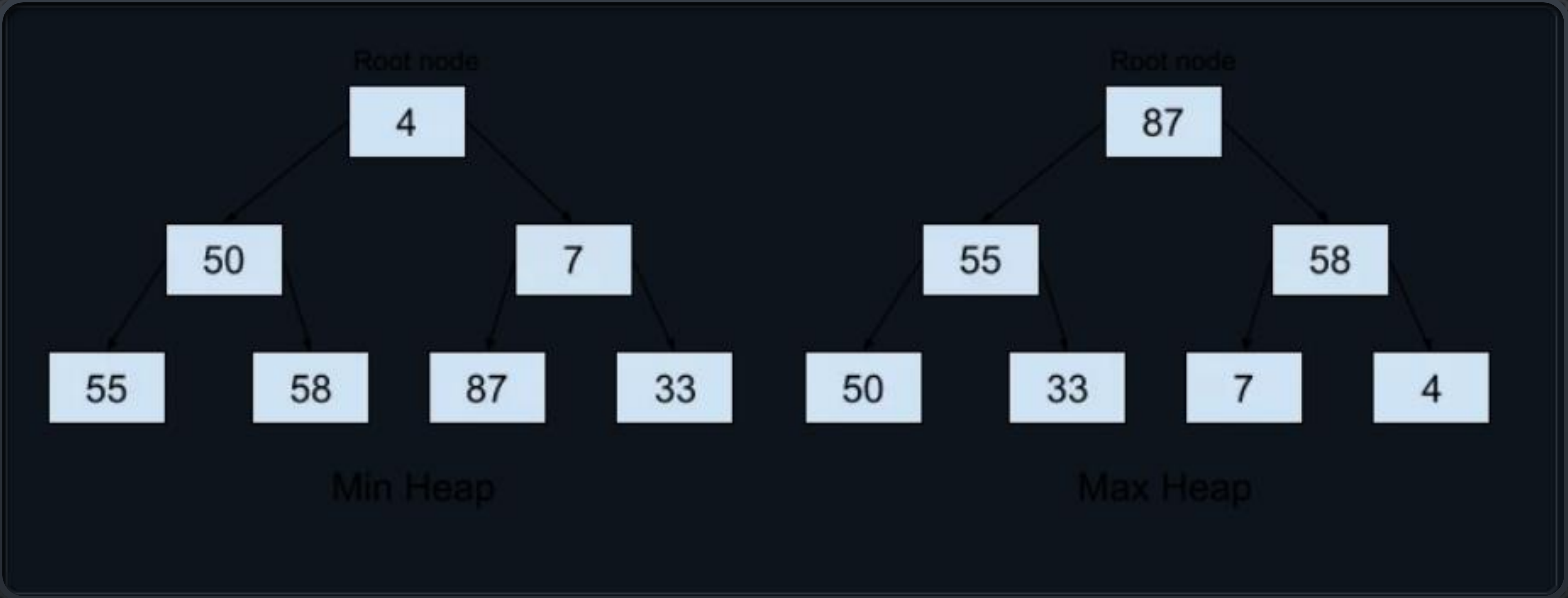
HEAPS AND GRAPHS

INTRODUCTION TO HEAPS

- HEAPS IN PYTHON ARE COMPLETE BINARY TREES IN WHICH EACH NODE IS EITHER SMALLER THAN EQUAL TO OR GREATER THAN EQUAL TO ALL ITS CHILDREN (SMALLER OR GREATER DEPENDING ON WHETHER IT IS A MAX-HEAP OR A MIN-HEAP).
- HENCE THE ROOT NODE OF A HEAP IS EITHER THE SMALLEST OR THE GREATEST ELEMENT. **THE HEAP DATA STRUCTURE IS GENERALLY USED TO REPRESENT A PRIORITY QUEUE.**
- HEAPS IN PYTHON, BY DEFAULT, ARE MIN-HEAPS.

TWO TYPES

- **Min-Heap**: Min heap is the heap in which all nodes are lesser than their children. The root contains the lowest value in a min-heap.
- **Max-Heap**: Max heap is the heap in which all nodes are greater than their children. The root contains the highest value in a max-heap.



