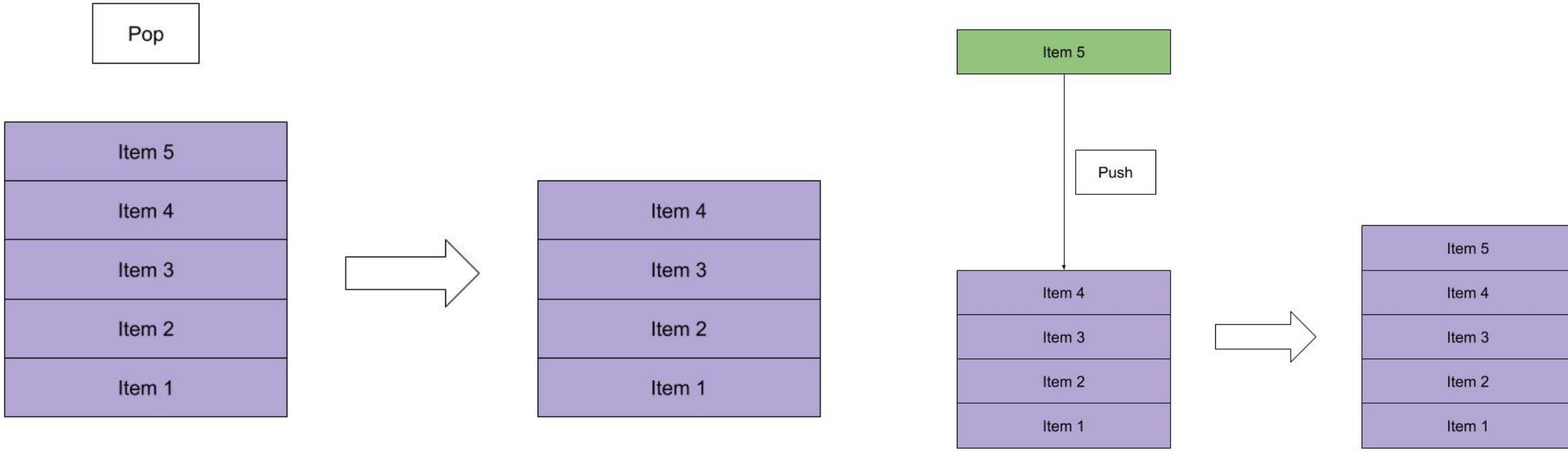
STACK, QUEUES AND BINARY TREES

Introduction

- *Stacks* and *Queues* are some of the earliest data structures defined in computer science.
- Simple to learn and easy to implement, their uses are common.
- It's common for Stacks and Queues to be implemented with an Array or Linked List. We'll be relying on the List data structure to accommodate both Stacks and Queues.

