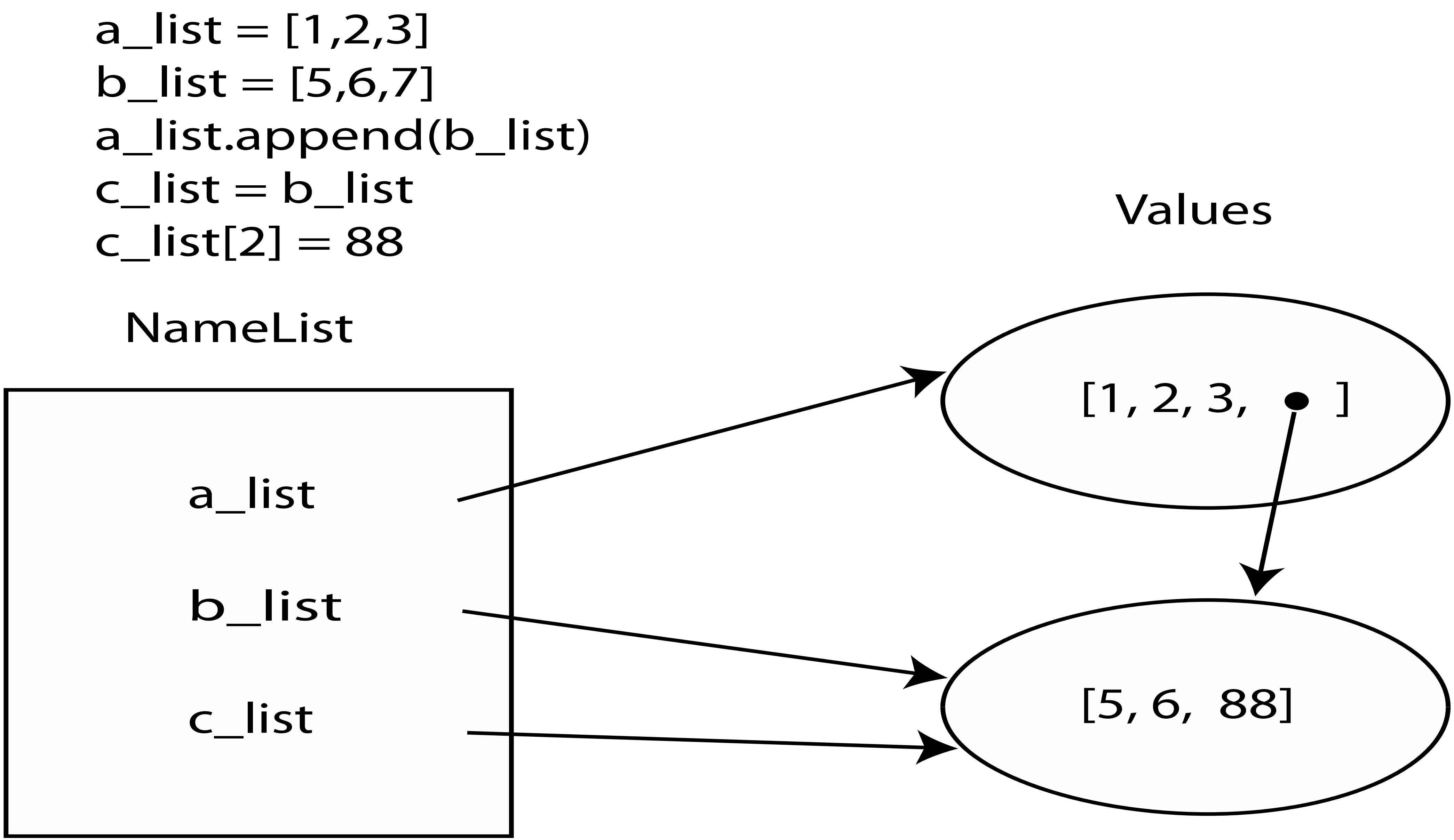


FIGURE 7.10 Final state of copying example.

Tuples

Tuples

- Tuples are simply immutable lists
- They are printed with (,)

```
>>> 10,12          # Python creats a tuple
(10, 12)
>>> tup = 2,3      # assigning a tuple to a variable
>>> tup
(2, 3)
>>> (1)            # not a tuple, a grouping
1
>>> (1,)           # comma makes it a tuple
(1,)
>>> x,y = 'a',3.14159    # from on right, multiple assignments
>>> x
'a'
>>> y
3.14159
>>> x,y            # create a tuple
('a', 3.14159)
```

The question is, Why?

