AIO25 - M1W3

GRID137

June 20, 2025

Contents

List of Figures

List of Tables

Chapter 1

Classes and Objects

Chapter 2

OOP with Python (Custom Pytorch Class)

Chapter 3

Database - SQL(3)

Chapter 4

OOP + Data structure (Graph and Tree)

4.1 Stack and Queue

Stack and Queue definition

Stack and Queue are special uses of List in Python, with specific constraints:

- Stack: only add/remove element from one end (LIFO).
- Queue: Add at one end, remove from the other one (FIFO).

Queues and stacks share a common interface:

- push - adds an element. (For queues, this is also known as enqueue).
- pop - queries/removes an element. (For queues, this is also known as dequeue).

4.1.1 Stack

Stack Visualization

Stacks return elements in the reverse order in which they are stored; that is, the most recent element to be added is returned. We call this kind of data structure last-in-first-out (LIFO).

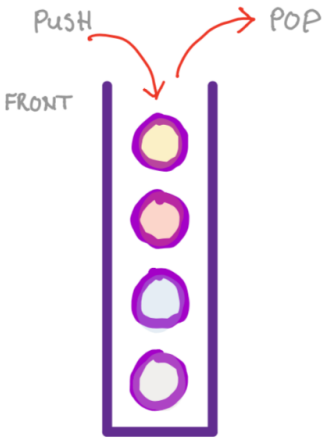

Figure 4.1: Stack visualized

Push/Pop Visualization

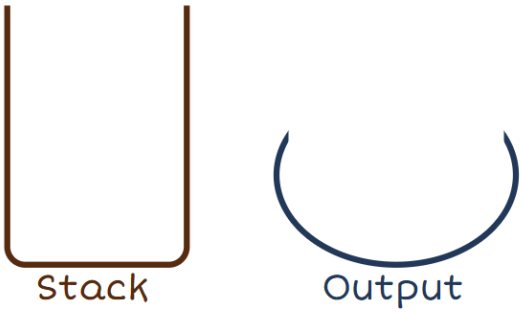

Figure 4.2: Create an Empty stack

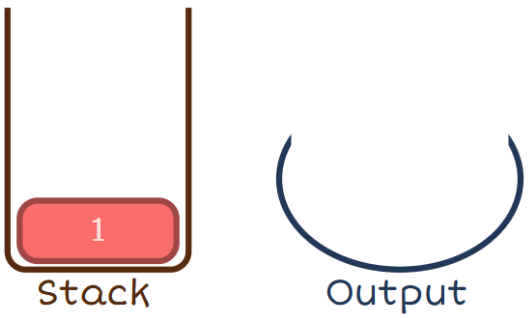

Figure 4.3: Push in the first element

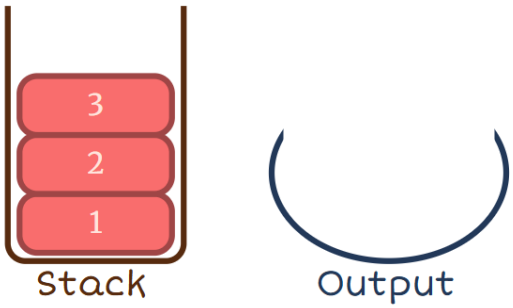

Figure 4.4: Push in the second and third element

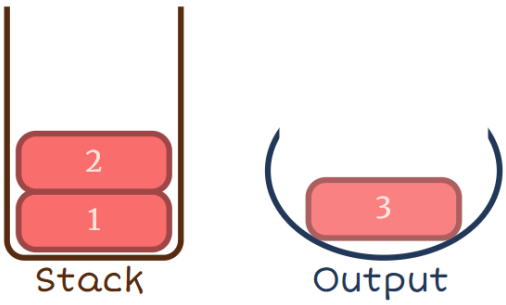

Figure 4.5: Pop out the element at the top of the stack

Coding

```
class MyStack:
  def _init_(self, capacity):
    self._capacity = capacity
    self._stack = []

  def push(self, value):
    if self.is_full():
      print('Do nothing!')
    else:
      self._stack.append(value)

  def pop(self):
    if self.is_empty():
      print('Do nothing')
      return None
    else:
      return self._stack.pop()

  def print(self):
    print(self._stack)

  def is_full(self):
    return len(self._stack) == self._capacity
```

Listing 4.1: Define Stack data structure class

```
stack1 = MyStack(5)
stack1.push(12)
stack1.push(8)
stack1.push(21)
stack1.push(33)
stack1.push(34)
stack1.push(35)
stack1.print()

//Output: Do nothing!
      [12, 8, 21, 33, 34]
```

Listing 4.2: Push and Pop example

4.1.2 Queue

Queue Visualization

Queues return elements in the order in which they were stored. We call this kind of data structure first-in-first-out (FIFO).