Stack

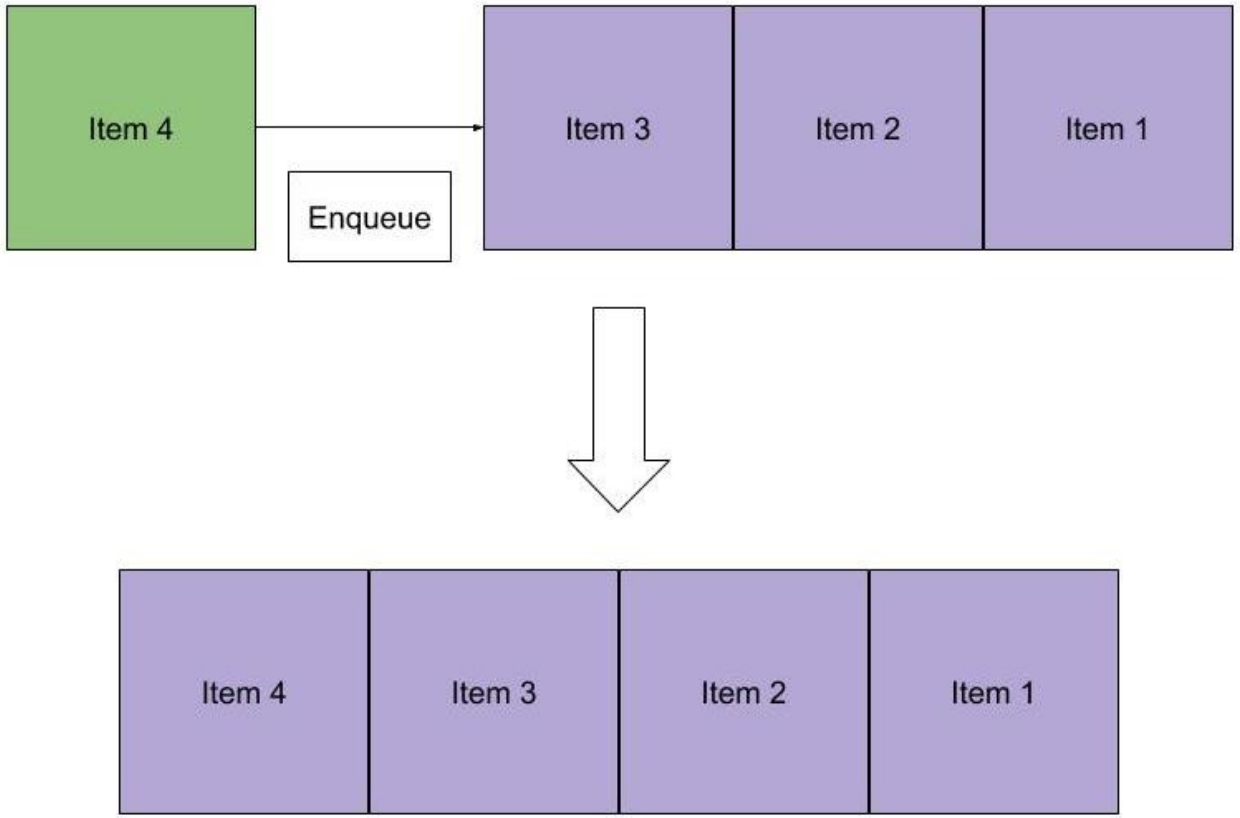
- Stacks, like the name suggests, follow the **Last-in-First-Out** (LIFO) principle. As if stacking coins one on top of the other, the last coin we put on the top is the one that is the first to be removed from the stack later.
- To implement a stack, therefore, we need two simple operations:
- push – adds an element to the top of the stack
- pop – removes the element at the top of the stack