VISUAL REPRESENTATION

Representation

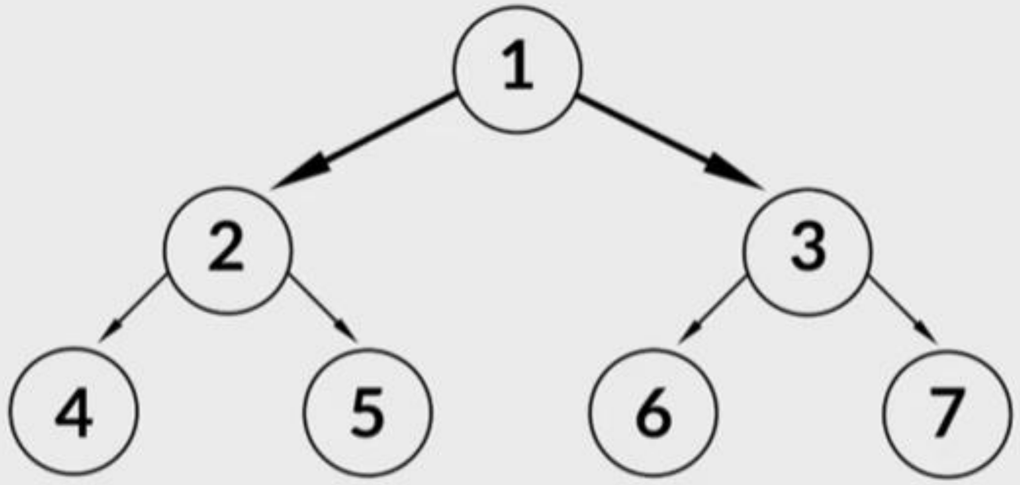
- The heap data structure is theoretically in the form of a binary tree.
- but due to its property of completeness the heap is stored in the form of an array in the memory.
- The first element contains the minimum element (in the case of min-heap).

Continued

- The heap, which is in the form of a tree, is stored in the array, and its elements are indexed in the following manner:
- The root element will be at the 0th position of the array, that is, Heap[0].
- For any other node, say Heap[i], we have the following:
  - The parent node is given by : Heap[(i -1) / 2].
  - The left child node is given by : Heap[(2 * i) + 1]
  - The right child node is given by : Heap[(2 * i) + 2]

REPRESENTING OUR HEAP

Picture in picture



0	1	2	3	4	5	6
1	2	3	4	5	6	7

PARENT INDEX

$$\left\lfloor \frac{(\text{INDEX} - 1)}{2} \right\rfloor$$

LEFT CHILD INDEX

2 * INDEX + 1

RIGHT CHILD INDEX

2 * INDEX + 2

IMPLEMENTATION USING HEAPQ

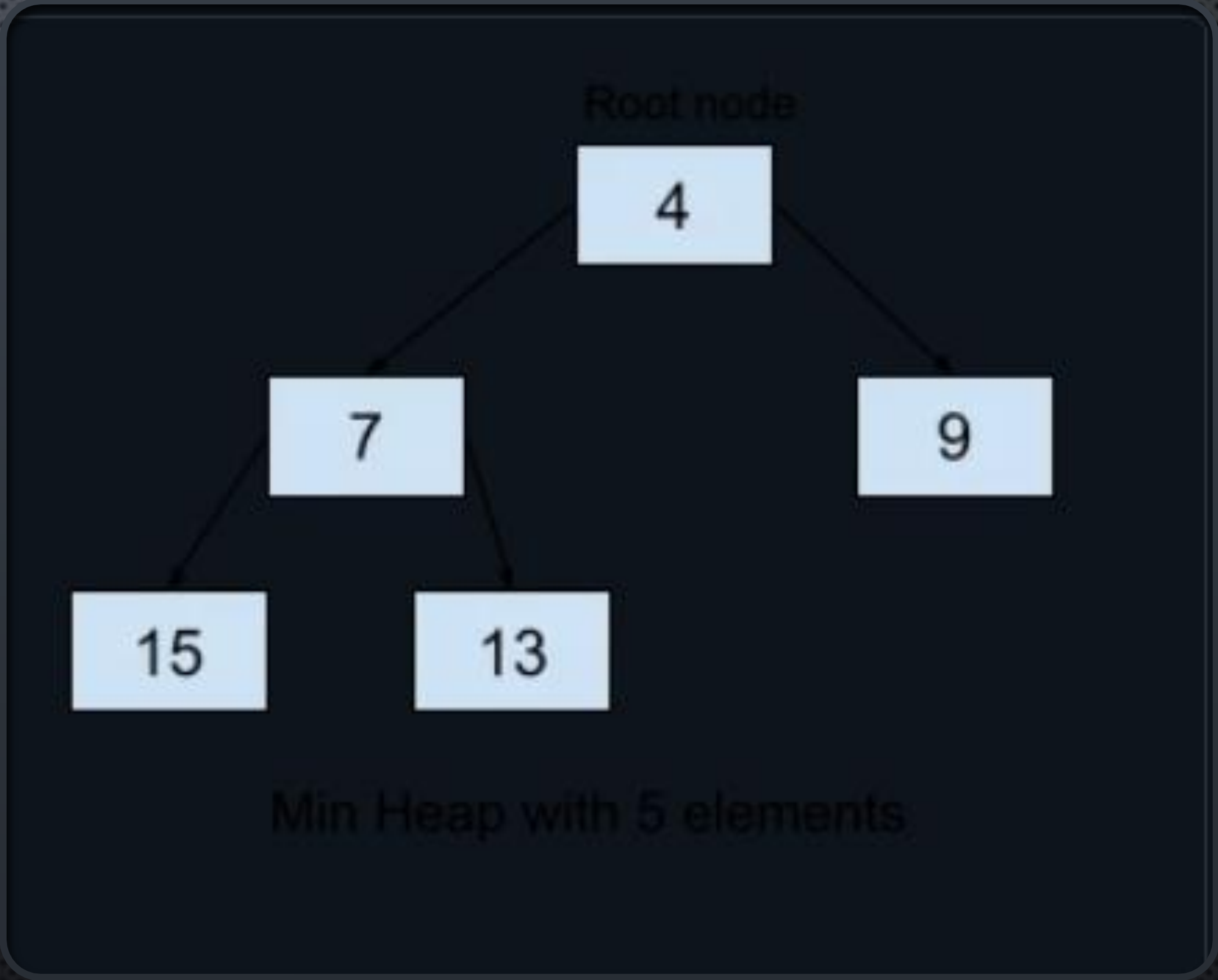
- Python has the "heapq" module for the implementation of Heap Queue (or simply heap).
- It contains the functionality that the smallest element will always be at the top and will be popped when called the pop function.
- Whenever elements are either pushed or popped, heap property will be maintained, and heap[0] will always give us the smallest element.