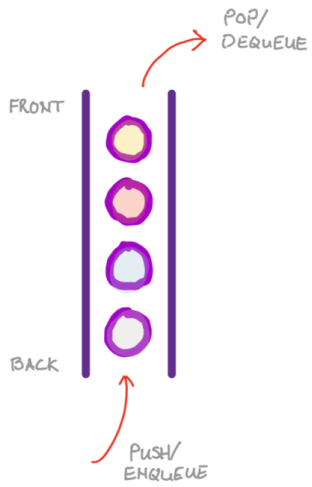

Figure 4.6: Queue visualized

Queue/Dequeue Visualization

Figure 4.7: Create an Empty Queue

Figure 4.8: Enqueue the first element

Figure 4.9: Enqueue the second and third element

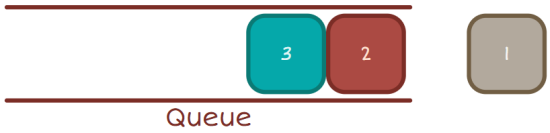

Figure 4.10: Dequeue the element at the front of the queue

Coding

```
class MyQueue:
  def _init_(self, capacity):
    self._capacity = capacity
    self._data = []

  def is_empty(self):
    return len(self._data) == 0

  def dequeue(self):
    if self.is_empty():
      print('Do nothing')
      return None
    else:
      return self._data.pop(0)

  def print(self):
    print(self._data)
```

Listing 4.3: Define Queue data structure class

```
queue = MyQueue(5)
queue.print()
```

```
3
4   queue.enqueue(9)
5   queue.enqueue(5)
6   queue.enqueue(2)
7   queue.enqueue(1)
8   queue.enqueue(0)
9   queue.enqueue(6)
10
11  //Output:
12  Do nothing!
13  [9, 5, 2, 1, 0]
```

Listing 4.4: Example

4.2 Tree

4.2.1 What is a Tree ?

Tree definition

A non-linear data structure where nodes are organized in a hierarchy.