- The real question is, why have an immutable list, a tuple, as a separate type?
- An immutable list gives you a data structure with some integrity, some permanent-ness if you will
- You know you cannot accidentally change one.

Lists and Tuple

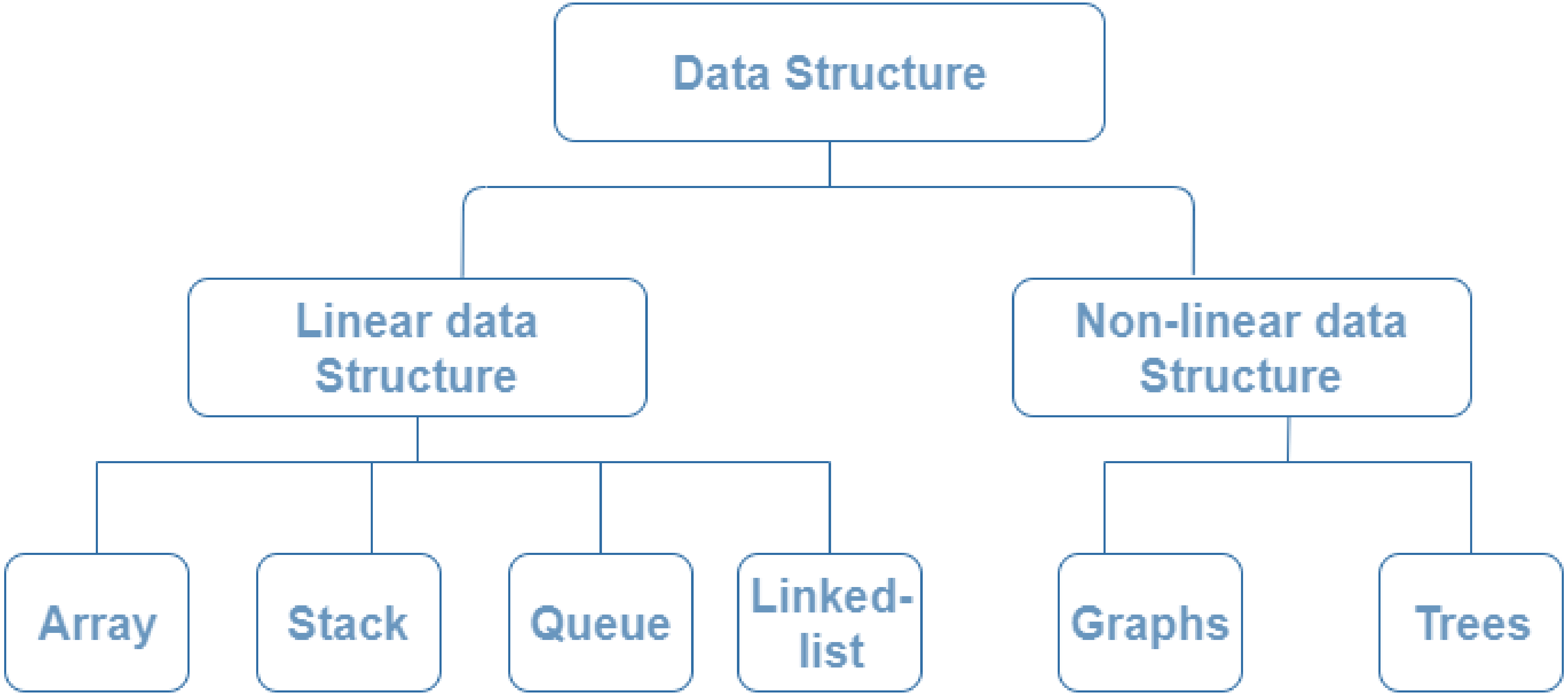
- Everything that works with a list works with a tuple ***except*** methods that modify the tuple
- Thus indexing, slicing, len, print all work as expected
- However, ***none*** of the mutable methods work: `append, extend, del`

Commas make a tuple

For tuples, you can think of a comma as the operator that makes a tuple, where the ( ) simply acts as a grouping:

```
myTuple = 1,2   # creates (1,2)
myTuple = (1,) # creates (1)
myTuple = (1)   # creates 1 not (1)
myTuple = 1,    # creates (1)
```