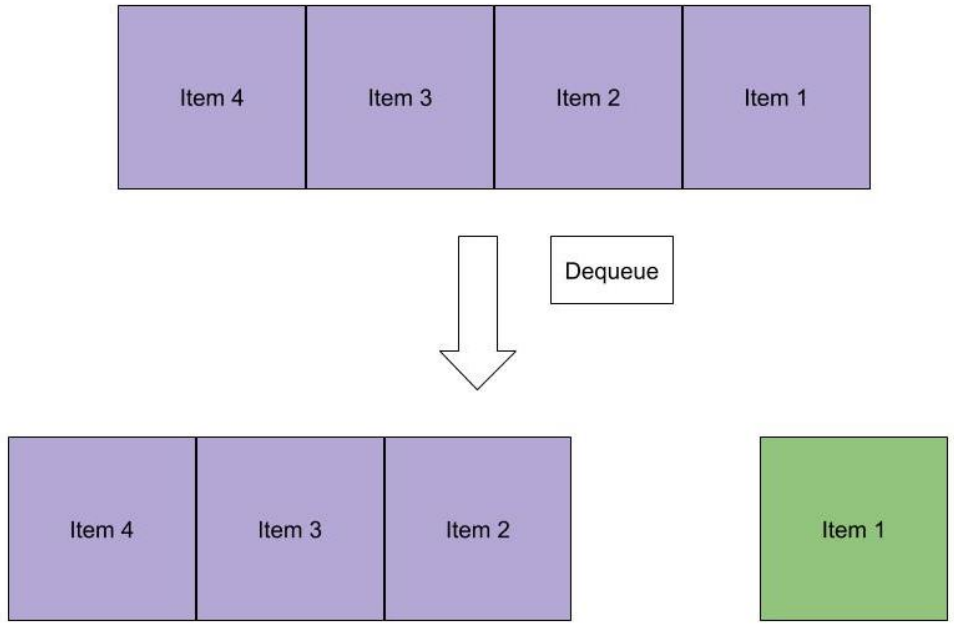


VISUAL REPRESENTATION FOR PUSH AND POP

Queues

- Queues, like the name suggests, follow the **First-in-First-Out** (FIFO) principle. As if waiting in a queue for the movie tickets, the first one to stand in line is the first one to get a ticket.
- To implement a queue, therefore, we need two simple operations
- enqueue - adds an element to the end of the queue
- dequeue - removes the element at the beginning of the queue





VISUAL REPRESENTATION OF QUEUES

Stacks and Queues using Lists

- Python's built-in List data structure comes bundled with methods to simulate both *stack* and *queue* operations.

```
letters = []
# Let's push some letters into our list
letters.append('c')
letters.append('a')
letters.append('t')
letters.append('g')
# Now let's pop letters, we should get 'g'
last_item = letters.pop()
print(last_item)
# If we pop again we'll get 't'
last_item = letters.pop()
print(last_item)
# 'c' and 'a' remain
print(letters)
# ['c', 'a']
```

CONTINUED

- We can use the same functions to implement a Queue. The pop function optionally takes the index of the item we want to retrieve as an argument.
- So we can use pop with the first index of the list i.e. 0, to get queue-like behavior.

```
fruits = []
# Let's enqueue some fruits into our list
fruits.append('banana')
fruits.append('grapes')
fruits.append('mango')
fruits.append('orange')
# Now let's dequeue our fruits, we should get 'banana'
first_item = fruits.pop(0)
print(first_item)
# If we dequeue again we'll get 'grapes'
first_item = fruits.pop(0)
print(first_item)
# 'mango' and 'orange' remain
print(fruits)
```

Stacks and Queues with the deque library

- Python has a deque (pronounced 'deck') library that provides a sequence with efficient methods to work as a stack or a queue.
- deque is short for *Double Ended Queue* - a generalized queue that can get the first or last element that's stored.
- But when it comes to queue, the above list implementation is not efficient. In queue when pop() is made from the beginning of the list which is slow. This occurs due to the properties of list, which is fast at the end operations but slow at the beginning operations, as all other elements have to be shifted one by one.
- So, we prefer the use of dequeue over list, which was specially designed to have fast appends and pops from both the front and back end.

EXAMPLE

```
from collections import deque
# you can initialize a deque with a list
numbers = deque()
# Use append like before to add elements
numbers.append(99)
numbers.append(15)
numbers.append(82)
numbers.append(50)
numbers.append(47)
# You can pop like a stack
last_item = numbers.pop()
print(last_item) # 47
print(numbers) # deque([99, 15, 82, 50])
# You can dequeue like a queue
first_item = numbers.popleft()
print(first_item) # 99
print(numbers) # deque([15, 82, 50])
```

Stricter implementations in Python

- If your code needed a stack and you provide a `List`, there's nothing stopping a programmer from calling `insert`, `remove` or other list functions that will affect the order of your stack! This fundamentally ruins the point of defining a stack, as it no longer functions the way it should.
- There are times when we'd like to ensure that only valid operations can be performed on our data.
- We can create classes that only exposes the necessary methods for each data structure.