Basic Terminologies

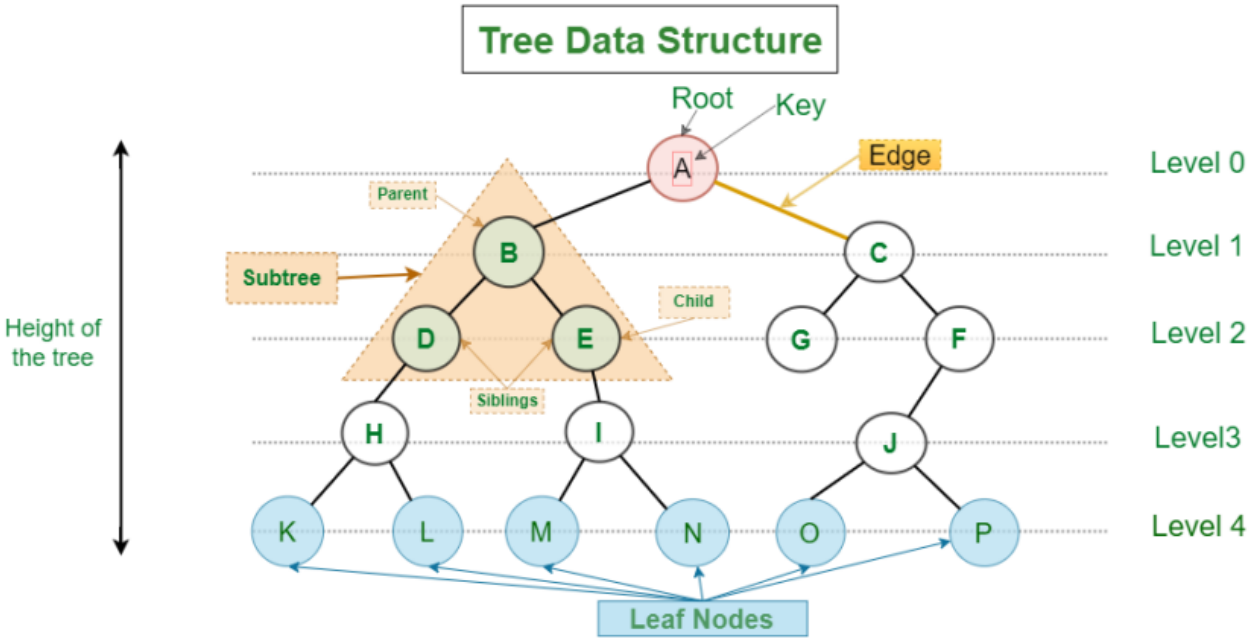

Figure 4.11: Tree terminologies

- Node: A basic unit of a tree containing data and links to its children.
- Root: The topmost node of the tree. There is only one root in a tree.

- Parent: A node that has one or more child nodes.
- Child: A node that descends from another node (its parent).
- Leaf: A node that has no children.
- Internal Node: A node that has at least one child.
- Edge: A link between a parent and a child node.
- Path: A sequence of nodes and edges connecting a node with a descendant.
- Subtree: A tree formed by a node and its descendants.
- Level (or Depth): The number of edges from the root to the node (root is level 0).
- Height of a Node: The number of edges on the longest path from the node to a leaf.
- Height of tree: The height of the root node.
- Degree of Node: The number of children of a node.

4.2.2 Tree Implementation

```
class TreeNode:
  def __init__(self, data):
    self.data = data
    self.children = []
    self.parent = None

  def add_child(self, child):
    child.parent = self
    self.children.append(child)

  def get_level(self): #to get level of each node
    level =0
    p = self.parent
    while p:
      level += 1
      p = p.parent

    return level
  def print_tree(self):
    space = ' ' * self.get_level() * 3
    prefix = space + '|__' if self.parent else ''
    print(prefix + self.data)#add prefix
    if self.children:
      for child in self.children:
        child.print_tree()
```

Listing 4.5: Defining Tree Node

```
def create_tree():
    a_node = TreeNode("A")
    b_node = TreeNode("B")
    c_node = TreeNode("C")
    d_node = TreeNode("D")
    e_node = TreeNode("E")
    f_node = TreeNode("F")
    g_node = TreeNode("G")

    a_node.add_child(b_node)
    a_node.add_child(c_node)

    b_node.add_child(d_node)
    b_node.add_child(e_node)

    c_node.add_child(f_node)
    c_node.add_child(g_node)
    return a_node

    tree = create_tree()
    tree.print_tree()
```

Listing 4.6: Example of creating a Tree from Nodes

Expected Result:

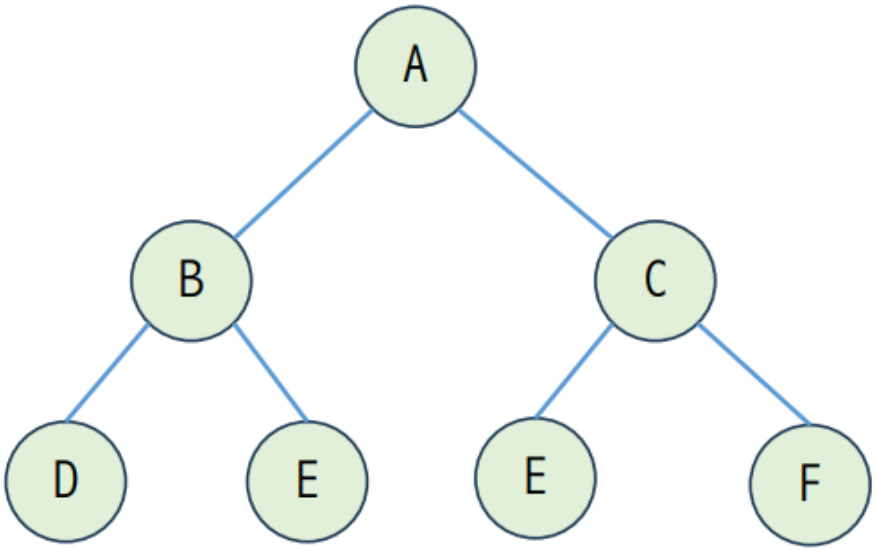

Figure 4.12: Expected Tree

4.2.3 Tree applications

Storing naturally hierarchical data

Data is organised hierarchically using trees.