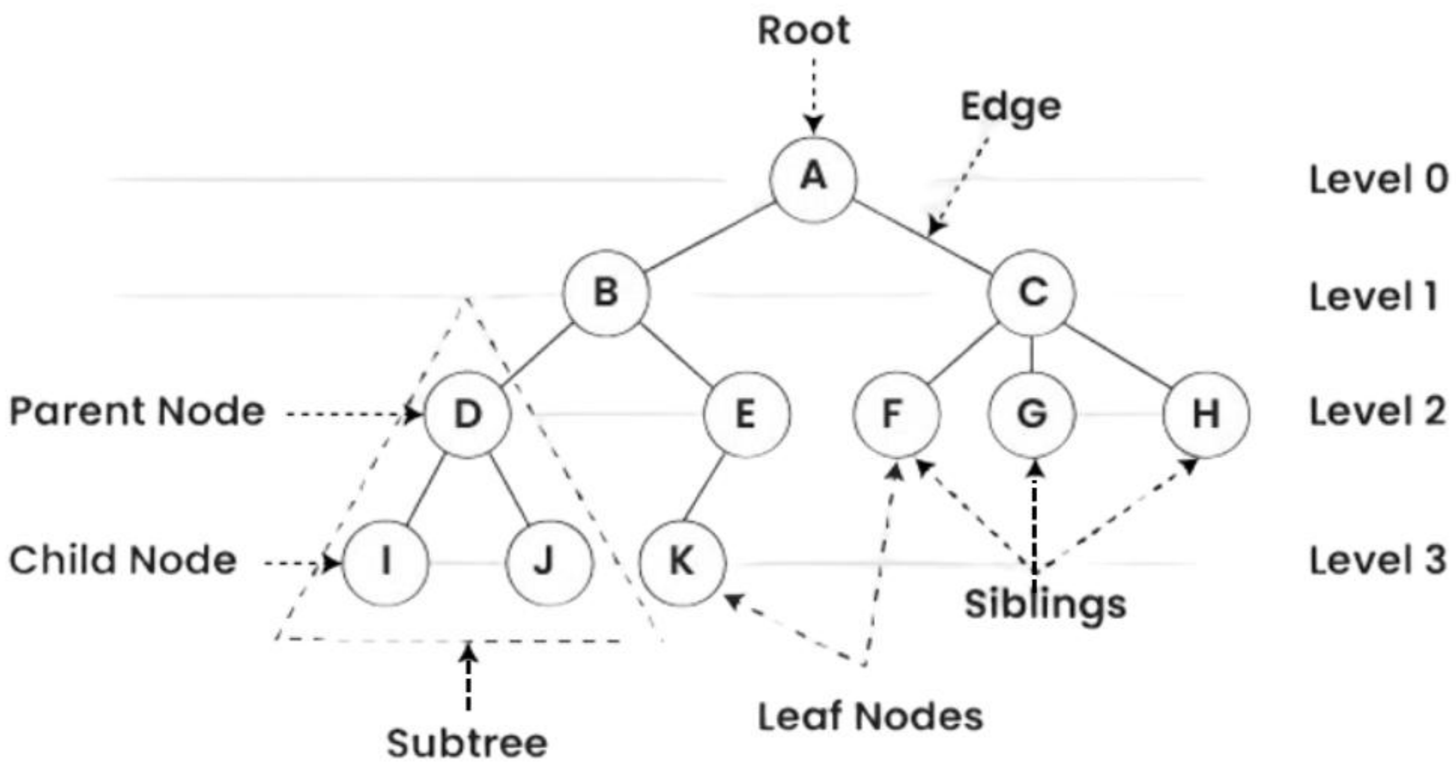
Trees


Data Structure
Linear data Structure
Non-linear data Structure
Array
Stack
Queue
Linked-list
Graphs
Trees

Tree data structure

- Not stored in a sequential manner i.e., they are not stored linearly.
- Arranged on multiple levels or hierarchical structure.

```
class Node:
  def __init__(self, data):
      self.data = data
      self.children = []
```



Types of Tree data structures

- **Binary tree** - Each node can have a **maximum of two children** linked to it.
- **Ternary Tree** - Each node has at **most three child nodes**, usually distinguished as “left”, “mid” and “right”.
- **N-ary Tree** or **Generic Tree** - Each node is a data structure that consists of **records** and a **list of references to its children**. Unlike the linked list, each node stores the address of multiple nodes.