EXAMPLE

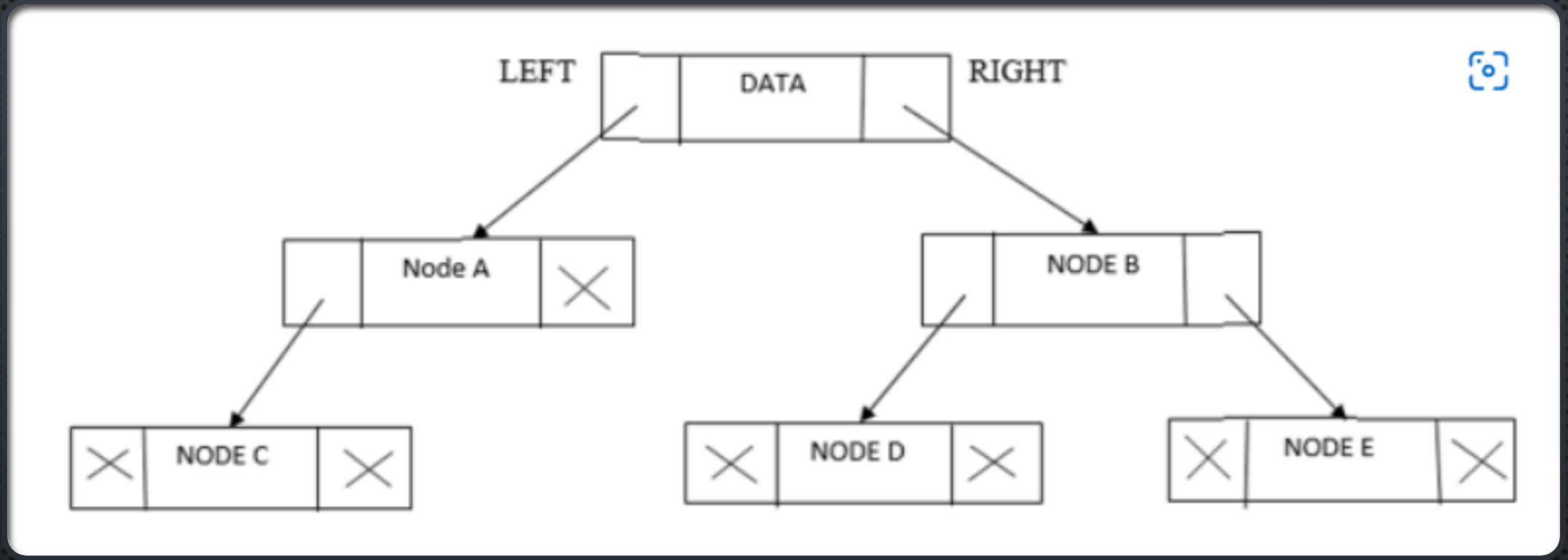
```
# A simple class stack that only allows pop and push operations
class Stack:
    def __init__(self):
        self.stack = []
    def pop(self):
        if len(self.stack) < 1:
            return None
        return self.stack.pop()
    def push(self, item):
        self.stack.append(item)
    def size(self):
        return len(self.stack)
# And a queue that only has enqueue and dequeue operations
class Queue:
    def __init__(self):
        self.queue = []
    def enqueue(self, item):
        self.queue.append(item)
    def dequeue(self):
        if len(self.queue) < 1:
            return None
        return self.queue.pop(0)
    def size(self):
        return len(self.queue)
```

Why have deque and list?

- Lists are built upon blocks of contiguous memory, meaning that the items in the list are stored right next to each other.
- This works great for several operations, like indexing into the `list`. Getting `myList[3]` is fast, as Python knows exactly where to look in memory to find it. This memory layout also allows slices to work well on lists.
- The contiguous memory layout is the reason that `list` might need to take more time to `.append()` some objects than others. If the block of contiguous memory is full, then it will need to get another block, which can take much longer than a normal `.append()`
- `deque`, on the other hand, is built upon a doubly linked list. In a linked list structure, each entry is stored in its own memory block and has a reference to the next entry in the list.
- A doubly linked list is just the same, except that each entry has references to both the previous and the next entry in the list. This allows you to easily add nodes to either end of the list.
- This constant-time addition and removal of entries onto a stack comes with a trade-off, however. Getting `myDeque[3]` is slower than it was for a list, because Python needs to walk through each node of the list to get to the third element.

