Data Structures in Python

1 Sorting and Searching

1.1 Bubble sort ($\sim O(n^2)$)

1. Compare the first two elements of the list to check them which one is greater.
2. For ascending ordering, place the larger value on the right.
3. Do the same for second, third elements and so on till the whole list is completed. This is the first round.
4. Execute steps (1), (2) and (3) for n number of times where n is the length of the list. (In second round you don't need to consider the last element. In third round no need to consider the last two elements and so on.)

Table 1: **Bubble sort, Total round = length(list) - 1 =3**

First round	Second round	Third round
$\underline{7}\ \underline{2}\ 5\ 4$	$\underline{2}\ \underline{5}\ 4\ 7$	$\underline{2}\ \underline{4}\ 5\ 7$
$2\ \underline{7}\ \underline{5}\ 4$	$2\ \underline{5}\ \underline{4}\ 7$	2 4 5 7
$2\ 5\ \underline{7}\ \underline{4}$	2 4 5 7	
2 5 4 7		
Iteration num. = 3	Iteration num. = 2	Iteration num. = 1

```python
def bubble_sort(input_list):

    L = len(input_list)
    for i in range(L): # Number of rounds
        for j in range(0, L-i-1):
            # Traverse the array from 0 to L-(i+1)
            # Last i elements are already in place
            if input_list[j] > input_list[j+1]:
                swap_value_by_index(input_list, j, j+1)

def swap_value_by_index(a_list, a, b):
    a_list[b], a_list[a] = a_list[a], a_list[b]

input_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
print("Before sorting:")
print(input_list)
bubble_sort(input_list)
print("After bubble sort:")
print(input_list)
```

Output:

```
Before sorting:
[54, 26, 93, 17, 77, 31, 44, 55, 20]
After bubble sort:
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

1.2 Selection sort

Number of comparisons $\sim O(n^2)$
Number of swaps $\sim O(n)(?)$

1. The first element is compared with all other elements in the list one-by-one.
2. If any element is found to be greater than any other element, then they are swapped.
3. After the first iteration, the smallest number is placed at the first position. The same procedure is repeated for the second element and so on.

Table 2: **Selection sort, Total round = length(list) - 1 =3**

First round	Second round	Third round
$\underline{7}$ $\underline{2}$ 5 4	2 $\underline{7}$ $\underline{5}$ 4	2 4 $\underline{7}$ $\underline{5}$
$\underline{2}$ 7 $\underline{5}$ 4	2 $\underline{5}$ 7 $\underline{4}$	2 4 5 7
$\underline{2}$ 7 5 $\underline{4}$	2 4 7 5	
2 7 5 4		
Iteration num. = 3	Iteration num. = 2	Iteration num. = 1

```
def selection_sort(input_list):

    L = len(input_list)

    for i in range(L):
        for j in range(i+1, L):
            min_idx = i
            if input_list[min_idx] > input_list[j]:
                min_idx = j
                swap_value_by_index(input_list, min_idx, i)

def swap_value_by_index(a_list, a, b):
    a_list[b], a_list[a] = a_list[a], a_list[b]
```

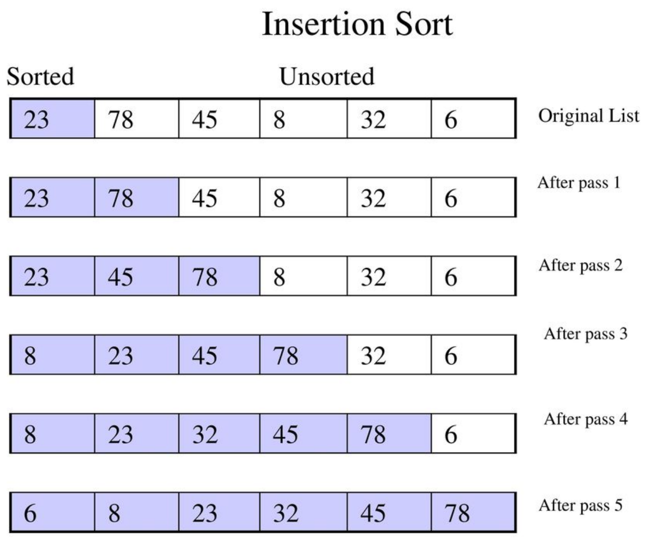
```
input_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
print("Before sorting:")
print(input_list)
selection_sort(input_list)
print("After selection sort:")
print(input_list)
```

Output:

```
Before sorting:
[54, 26, 93, 17, 77, 31, 44, 55, 20]
After selection sort:
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

1.3 Insertion sort ($\sim O(n^2)$)

1. The first iteration starts with comparing the first and second elements and ordering them accordingly.
2. In second iteration, the third element is compared with the first and second elements and placed in a suitable position by shifting other elements.
3. The procedure is repeated for all elements in the array.



```
def insertion_sort(input_list):

    for index in range(1, len(input_list)):
        current_value = input_list[index]
        pos = index

        while pos > 0 and input_list[pos - 1] > current_value:
            input_list[pos] = input_list[pos - 1]
            pos -= 1
        input_list[pos] = current_value

input_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
print("Before sorting:")
print(input_list)
insertion_sort(input_list)
print("After insertion sort:")
print(input_list)
```

Output:

```
Before sorting:
[54, 26, 93, 17, 77, 31, 44, 55, 20]
After insertion sort:
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

2 Stack

Stacks are the data structures in which the items to be inserted or removed follow the **Last-in-First-out** (LIFO) principle. The insertion or deletion of items takes place one end of the stack known as the **top** of the stack.

When an item is added into a stack, the operation is called **push** and when an item is removed from the stack, the operation is called **pop**.

Most microprocessors use a stack-based architecture. When a method (subroutine) is called, its return address is pushed onto a stack and when it returns, the address is popped off.

Applications of stack:

- Reversing a word: You push a word onto a stack letter-by-letter and then pop letters from the stack.
- Doing 'undo' mechanism in text editors. This is accomplished by keeping all text changes in a stack.
- Compiler's syntax check for matching parentheses is acomplished by using stack.

```
class Stack():

    # Constructor
    def __init__(self):
        self.items = []

    # Push item onto stack
    def push(self, item):
        self.items.append(item)

    # Pop item from stack
    def pop(self):
        return self.items.pop()

    # Check whether the stack is empty or not
    def is_empty(self):
        return self.items == []

    # Peek at the topmost element of the stack
    def peek(self):
        if not self.is_empty():
            return self.items[-1]

    # Display the stack
    def display(self):
        #print(self.items[::-1])
        for i in self.items[::-1]:
            st = '| '+str(i)+' |'
            print(st)
            print('-'*len(st))

# Reversing a string using stack
def reverse_string(input_str):

    stack = Stack()
    for x in input_str:
        stack.push(x)

    rev_str = ""
    while not stack.is_empty():
        rev_str += stack.pop()

    return rev_str

# Converting decimal --> binary a string using stack
def convert_binary(dec_num):

    '''
    Example : 242
    --------------
    242 / 2 = 121 -> 0
    121 / 2 = 60 -> 1
    60 / 2 = 30  -> 0
    30 / 2 = 15  -> 0
    15 / 2 = 7   -> 1
    7 / 2 = 3    -> 1
    3 / 2 = 1    -> 1
    1 / 2 = 0    -> 1
    '''
    s = Stack()

    while dec_num > 0:
        remainder = dec_num % 2
        s.push(remainder)
        dec_num = dec_num // 2

    bin_num = ""
    while not s.is_empty():
        bin_num += str(s.pop())
    return bin_num

# Checking the balancing and ordering of parentheses
def is_paren_balanced(paren_string):
    s = Stack()
    is_balanced = True
    index = 0

    while index < len(paren_string) and is_balanced:
        paren = paren_string[index]
        if paren in "([{":
            s.push(paren)
        elif paren in ")}]":
            if s.is_empty():
                is_balanced = False
            else:
                top = s.pop()
                if not is_match(top, paren):
                    is_balanced = False
        index += 1

    if s.is_empty() and is_balanced:
        return True
    else:
        return False

def is_match(p1, p2):
    if p1 == "(" and p2 == ")":
        return True
    elif p1 == "{" and p2 == "}":
        return True
    elif p1 == "[" and p2 == "]":
        return True
    else:
        return False

s = Stack()

s.push(1)
s.push('A')
s.push(31)
s.push(4)
s.display()
```

```
print('Popped item =', s.pop())
s.display()
print('Reverse of \'Hello\' is', reverse_string("Hello"))
print('Binary of \'242\' is', convert_binary(242))
print('[a+b][c+d]:', is_paren_balanced("[a+b][c+d]"))
print('[a+b][c+}d]:', is_paren_balanced("[a+b][c+}d]"))
```

Output:

```
| 4 |
-----
| 31 |
------
| A |
-----
| 1 |
-----
Popped item = 4
| 31 |
------
| A |
-----
| 1 |
-----
Reverse of 'Hello' is olleH
Binary of '242' is 11110010
[a+b][c+d]: True
[a+b][c+}d]: False
```

3 Queue

Queue is an abstract data type or a linear data structure in which the addition of items takes place from one end called the **REAR**(also called **tail**), and the removal of existing items takes place from the other end called as **FRONT**(also called **head**).

This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first. The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.

Applications of Queue:

- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.
- Phone answering system: The person who calls first gets a response first from the phone answering system.

```
class Queue:

    # Constructor
    def __init__(self):
        self.items = []

    # Check whether the queue is empty or not
    def is_empty(self):
        return self.items == []

    # Insert new item to the rear of the queue
    def enqueue(self, item):
        self.items.insert(0, item)

    # Removes the front item from the queue and returns it
    def dequeue(self):
        return self.items.pop()

    # Returns the number of items in the queue
    def size(self):
        return len(self.items)

    # Display the queue
    def display(self):
        for i in self.items:
            print('| {} '.format(i), end='')
        print('|')

q = Queue()
q.enqueue('hello')
q.enqueue('dog')
q.enqueue(3)
q.display()

print('Removed item =', q.dequeue())
q.display()

q.enqueue(61)
q.display()
print('Queue size =', q.size())
```

Output:

```
| 3 | dog | hello |
Removed item = hello
| 3 | dog |
| 61 | 3 | dog |
Queue size = 3
```

4 Deque

Deque or double ended queue is a generalized version of queue data structure that allows insert and delete at both ends.

What makes a deque different is the unrestrictive nature of adding and removing items. New items can be added at either the front or the rear. Likewise, existing items can be removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure.

```
class Deque:

    # Constructor
    def __init__(self):
        self.items = []

    # Check whether the deque is empty or not
    def is_empty(self):
        return self.items == []

    # Insert new item at the front of the deque
    def add_front(self, item):
        self.items.append(item)

    # Insert new item to the rear of the deque
    def add_rear(self, item):
        self.items.insert(0,item)

    # Removes the front item from the deque and returns it
    def remove_front(self):
        return self.items.pop()

    # Removes the rear item from the deque and returns it
    def remove_rear(self):
        return self.items.pop(0)

    # Returns the number of items in the deque
    def size(self):
        return len(self.items)

    # Display the deque
    def display(self):
        for i in self.items:
            print('| {} '.format(i), end='')
        print('|')
```

```python
def main():

    # Main function is used so as not to run this part
    # when imported from outside
    d = Deque()
    d.add_rear(4)
    d.display()
    d.add_rear('dog')
    d.display()
    d.add_front('cat')
    d.display()
    d.add_front(45)
    d.display()
    print('Deque size =', d.size())

    d.display()
    print('Removed item from rear =', d.remove_rear())
    d.display()
    print('Removed item from front =', d.remove_front())
    d.display()

if __name__ == '__main__':
    main()
```

Output:

```
| 4 |
| dog | 4 |
| dog | 4 | cat |
| dog | 4 | cat | 45 |
Deque size = 4
| dog | 4 | cat | 45 |
Removed item from rear = dog
| 4 | cat | 45 |
Removed item from front = 45
| 4 | cat |
```

4.1 Palindrome Checker

An interesting problem that can be easily solved using the deque data structure is the classic palindrome problem. A palindrome is a string that reads the same forward and backward, for example, radar, madam etc.

```
from deque import Deque

def pal_checker(input_str):

    d = Deque()
    is_pal = True

    for ch in input_str:
        d.add_rear(ch)

    while d.size() > 1 and is_pal:
        first = d.remove_front()
        last = d.remove_rear()
        if first != last:
            is_pal = False

    return is_pal


print(pal_checker("technique"))
print(pal_checker("radar"))
```

Output:

```
False
True
```

4.2 Python collection.deque module

Deque can be implemented in python using the module ”collections”.