CONTINUED

- THE MODULE CONTAINS THE FOLLOWING MAJOR FUNCTIONS FOR HEAP:
- **HEAPIFY**(ITERABLE_NAME): WE USE THIS FUNCTION TO PASS ANY ITERABLE (FOR EXAMPLE A LIST) AND IT CONVERTS IT INTO A HEAP DATA STRUCTURE.
- **HEAPPUSH**(HEAP_NAME, ELEMENT_TO_BE_INSERTED): AS THE NAME SUGGESTS, THIS FUNCTION PUSHES/ ADDS AN ELEMENT TO THE HEAP. WE NEED TO PASS THE HEAP NAME AND ELEMENT TO BE INSERTED AS A PARAMETER. THE FUNCTION TAKES CARE OF REARRANGING THE HEAP (IF NEED BE) TO SATISFY THE HEAP PROPERTY.
- **HEAPPOP**(HEAP_NAME): AS THE NAME SUGGESTS, THIS FUNCTION POPS/REMOVES AN ELEMENT FROM THE HEAP PASSED AS A PARAMETER. THE FUNCTION TAKES CARE OF REARRANGING THE HEAP (IF NEED BE) TO SATISFY THE HEAP PROPERTY.

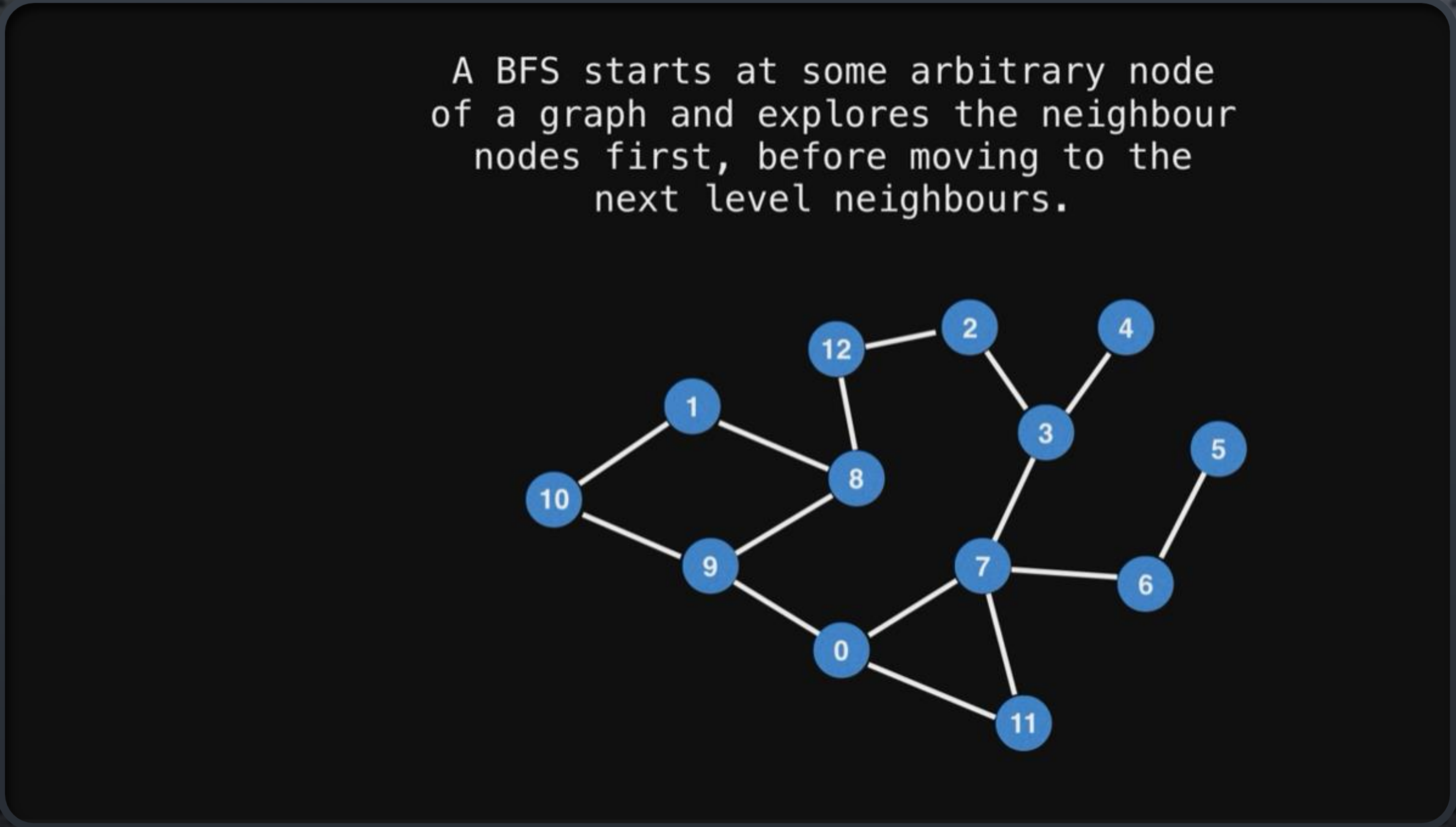
PRACTICAL IMPLEMENTATION

- WE WILL IMPLEMENT A MIN-HEAP IN PYTHON. WE USE A `LIST` `[15, 7, 9, 4, 13]` IN THE CODE AND CONVERT IT TO A HEAP USING THE `HEAPIFY` FUNCTION. THE HEAP MADE WOULD LOOK LIKE THIS:



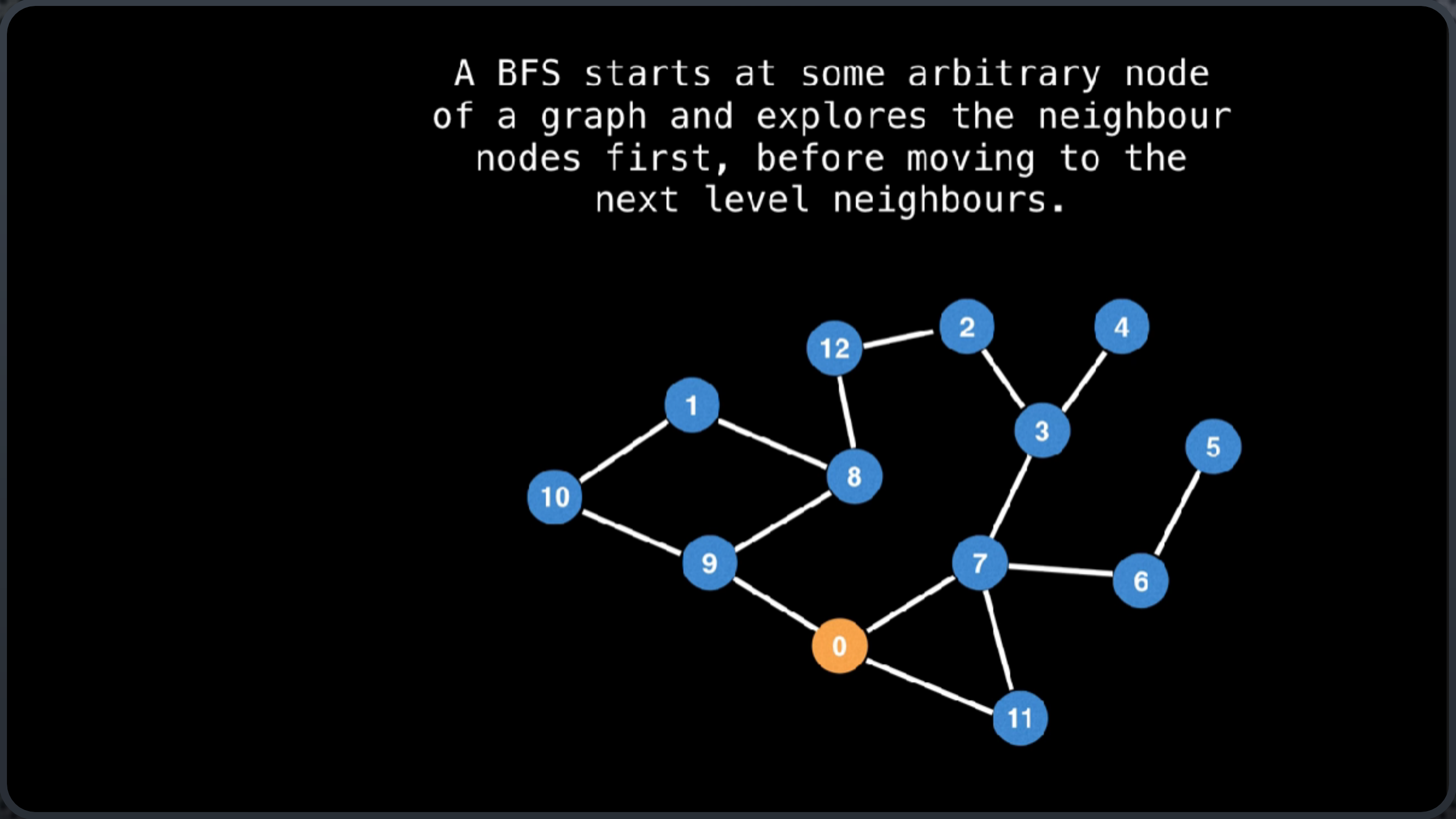
CODE

```
IMPORT HEAPQ
LIS = [15, 7, 9, 4, 13]
HEAPQ.HEAPIFY(LIS)
PRINT ("THE HEAP LOOKS LIKE: ")
PRINT(LIS)
PRINT ("THE POPPED ITEM USING HEAPPUSHPOP() IS : ",END="")
PRINT (HEAPQ.HEAPPOP(LIS))
PRINT ("AFTER POPPING, THE HEAP LOOKS LIKE: ")
PRINT(LIS)
PRINT ("AFTER PUSHING 2, THE HEAP LOOKS LIKE: ")
HEAPQ.HEAPPUSH(LIS, 2)
PRINT(LIS)
```