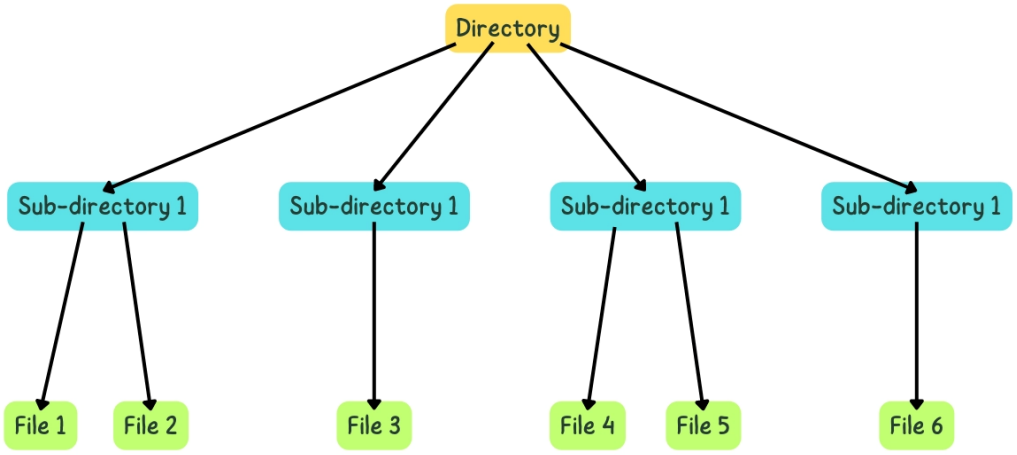

Figure 4.13: Computer's file system

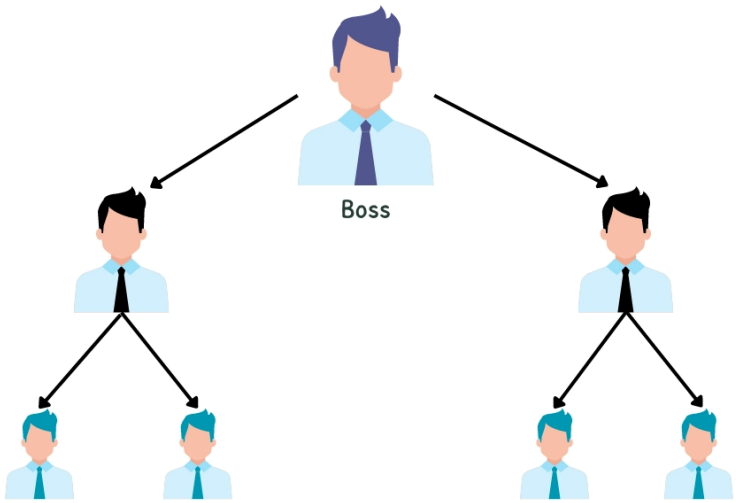

Figure 4.14: Organisational structure

Decision trees

In machine learning and data mining, decision trees are represented by tree data structures. Each node in a decision tree represents a choice or a feature, and the child nodes represent the feature's possible outcomes or values.

Example 1: A student set a rule for himself about whether to study or play. If there are more than two days left until the day, he will go out. If there are less than two days left and there is a football match tonight, he will sing at his friend's house and watch the match together. He will only study in other cases.

The yellow ellipse represents the decision that needs to be made. This decision depends on the answers to the questions in the gray rectangles. Based on the answers, the final decision is given in the green (play) and red (study) circles.