```
import collections

# Initializing deque
d = collections.deque([1,2,3])
print (d)

# append() inserts element at right end
d.append(4)
print ('Insert 4 at right:', d)

# appendleft() inserts element at left end
d.appendleft(6)
print ('Insert 6 at left:', d)
```

```
# pop() deletes element from right end
d.pop()
print ('Delete from right:', d)

# popleft() to deletes element from left end
d.popleft()
print ('Delete from right:', d)
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
print('~'*40)
d = collections.deque([1, 2, 3, 3, 4, 2, 4])
print (d)

# index(element, beg, end) : This function returns the first index
# of the value element, starting searching from beg till end index.
print ("Number 4 first occurs at: {}".format(d.index(4, 2, 5)))

# insert(i, a) insert the value i at index a
d.insert(4, 3)
print ("After inserting 3 at 5th position: \n{}".format(d))

# count() counts the occurrences of 3
print ("Count of 3 : {}".format(d.count(3)))

# remove() removes the first occurrence of 3
d.remove(3)
print ("After deleting first occurrence of 3: \n{}".format(d))
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
print('~'*40)
d = collections.deque([1, 2, 3,])
print(d)
# extend() to add multiple numbers to right end
d.extend([4,5,6])
print ("The deque after extending deque at end: ")
print (d)

# extendleft() add multiple numbers to left end
d.extendleft([7,8,9])
print ("The deque after extending deque at beginning: ")
print (d)

# rotate(n) to rotates the deque by n to the right
d.rotate(-3) # rotates to the left by 3

# printing modified deque
```

```
print ("The deque after rotating deque: ")
print (d)
```

Output:

```
deque([1, 2, 3])
Insert 4 at right: deque([1, 2, 3, 4])
Insert 6 at left: deque([6, 1, 2, 3, 4])
Delete from right: deque([6, 1, 2, 3])
Delete from right: deque([1, 2, 3])
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
deque([1, 2, 3, 3, 4, 2, 4])
Number 4 first occurs at: 4
After inserting 3 at 5th position:
deque([1, 2, 3, 3, 3, 4, 2, 4])
Count of 3 : 3
After deleting first occurrence of 3:
deque([1, 2, 3, 3, 4, 2, 4])
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
deque([1, 2, 3])
The deque after extending deque at end:
deque([1, 2, 3, 4, 5, 6])
The deque after extending deque at beginning:
deque([9, 8, 7, 1, 2, 3, 4, 5, 6])
The deque after rotating deque:
deque([1, 2, 3, 4, 5, 6, 9, 8, 7])
```

5 Linked List

A linked list is a linear collection of data elements in which the linear ordering is maintained by each element pointing to the next (not by physical placements in memory). It is a data structure consisting of a group of nodes which together form a sequence. Each node is composed of two parts: **data** and a **reference** (link) to the next node in the sequence.

Advantages:

1. Unlike list (array), they are dynamic in nature which allocates memory whenever required.
2. This structure allows efficient insertion and removal of elements.
3. Stacks and queues can easily be implemented.
4. Linked list reduces the access time.

Disadvantages:

1. The memory is wasted as the reference requires extra memory for storage.
2. No element can be accessed randomly.
3. Reverse traversing is difficult.

```python
class Node:

    # Constructor
    def __init__(self, data):
        self.data = data
        self.next = None


class LinkedList:

    # Constructor
    def __init__(self):
        self.head = None

    # Display the linked list
    def display(self):
        cur_node = self.head
        while cur_node:
            print('['+str(cur_node.data)+'] ---> ', end='')
            cur_node = cur_node.next
        print('None')
```

```
# Returns the length of the linked list
def length(self):
    count = 0
    cur_node = self.head

    while cur_node:
        count += 1
        cur_node = cur_node.next
    return count

# Insert item at the end of linked list
def insertLast(self, data):
    new_node = Node(data)

    if self.head is None:
        self.head = new_node
        return None

    last_node = self.head
    while last_node.next:
        last_node = last_node.next
    last_node.next = new_node