A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

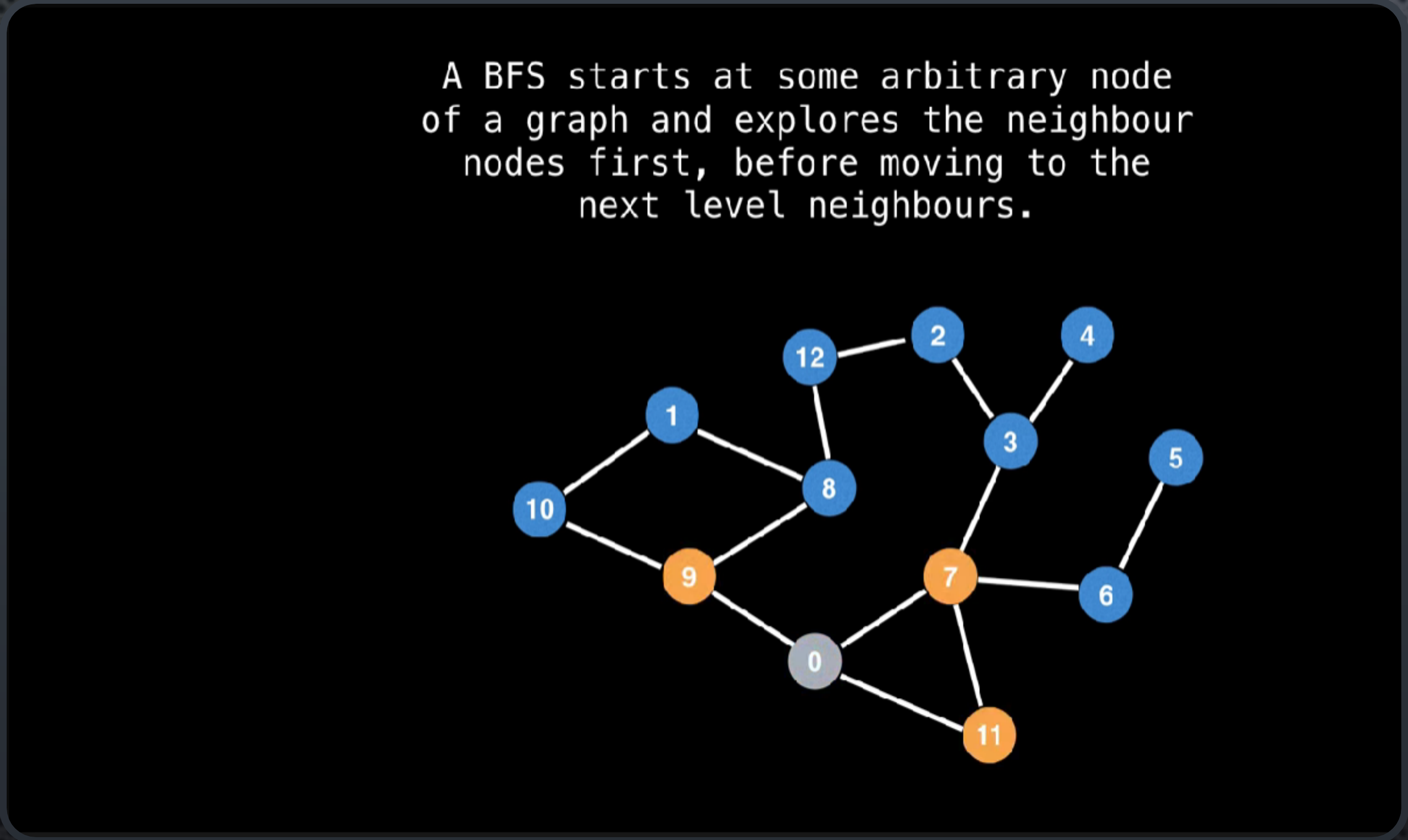