Basic Operations Of Tree Data Structure

- **Create** – create a tree in the data structure.
- **Insert** − Inserts data in a tree.
- **Search** − Searches specific data in a tree to check whether it is present or not.
- **Traversal**

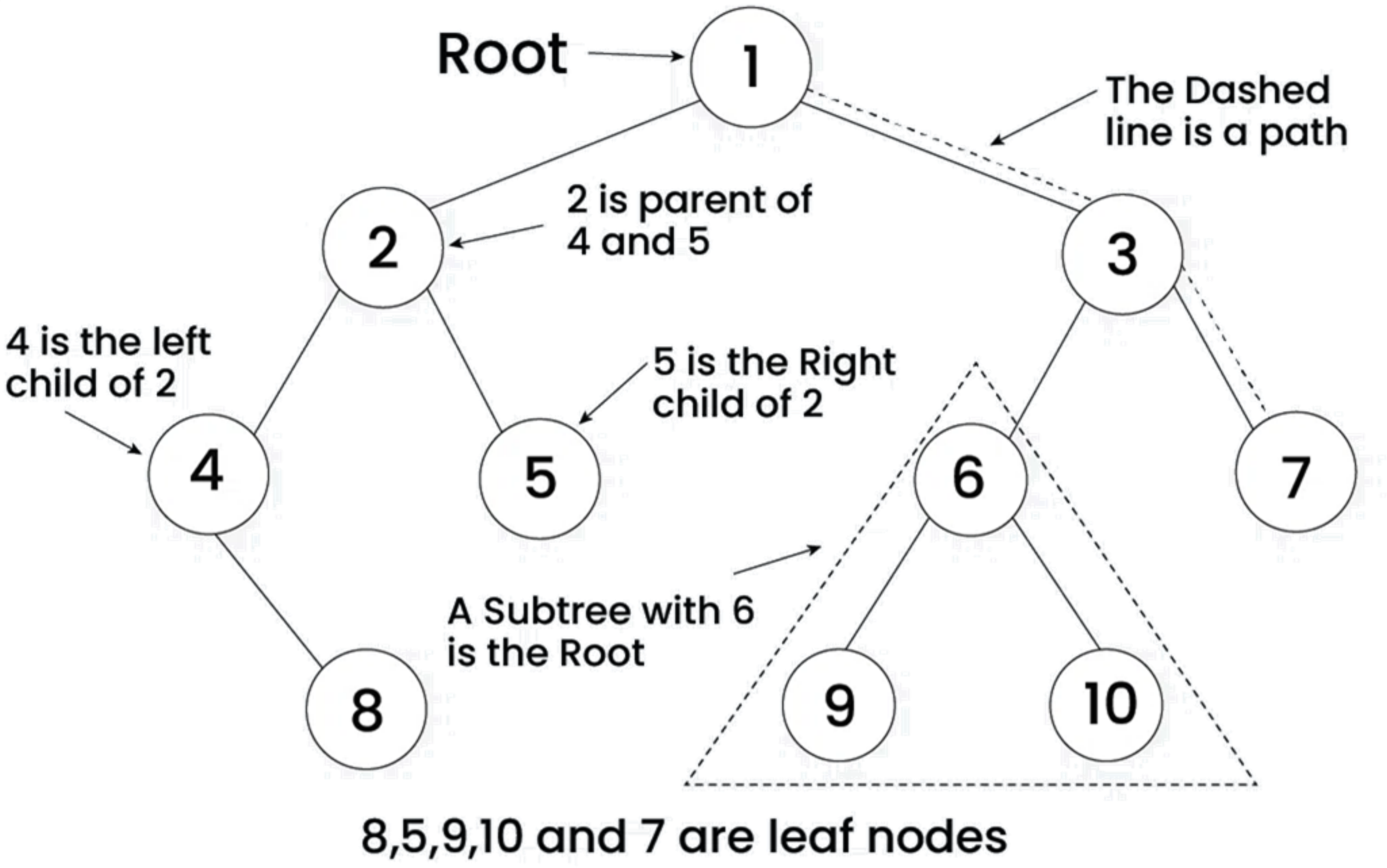
Implementation of Tree Data Structure