# Insert item at the beginning of linked list
def insertFirst(self, data):
    new_node = Node(data)

    new_node.next = self.head
    self.head = new_node

# Insert item at a specific position of linked list
def insert_at_pos(self, pos, data):

    if pos < 0 or pos > self.length():
        print('No node exists at this position')
        return None

    new_node = Node(data)
    cur_node = self.head
    if pos == 0:
        self.insertFirst(data)
    else:
        for i in range(pos-1):
            cur_node = cur_node.next
        new_node.next = cur_node.next
        cur_node.next = new_node

# Delete item by value (key)
def delete_by_key(self, key):

    cur_node = self.head

    if cur_node and cur_node.data == key:
        self.head = cur_node.next
        cur_node = None
        return None

    prev_node = None
    while cur_node and cur_node.data != key:
        prev_node = cur_node
        cur_node = cur_node.next

    if cur_node is None:
        return None

    prev_node.next = cur_node.next
    cur_node = None

# Delete item by position(index)
def delete_at_pos(self, pos):

    cur_node = self.head

    if pos == 0:
        self.head = cur_node.next
        cur_node = None
        return None

    prev_node = None
    count = 1
    while cur_node and count != pos:
        prev_node = cur_node
        cur_node = cur_node.next
        count += 1

    if cur_node is None:
        return

    prev_node.next = cur_node.next
```

```
            cur_node = None

    # Reversing the linked list
    def reverse(self):
        cur_node = self.head
        prev_node = None

        while cur_node:
            next_node = cur_node.next
            # next_node for saving cur_node.next
            cur_node.next = prev_node
            prev_node = cur_node
            cur_node = next_node
        self.head = prev_node

    # Search the element in a linked list using key
    def searchkey(self, key):
        cur_node = self.head
        pos = 0

        if self.head == None:
            print('No such value found in the Linked List')
        else:
            while cur_node:
                if cur_node.data == key:
                    print("Search value index = "+str(pos))
                    return None
                cur_node = cur_node.next
                pos += 1
            if cur_node == None:
                print('No such value found in the Linked List')

llist = LinkedList()
llist.insertLast("A")
llist.insertLast("B")
llist.insertLast("C")
llist.display()

llist.insert_at_pos(2, 'D')
llist.insert_at_pos(1, 5)
llist.display()

llist.delete_by_key("B")
llist.display()
llist.reverse()
llist.display()

llist.searchkey('A')
llist.display()
```

Output:

```
[A] ---> [B] ---> [C] ---> None
[A] ---> [5] ---> [B] ---> [D] ---> [C] ---> None
[A] ---> [5] ---> [D] ---> [C] ---> None
[C] ---> [D] ---> [5] ---> [A] ---> None
Search value index = 3
[C] ---> [D] ---> [5] ---> [A] ---> None
```

6 Binary Search Tree

```
# Binary Search Tree
class Node():
    # Constructor
    def __init__(self, data):
        self.data = data
        self.leftChild = None
        self.rightChild = None


class BinarySearchTree():
    # Constructor
    def __init__(self, data):
        self.root = Node(data)


    # Find a Node in a BST
    def findNode(self, data):
        if self.root.data == data:
            return self.root
        else:
            currentNode = self.root
            while data != currentNode.data:
                if data < currentNode.data:
                    currentNode = currentNode.leftChild
                else:
                    currentNode = currentNode.rightChild
                if currentNode == None: # If no child
                    return None
            return currentNode


    # Insert a New Node in a Binary Search Tree
    def insert(self, data):
        currentNode = self.root
        while True:
            if data < currentNode.data:
                if not currentNode.leftChild:
                    currentNode.leftChild = Node(data)
                    return None
                currentNode = currentNode.leftChild
            else:
                if not currentNode.rightChild:
```

```
                currentNode.rightChild = Node(data)
                return None
            currentNode = currentNode.rightChild


def getPredecessor(self,node):
    if node.rightChild:
        return self.getPredecessor(node.rightChild)
    return node