BFS



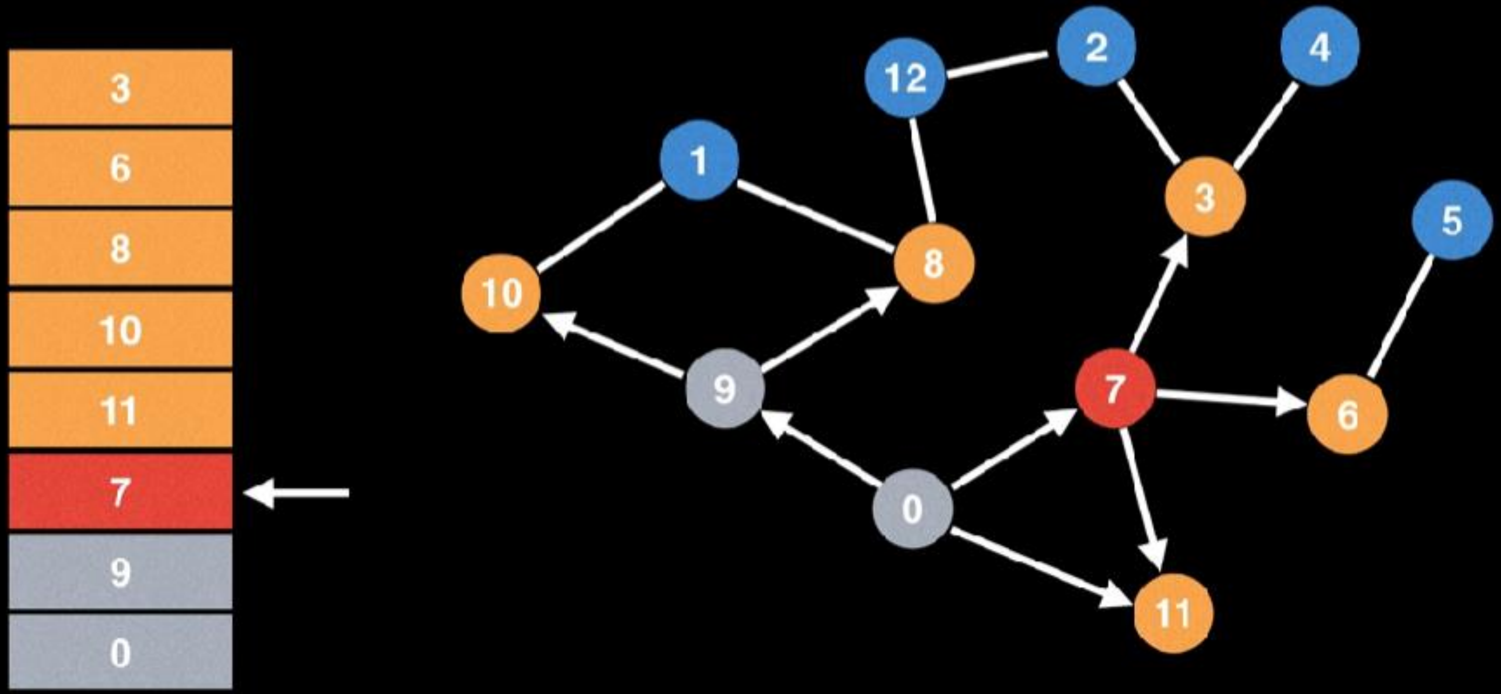
A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