Binary Trees

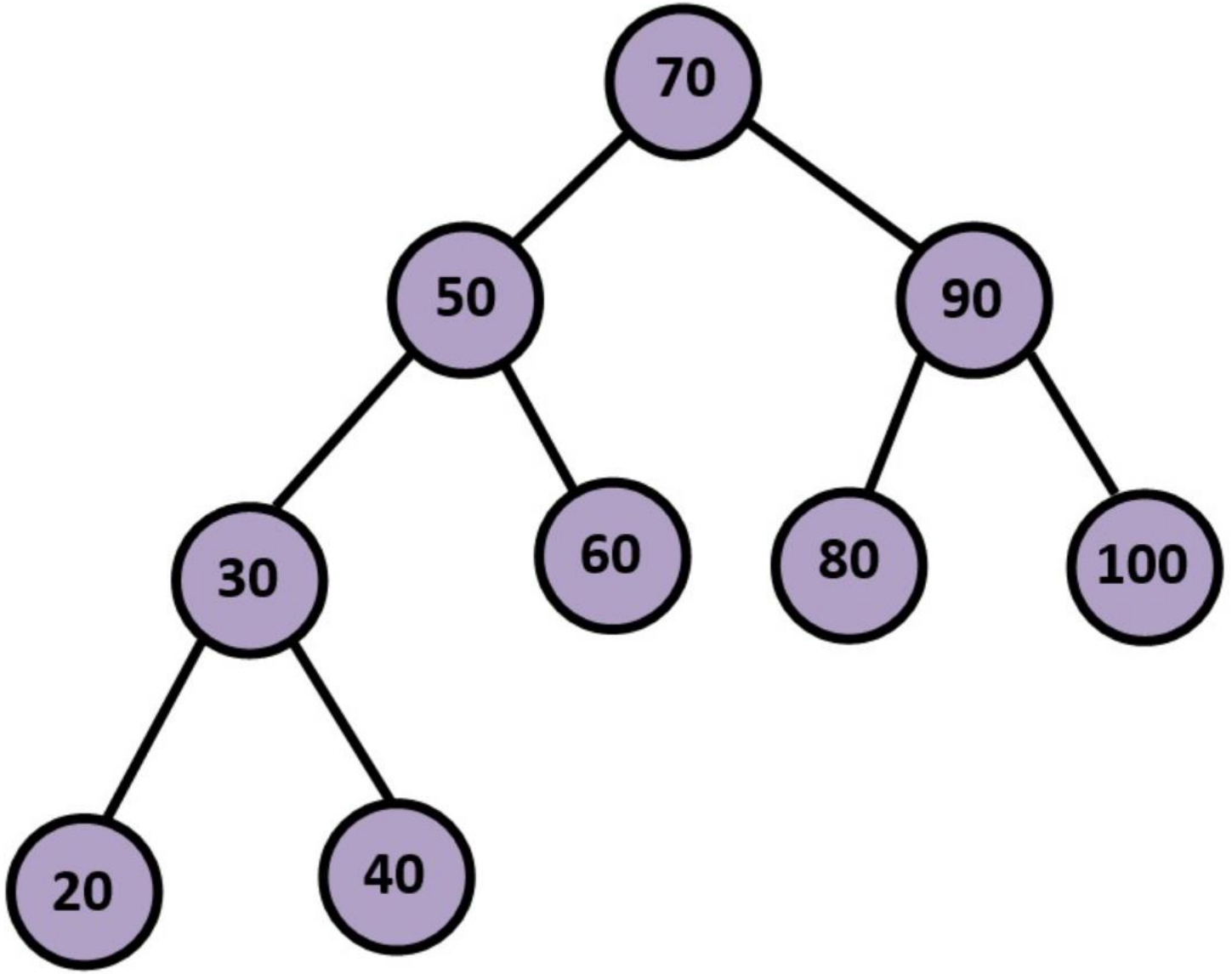
- A binary tree in Python is a nonlinear data structure used for data search and organization.
- The binary tree is comprised of nodes, and these nodes, each being a data component, have left and right child nodes.
- Unlike other data structures, such as, Arrays, Stack and Queue, Linked List which are Linear type data structures, whereas Trees are Hierarchical types of data structures.
- Binary search tree or BST in short, whose nodes each store keys greater than their left child nodes and less than all the right child nodes.
- As the data in a binary tree is organized, it allows operations like insertion, deletion, update and fetch.



VISUAL REPRESENTATION OF A TREE

Components

- Binary tree consists of below components,
- Data, Pointer to Left node & Pointer to Right node.
- Node: Elementary unit of Binary Tree
- Root: Root unit of a binary tree.
- Leaf: Leaves of Binary tree which have nodes without children
- Level: Root is of level 0, children of the root node are level 1, and grandchildren of the root node is level 2
- Parent: Parent of Node that is one level above the node.
- Child: Children of the node are nodes that are one level below the parent node.


