1. **Binary Search Tree**

class Node:

def \_\_init\_\_(self, value):

self.value = value

self.left = None

self.right = None

class binarySearchTree:

def \_\_init\_\_(self):

self.root = None

def insert(self, value):

if not self.root:

self.root = Node(value)

else:

current = self.root

while True:

if value < current.value:

if current.left:

current = current.left

else:

current.left = Node(value)

break

elif value > current.value:

if current.right:

current = current.right

else:

current.right = Node(value)

break

else:

break

def search(self, value):

current = self.root

while current:

if current.value == value:

return True

elif value < current.value:

current = current.left

else:

current = current.right

return False

if \_\_name\_\_ == "\_\_main\_\_":

bst = binarySearchTree()

# Insert elements

bst.insert(5)

bst.insert(3)

bst.insert(7)

bst.insert(2)

bst.insert(4)

bst.insert(6)

bst.insert(8)

# Search for an element

print(bst.search(4))

print(bst.search(9))

1. **Heap**

class MinHeap:

def \_\_init\_\_(self):

self.heap = []

def insert(self, value):

self.heap.append(value)

self.\_bubble\_up(len(self.heap) - 1)

def extract\_min(self):

if not self.heap:

return None

# Swap root with the last element

self.\_swap(0, len(self.heap) - 1)

# Pop the last element (min value)

min\_value = self.heap.pop()

# Bubble down the root

self.\_bubble\_down(0)

return min\_value

def \_bubble\_up(self, index):

while index > 0:

parent\_index = (index - 1) // 2

# Compare with parent and swap if needed

if self.heap[parent\_index] > self.heap[index]:

self.\_swap(parent\_index, index)

index = parent\_index

else:

break

def \_bubble\_down(self, index):

while True:

left\_child\_index = 2 \* index + 1

right\_child\_index = 2 \* index + 2

smallest = index

# Find the smallest child

if left\_child\_index < len(self.heap) and self.heap[left\_child\_index] < self.heap[smallest]:

smallest = left\_child\_index

if right\_child\_index < len(self.heap) and self.heap[right\_child\_index] < self.heap[smallest]:

smallest = right\_child\_index

# Swap with the smallest child if needed

if smallest != index:

self.\_swap(index, smallest)

index = smallest

else:

break

def \_swap(self, i, j):

self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

if \_\_name\_\_ == "\_\_main\_\_":

heap = MinHeap()

# Insert elements

heap.insert(5)

heap.insert(3)

heap.insert(7)

heap.insert(2)

heap.insert(4)

# Extract min (smallest element)

print(heap.extract\_min())

# Extract min again

print(heap.extract\_min())