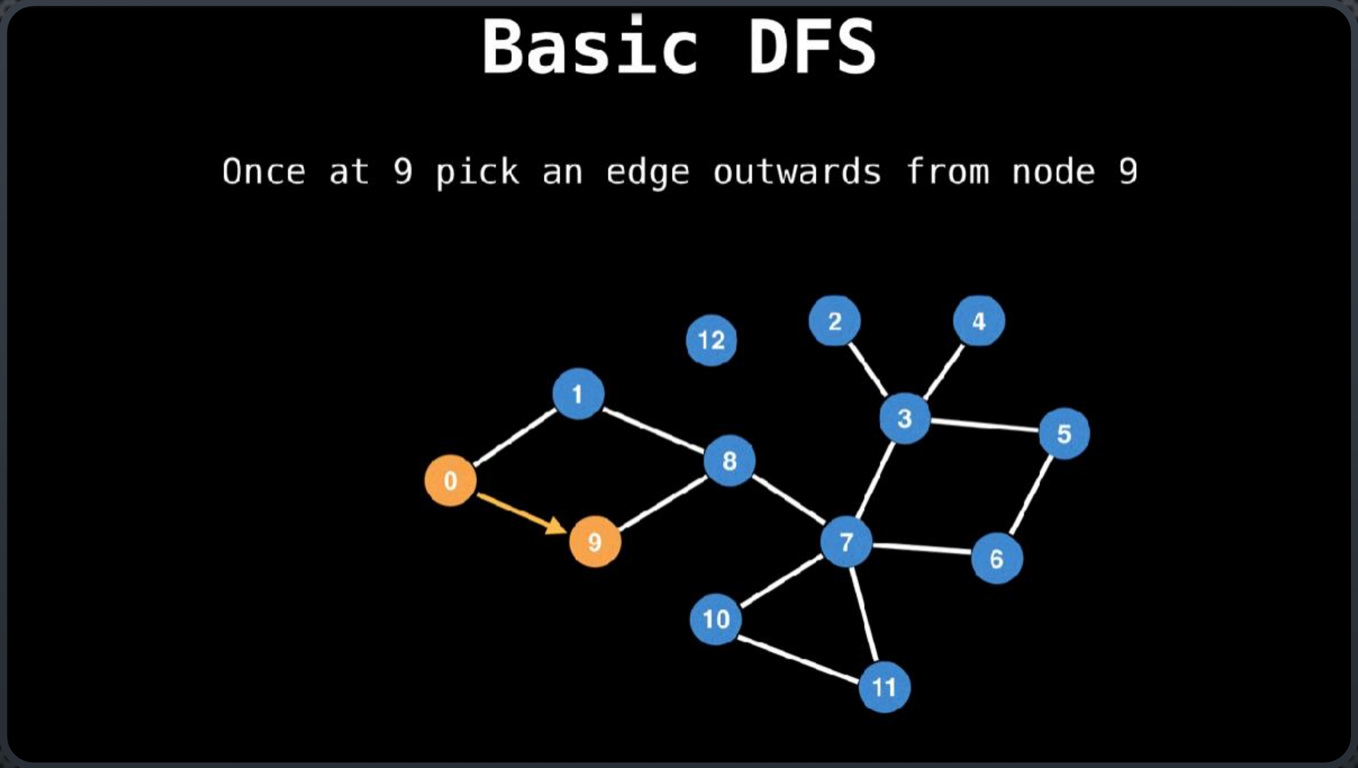
COTINUED

COTINUED


A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

A BFS starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

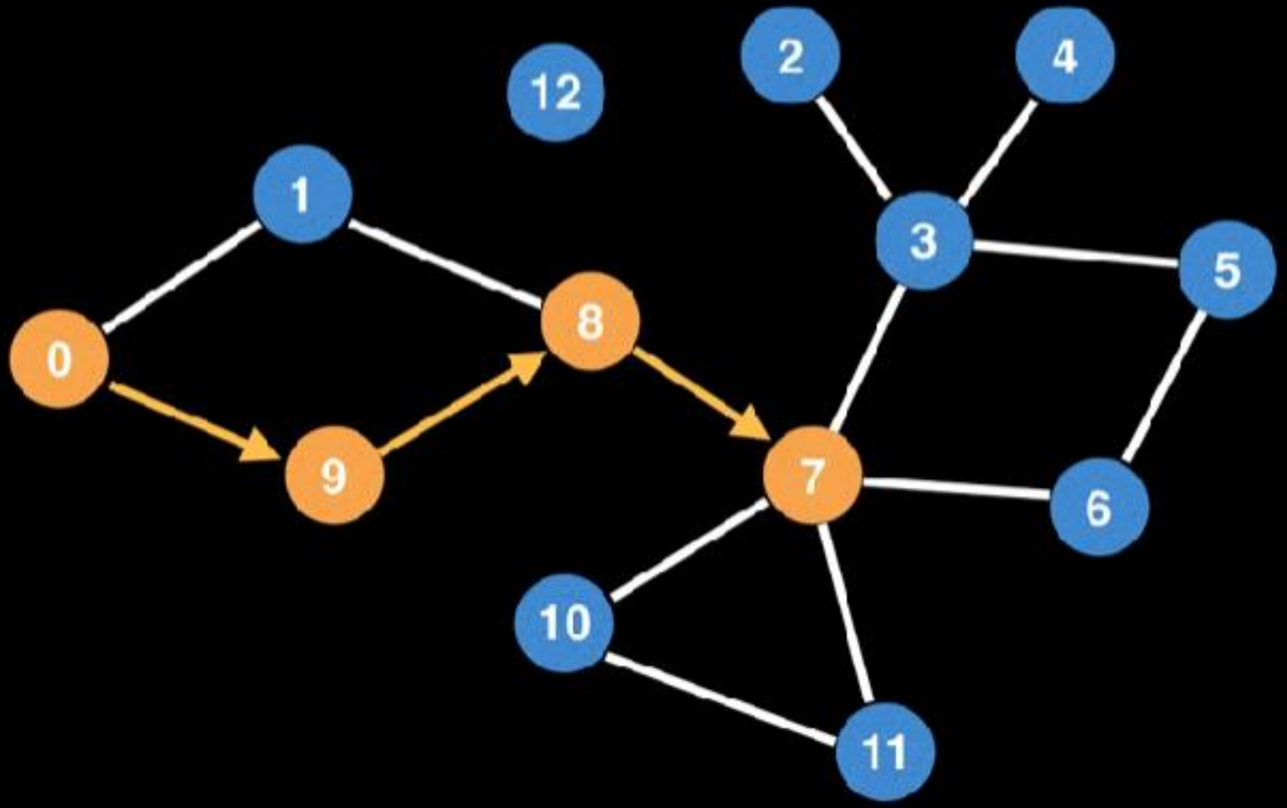


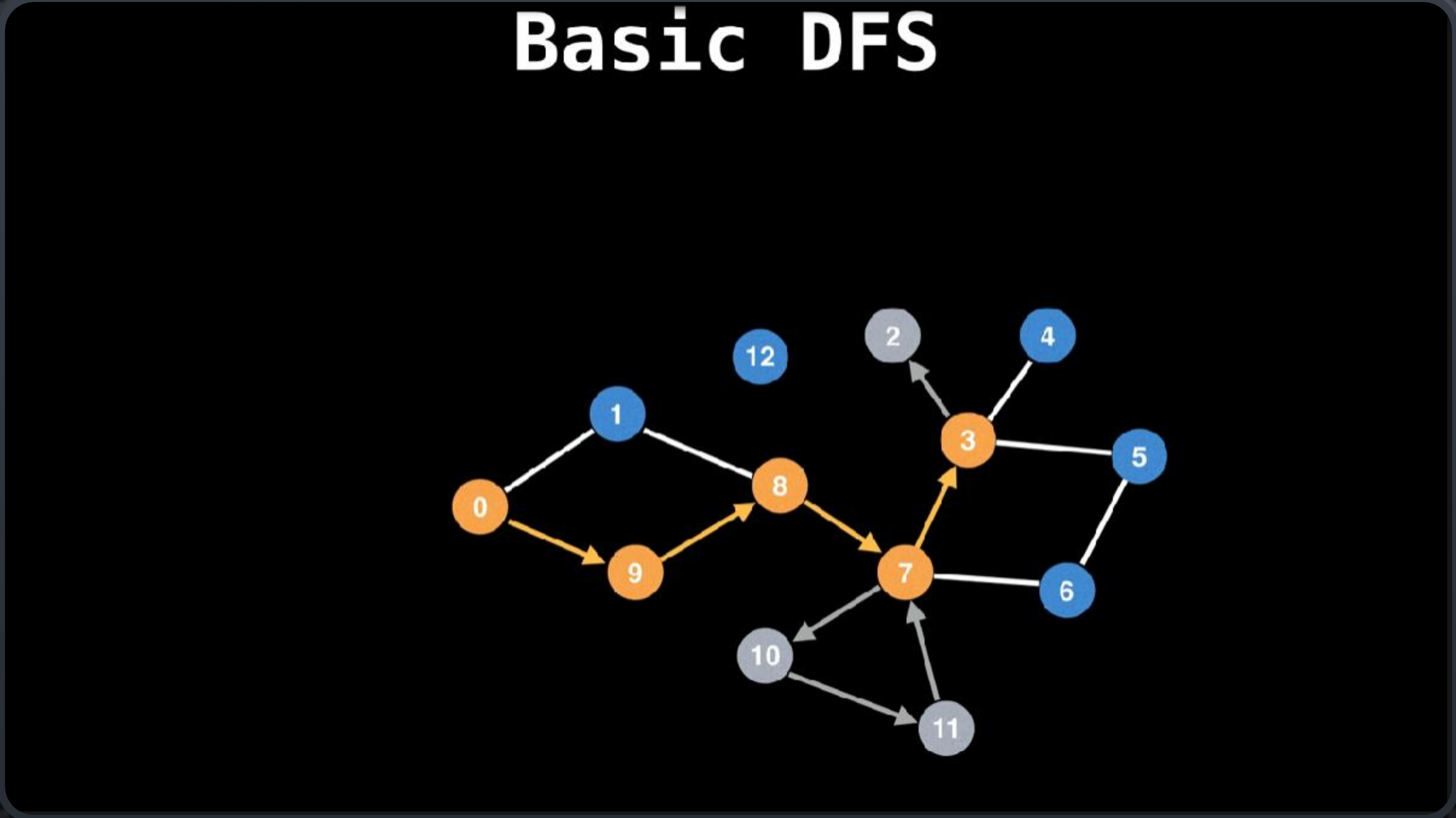

Basic DFS

Once at 9 pick an edge outwards from node 9

DFS

Basic DFS





BACKTRACK

PROS AND CONS

- **Advantages of Graph:**
- By using graphs we can easily find the shortest path, neighbors of the nodes, and many more.
- Graphs are used to implement algorithms like DFS and BFS.
- It is used to find minimum spanning tree which has many practical applications.
- It helps in organizing data.
- Because of its non-linear structure, helps in understanding complex problems and their visualization.
- **Disadvantages of Graph:**
- Graphs use lots of pointers which can be complex to handle.
- It can have large memory complexity.
- If the graph is represented with an adjacency matrix then it does not allow parallel edges and multiplication of the graph is also difficult.