70
50
90
30
60
80
100
20
40

ALGORITHM FOR BINARY TREE IN PYTHON

STEP 1: WE NEED TO CREATE A NODE CLASS FOR BINARY TREE DECLARATION. CREATION OF NODE CONSTRUCTOR:

```
CLASS BTNODE:
DEF __INIT__(SELF, KEY):
SELF.VAL = KEY
SELF.LEFT = NONE
SELF.RIGHT = NONE
```

- HERE, WE CAN HAVE KEY-VALUE, BUT IF THERE ISN'T ANY VALUE, THE USER CAN SET IT TO NONE. BOTH CHILD NODES LEFT, AND RIGHT CAN ALSO BE ASSIGNED TO NONE.

CONTINUED

- **STEP 2:** WE CAN INSERT DATA. IF THE NODE DOES NOT HAVE ANY VALUE, THE USER CAN SET A VALUE AND RETURN IT. IF THE USER TRIES TO INSERT ANY DATA THAT EXISTS, WE CAN SIMPLY RETURN THE VALUE. IF THE GIVEN VALUE IS LESS THAN THE NODE VALUE AND HAS A LEFT NODE, THEN WE NEED TO CALL THE INSERT METHOD ON THE LEFT CHILD NODE.
- **STEP 3:** THERE ARE TWO USEFUL HELPER FUNCTIONS, GETMIN, AND GETMAX. THESE ARE SIMPLE RECURSIVE FUNCTIONS THAT TRAVERSE THE EDGES OF THE TREE TO STORE THE SMALLEST OR LARGEST VALUES.
- **STEP 4:** DELETE OPERATION IS ALSO A RECURSIVE FUNCTION BUT RETURNS A NEW STATE OF THE GIVEN NODE AFTER A DELETION OPERATION. ON DELETING ANY CHILD NODE, THE PARENT NODE WILL SET LEFT OR RIGHT DATA TO NONE.

EXAMPLE #1

```
class BTNode:
  def __init__(self, val):
    self.left = None
    self.right = None
    self.val = val
  def rootNode(self):
     print(self.val)
  root = BTNode('A')
  root.rootNode()
```

So here, we have created a Node class and assigned a Root node value ‘A.’

INSERTING NODES IN BINARY TREE

```
class BTNode:
    def __init__(self, val):
        self.leftNode = None
        self.rightNode = None
        self.val = val
    def insertNode(self, val):
        if self.val:
            if val < self.val:
                if self.leftNode is None:
                    self.leftNode = BTNode(val)
                else:
                    self.leftNode.insertNode(val)
            elif val > self.val:
                if self.rightNode is None:
                    self.rightNode = BTNode(val)
                else:
                    self.rightNode.insertNode(val)
            else:
                self.val = val
    def displayTree(self):
        if self.leftNode:
            self.leftNode.displayTree()
        print( self.val),
        If self.rightNode:
            self.rightNode.displayTree()
```

CONTINUED

```
root = BTNode('A')
root.insertNode('B')
root.insertNode('C')
root.insertNode('D')
root.insertNode('E')
root.insertNode('F')
root.displayTree()
```

Types of Binary Trees

- Full Binary Tree: Special type of Binary Tree where every parent node or an internal node has either 2 or no child nodes.
- Perfect Binary Tree: A Binary tree in which each internal node has exactly two children and all leaf nodes at the same level.
- Complete Binary Tree: It is the same as a Full Binary Tree, but all leaf nodes must be at the left, and every level must have both left and right child nodes. And the last leaf node should not have the right child.

CONTINUED

- Pathological Tree: The Binary Tree has a single child, i.e. either left node or right node.
- Skewed Binary Tree: It is similar to a pathological tree in which the binary tree is either dominated by left or right nodes. And it has two types: Left Skewed Binary tree and Right Skewed Binary Tree.
- Balanced Binary Tree: Type of Binary Tree in which difference between the height of left and right subtree for each child node is 0 or 1