# Delete node from Binart Search Tree
# To-Implement
# def deleteNode(self, data):
#     currentNode = self.root
#     isLeftChild = True
#     # 'isLeftChild' is used to check whether the current
#     # is whether the left or right child of parents
#     while data != currentNode:
#         parentNode = currentNode
#         if data < currentNode.data:
#             isLeftChild = True
#             currentNode = currentNode.leftChild
#         else:
#             isLeftChild = False
#             currentNode = currentNode.rightChild
#         if currentNode == None:
#             return False # No no found


# Remove Node from Binary Search Tree
def remove(self,data):
    if self.root:
        self.root = self.removeNode(data, self.root)

def removeNode(self, data, node):
    if not node:                                    # If node is
       Empty i.e None, return the node
        return node

    # If data < root, traverse through the left child till we
       get the "data" node
    # Look for node we want to get rid of
    if data < node.data:                            # If the node
       data to be deleted is less than root, go to left
```

```
        node.leftChild = self.removeNode(data, node.leftChild)
            # Tell parent node that its left child is None
    elif data > node.data:
        node.rightChild = self.removeNode(data, node.rightChild)

    # Else we are standing at that node which we want to delete.
    # It may be a node with children (1 or 2) or a leaf node
    else:
        # If the current node is a leaf node i.e no left and
            right child
        if not node.leftChild and not node.rightChild:
            print('Removing leaf node...')
            del node
            return None                          # Returning
                None tells its parent Node that its left or
                right child has been deleted
        # If the node to be deleted is a parent and has only
            one child (on right)
        # Node -> Right Child
        if not node.leftChild:
            print('Removing Node with single right child...')
            tempNode = node.rightChild           # Put the right
                child value in a temporary variable
            del node                             # Delete the
                parent node
            return tempNode                      # Return the
                temp node. Parent of deleted node --> Temp node

        # If the node to be deleted is a parent and has only
            one child (on Left)
        # Node -> Left Child
        elif not node.rightChild:
            print('Removing Node with single left child...')
            tempNode = node.leftChild
            del node
            return tempNode

        # If node to be deleted is a parent node with two
            children
        print('Removing node with two children...')
        # Fetch the largest node in left subtree to root in
            temp Node.
        tempNode = self.getPredecessor(node.leftChild)
        # Replace the node to be deleted data with the largest
            node value data
```

```python
        node.data = tempNode.data
        # Remove the node with largest value in the left
           subtree to root/node to be deleted.
        node.leftChild = self.removeNode(tempNode.data,
           node.leftChild)
    return node


# Find the parent node
def findParentNode(self, data):
    if data == self.root.data:
        return None # Data found at root node, no parent
    currentNode = self.root
    while data != currentNode.data:
        if data < currentNode.data:
            parentNode = currentNode # Saving the parent node
            currentNode = currentNode.leftChild
        else:
            parentNode = currentNode
            currentNode = currentNode.rightChild
        if currentNode == None: # data is not found
            return None
    return parentNode


# Get the Minimum Value in a BST
# The left most value is the Smallest Value
def getMinValue(self):
    currentNode = self.root
    while True:
        if not currentNode.leftChild:
            return currentNode.data
        currentNode = currentNode.leftChild


# Get the Maximum Value in a BST
# The right most value is the Maximum Value
def getMaxValue(self):
    currentNode = self.root
    while True:
        if not currentNode.rightChild:
            return currentNode.data
        currentNode = currentNode.rightChild
    # Traversing BST In-Order: Left --> Root --> Right
    def traverseInOrder(self, rootNode):
        currentNode = rootNode
        if currentNode.leftChild:
            self.traverseInOrder(currentNode.leftChild)
        print(currentNode.data, end=" ")
        if currentNode.rightChild:
            self.traverseInOrder(currentNode.rightChild)


    # Traversing BST Pre-Order: Root --> Left --> Right
    def traversePreOrder(self, rootNode):
        currentNode = rootNode
        print(currentNode.data, end=" ")
        if currentNode.leftChild:
            self.traversePreOrder(currentNode.leftChild)
        if currentNode.rightChild:
            self.traversePreOrder(currentNode.rightChild)