See example5-7.py in Pycharm

Binary Tree

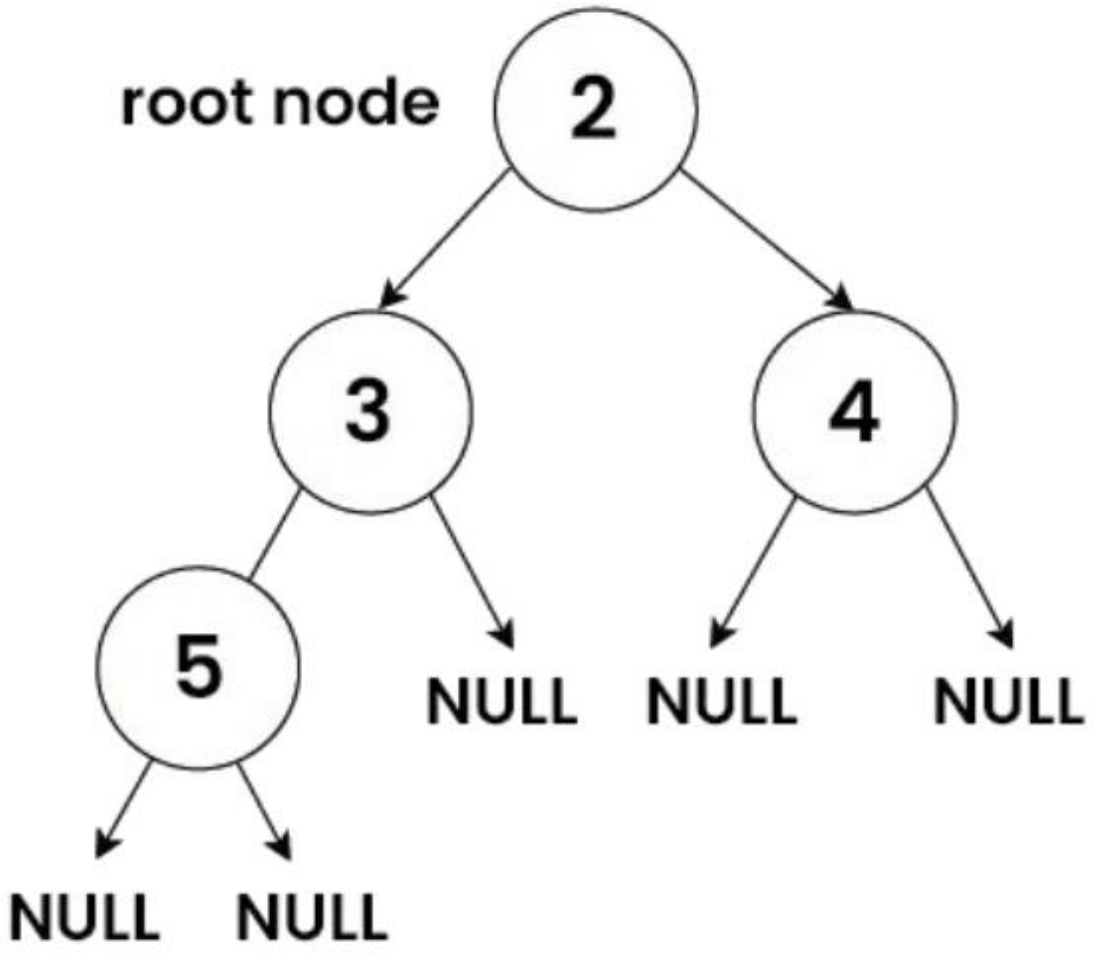
- Non-linear data structure where each node has at most two children.
- They are referred to as the left child and the right child

```python
# 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
```