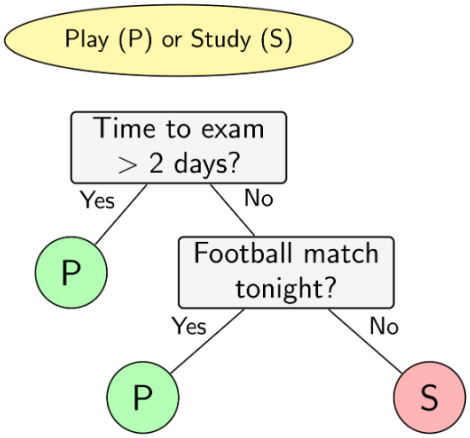

Figure 4.15: Example of decision making using a Tree

Classification using Decision tree: In the following example, the task is to find a simple boundary that separates these two classes. In other words, this is a classification problem, we need to build a classifier to decide which class a new data point belongs to. If a point has the first component, x1, less than the threshold, t1, we immediately decide that it belongs to the green class. In addition, if the second component, x2, is greater than the threshold, t2, we decide that it also belongs to the green class. Next, if the first component, x1, is greater than threshold, t3, we decide that it belongs to the green class. Points that do not satisfy the above conditions are classified as red class.

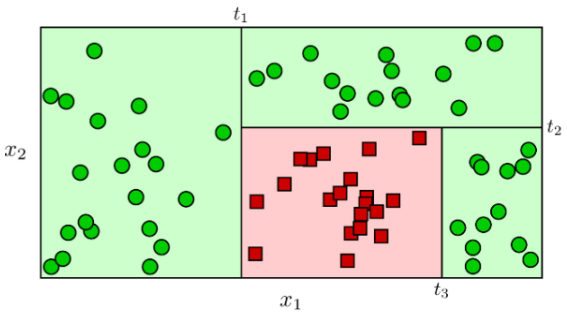

Figure 4.16: (a)

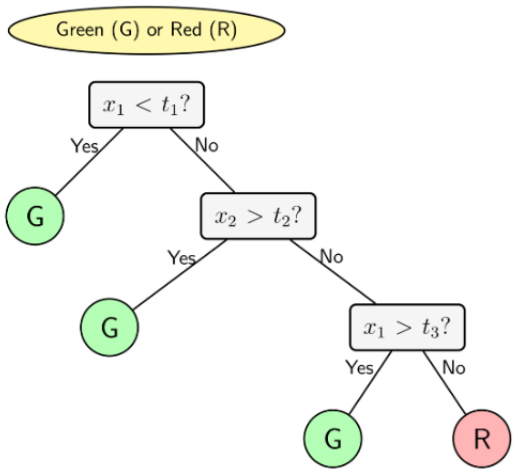

Figure 4.17: (b)

4.2.4 Types of Trees

Binary Tree

A binary tree is a tree data structure where each node has at most two children. These two children are usually referred to as the left child and right child. It is widely used in applications such as binary search trees and heaps.

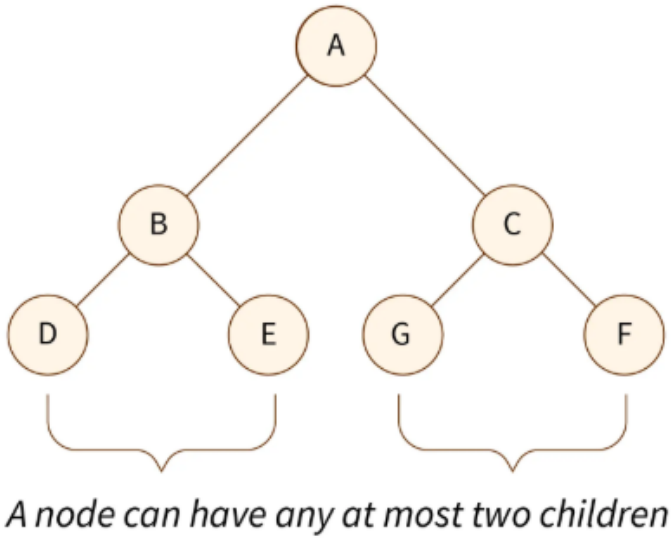

Figure 4.18: Binary Tree

Ternary Tree

It is a type of tree data structure where each node can have a maximum of three child nodes., commonly referred to as "left", "mid", and "right".