    # Traversing BST Pre Order: Left --> Right --> Root
    def traversePostOrder(self, rootNode):
        currentNode = rootNode
        if currentNode.leftChild:
            self.traversePostOrder(currentNode.leftChild)
        if currentNode.rightChild:
            self.traversePostOrder(currentNode.rightChild)
        print(currentNode.data, end=" ")


if __name__ == '__main__':
    bst = BinarySearchTree(10)
    bst.insert(13)
    bst.insert(14)
    bst.insert(5)
    bst.insert(1)

    foundNode = bst.findNode(75)
    if foundNode == None:
        print("Could not find the node with data = 75")
    else:
        print("Found the node with data = 75")

    foundNode = bst.findParentNode(1)
    if foundNode == None:
        print("Could not find the parent node with data = 1")
    else:
        print(f"The data in the parent node is {foundNode.data}")

    print("Inorder traversing --> ")
    bst.traverseInOrder(bst.root)
    print()
    print("Preorder traversing --> ")
    bst.traversePreOrder(bst.root)
    print()
    print("Preorder traversing --> ")
    bst.traversePostOrder(bst.root)
    print()

    bst.remove(10)
```

Output:

```
Could not find the node with data = 75
The data in the parent node is 5
Inorder traversing -->
1 5 10 13 14
Preorder traversing -->
10 5 1 13 14
Preorder traversing -->
1 5 14 13 10
Removing node with two children...
Removing Node with single left child...
```

7 Heap

Heap data structure is a *complete binary tree* that satisfies the *heap property*. It is also called as a binary heap. A *complete binary* tree is a special binary tree in which satisfies the following conditions -

- Each level is completely filled before another level is added.
- The bottom tier is filled in from left to right.

Heap Property is the property of a node which satisfies the following conditions -

- Max heap: Data(key) of each node is always greater than its child node(s).
- Min heap: Data(key) of each node is always smaller than its child node(s).

Another interesting property of a complete tree is that we can represent it using a single list. Because the tree is complete, for a node at index p in the parent,

- Its parent is at $\frac{p-1}{2}$ (integer division).
- Its left child is at $2p + 1$.
- Its right child is at $2p + 2$.

```
class Heap:
    def __init__(self, MAX_SIZE):
        self.heapList = []
        self.currentSize = 0
        self.MAX_SIZE = MAX_SIZE


    def insert(self, item):
        self.heapList.append(item)
        self.currentSize += 1
        self.bubbleUp(self.currentSize - 1)


    def bubbleUp(self, index):
        parentIdx = abs(index - 1) // 2
        while (index > 0) and (self.heapList[index] >
           self.heapList[parentIdx]):
            # Swap the child and parent nodes
            self.heapList[index], self.heapList[parentIdx] =
               self.heapList[parentIdx], \
                                self.heapList[index]
            index = parentIdx
            parentIdx = (index - 1) // 2


    def remove(self):
        lastItem = self.heapList[self.currentSize - 1]
        self.heapList[0] = lastItem
        self.currentSize -= 1
        self.heapList.pop() # Remove the last element in the list
        self.sinkDown(0)


    def sinkDown(self, index):
        itemAtIndex = self.heapList[index] # Saving the item at
           index
        largerIdx = self.getLargerChildIdx(index)

        while (largerIdx != -1) and (itemAtIndex <
           self.heapList[largerIdx]):
            self.heapList[index], self.heapList[largerIdx] =
               self.heapList[largerIdx], \
                                         self.heapList[index]
            index = largerIdx
            largerIdx = self.getLargerChildIdx(index)


    def getLargerChildIdx(self, index):
        leftChildIdx = 2*index + 1
        rightChildIdx = 2*index + 2
        if leftChildIdx >= self.currentSize:
            # No child exist
            return -1
        elif rightChildIdx >= self.currentSize:
            # Left child exist but not the right child
            return leftChildIdx
        else:
            # Both children exist
            if self.heapList[leftChildIdx] >
               self.heapList[rightChildIdx]:
                # Left child value is greater than the right child
                   value
                return leftChildIdx
            else:
                return rightChildIdx


if __name__ == '__main__':
    heap = Heap(10)
    for item in [82, 70, 53, 63, 27, 43, 37, 10, 51]:
        heap.insert(item)
    print(heap.heapList)
    heap.remove()
    print(heap.heapList)
```

Output:

```
[82, 70, 53, 63, 27, 43, 37, 10, 51]
[70, 63, 53, 51, 27, 43, 37, 10]
```