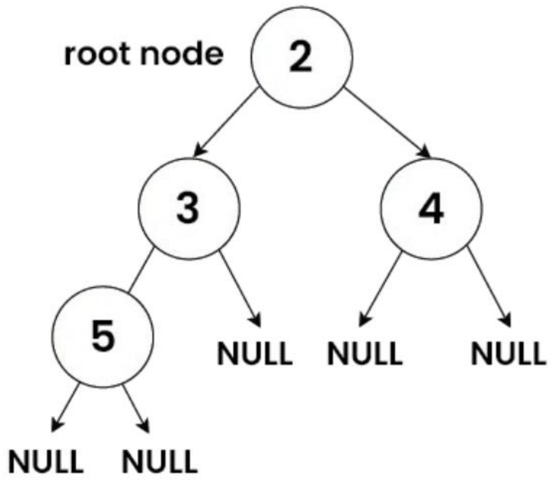


Creating a Binary Tree

- See example5-8.py in Pycharm



Creating a Binary Tree



- See example5-8.py in Pycharm

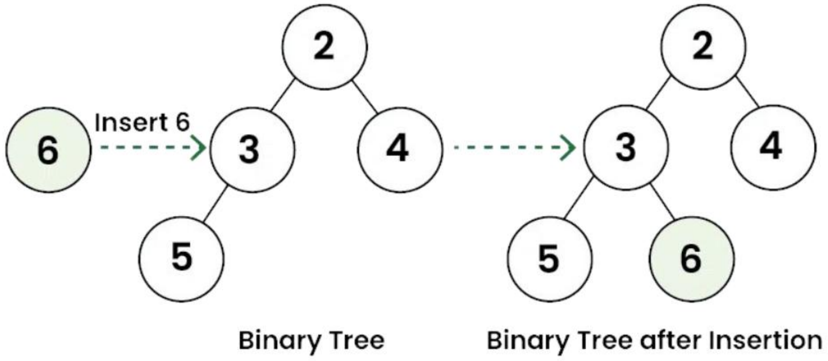
In the code, four nodes—firstNode, secondNode, thirdNode, and fourthNode—are created with values 2, 3, 4, and 5, respectively.

The tree structure is formed as follows:

- firstNode->left = secondNode connects secondNode to the left of firstNode.
- firstNode->right = thirdNode connects thirdNode to the right of firstNode.
- secondNode->left = fourthNode connects fourthNode to the left of secondNode.

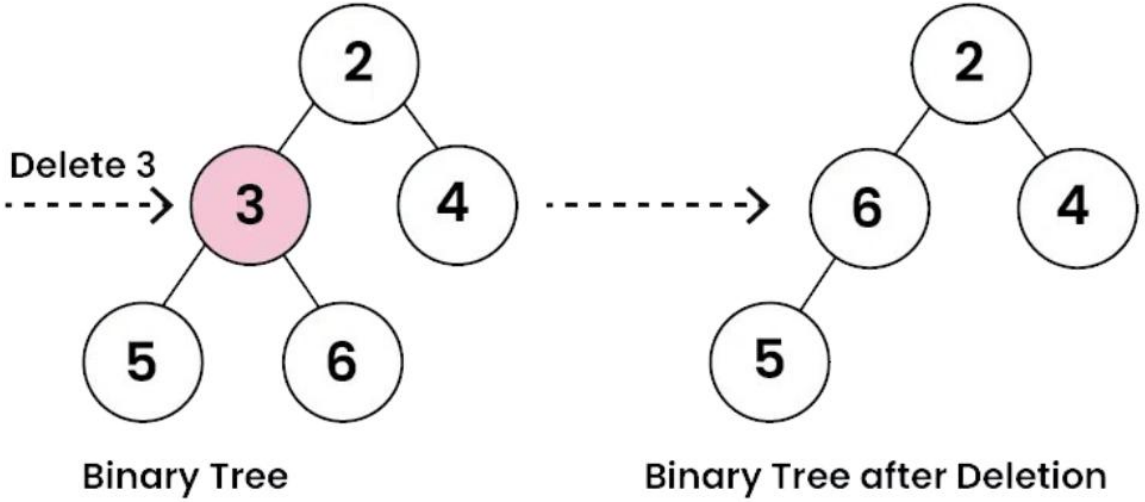
Operations On Binary Tree

- Traversal - visiting all the nodes of the binary tree
  - Depth-First Search (DFS) - start from the **root** and explore the depth nodes first.
  - Breadth-First Search (BFS) - explore all the nodes at the present depth level before moving on to the nodes at the next level.
- Insertion - add a new node into the binary tree



Operations On Binary Tree

- Searching in Binary Tree
- Deletion - visiting all the nodes of the binary tree



Data Structures in General

Organization of data

- We have seen strings, lists and tuples, binary tree so far
- Each is an organization of data that is useful for some things, not as useful for others.

A good data structure

- Efficient with respect to us (some algorithm)
- Efficient with respect to the amount of space used
- Efficient with respect to the time it takes to perform some operations

Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem-solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds.
5. Test your code, often and thoroughly.
6. If it was hard to write, it is probably hard to read. Add a comment.
7. All input is evil, unless proven otherwise.
8. A function should do one thing.
9. Use the right data structure for the job