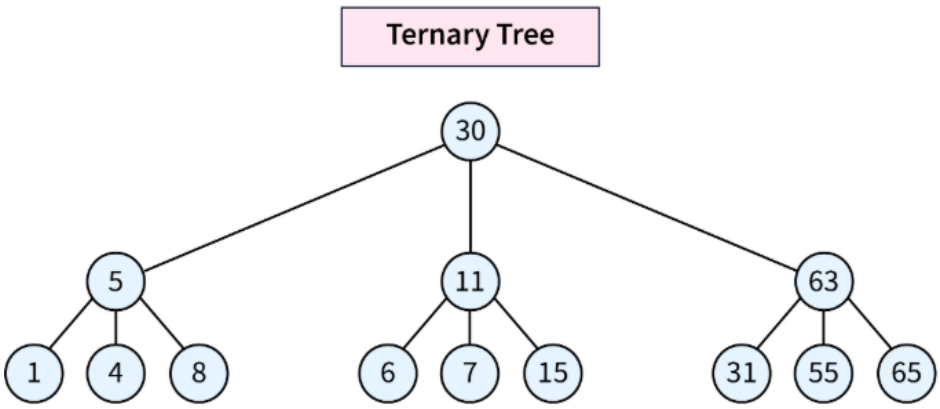

Figure 4.19: Ternary Tree

N-ary Tree

An N-ary Tree, also called a Generic Tree, is a type of tree data structure. In an N-ary tree, each node holds data records and references to its children. It prohibits duplicate references among its children.
Key features of an N-ary tree:

- Each node can have multiple children.
- The exact number of children for each node is not predetermined and may vary.

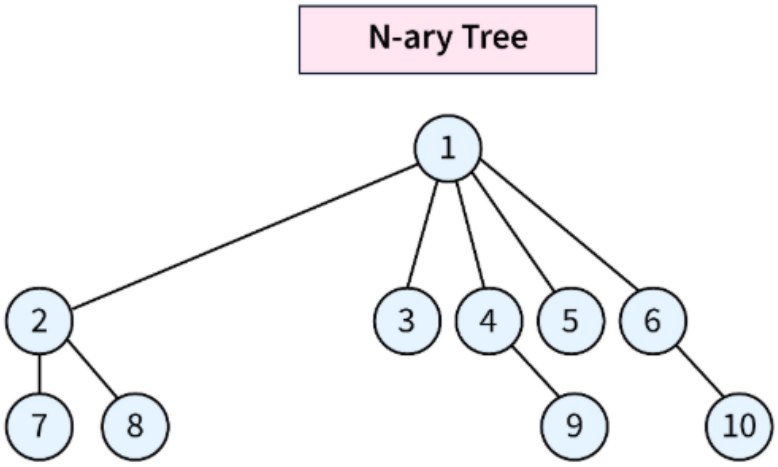

Figure 4.20: N-ary Tree

4.2.5 Algorithms on Trees

Traversal of Binary Tree

Traversal of Binary Tree

Traversal on a tree refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once, in a systematic way.

- Breadth first search: Explores all the nodes in a tree at the current depth before moving on to the vertices at the next depth level.
- Depth first search: The algorithm starts at the root node and explores as far as possible along each branch before backtracking

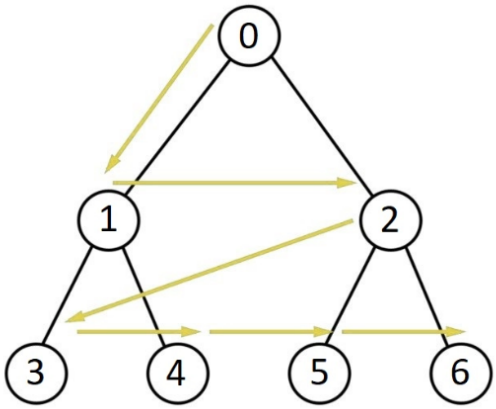

Figure 4.21: BFS

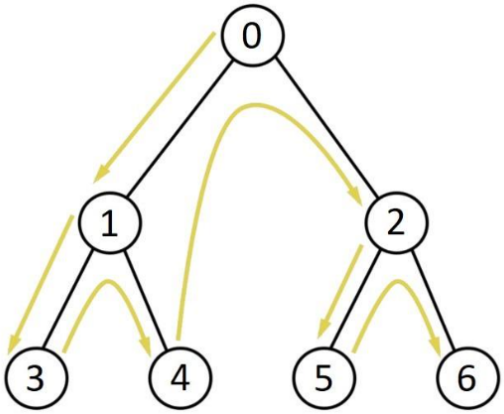

Figure 4.22: DFS

Chapter 5

Unix and Docker