DATA STRUCTURES AND ALGORITHMS DESIGN

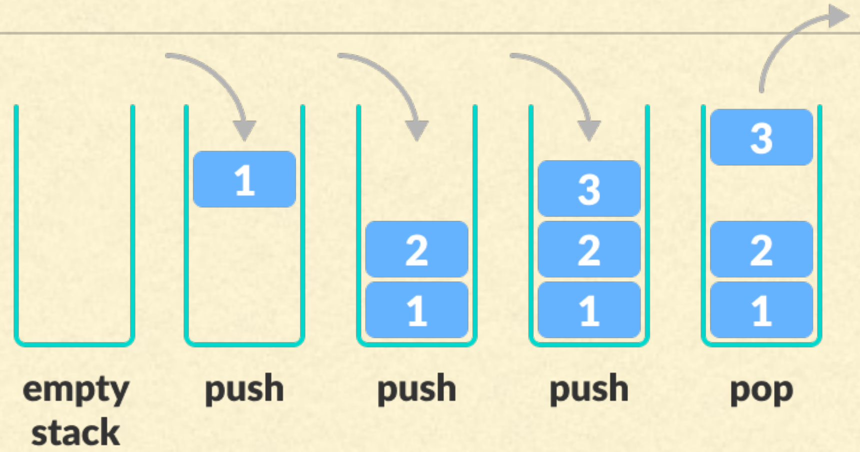
DATA STRUCTURES - ASSIGNMENTS

Harsha M S

“Talk is cheap, show me the code.”

- Linus Torvalds

STACK



- Push(n) - O(1)
- Pop(n) - O(1)
- Top - O(n)
- Search - O(n)

Initialize stack

0	1	2

top=-1

push(1)

0	1	2
1		

top

push(2),push(3)

0	1	2
1	2	3

top

3 =pop()

0	1	2
1	2	

top

2 =pop()

0	1	2
1		

top

1=pop()

0	1	2

top=-1

```python
class Stack:
        top = -1
        size = 0
        array = []

        def __init__(self,size):
                self.top = -1
                self.size = size
                self.array = [0]* size

        def is_empty(self):
                if self.top == -1:
                        return True
                return False

        def push(self, element):
                if self.top == self.size -1:
                        print('Stack overflow!')
                        return
                self.top+=1
                self.array[self.top] = element

        def pop(self):
                if self.top == -1:
                        print('Stack underflow!')
                        return
                removed_element = self.array[self.top]
                self.top -= 1
                return removed_element

        def print(self):
                print('Bounded Stack: ')
                print('Stack elements:',end =" ")
                if self.top == -1:
                        print("[]")
                else:
                        print([self.array[i] for i in range(0,self.top+1)])
```

```
>>> from BoundedStack import Stack
>>> s=Stack(3)
>>> s.is_empty()
True
>>> s.pop()
Stack underflow!
>>> s.push(1)
>>> s.print()
Bounded Stack:
Stack elements: [1]
>>> s.push(2)
>>> s.push(3)
>>> s.print()
Bounded Stack:
Stack elements: [1, 2, 3]
>>> s.push(4)
Stack overflow!
>>> s.is_empty()
False
>>> s.print()
Bounded Stack:
Stack elements: [1, 2, 3]
>>> s.pop()
3
>>> s.print()
Bounded Stack:
Stack elements: [1, 2]
>>> s.pop()
2
>>> s.pop()
1
>>> s.print()
Bounded Stack:
Stack elements: []
>>> s.pop()
Stack underflow!
>>> s.is_empty()
True
```

UNBOUNDED STACK

```python
class Stack:
        top = -1
        array = []

        def __init__(self):
                self.top = -1
                self.array = []

        def is_empty(self):
                if self.top == -1:
                        return True
                return False

        def push(self, element):
                self.array.append(element)
                self.top += 1

        def pop(self):
                if self.top == -1:
                        print('Stack underflow')
                        return
                self.top -= 1
                return self.array.pop()

        def print(self):
                print('UnBounded Stack: #Elements: '+str(len(self.array)))
                print('Elements:', end = ' ')
                print(self.array)
```

```
>>> from UnBoundedStack import Stack
>>> s=Stack()
>>> s.is_empty()
True
>>> s.push(1)
>>> s.push(2)
>>> s.push(3)
>>> s.push(4)
>>> s.print()
UnBounded Stack: #Elements: 4
Elements: [1, 2, 3, 4]
>>> s.is_empty()
False
>>> s.pop()
4
>>> s.pop()
3
>>> s.pop()
2
>>> s.pop()
1
>>> s.pop()
Stack underflow
>>> s.is_empty()
True
>>> s.print()
UnBounded Stack: #Elements: 0
Elements: []
```

QUEUE

Bounded Queue

```python
class Queue:
	head = -1
	tail = -1
	array = []
	size = 0

	def __init__(self, size):
		self.head = 0
		self.tail = -1
		self.size = size
		self.array = [0] * size

	def isFull(self):
		return self.size

	def enqueue(self, element):
		if self.tail == self.size -1:
			print('Queue overflow!')
			return
		self.tail += 1
		self.array[self.tail] = element

	def dequeue(self):
		if self.head == self.tail + 1:
			print('Queue underflow!')
			return
		element = self.array[self.head]
		self.head += 1
		return element

	def print(self):
		if self.head == self.tail +1:
			print('Queue is empty')
		else:
			print('Elements:',end =" ")
			print([ self.array[i] for i in range(self.head,self.tail+1)])
```

```
>>> from BoundedQueue import Queue
>>> q=Queue(3)
>>> q.print()
Queue is empty
>>> q.dequeue()
Queue underflow!
>>> q.enqueue(1)
>>> q.print()
Elements: [1]
>>> q.enqueue(2)
>>> q.enqueue(3)
>>> q.print()
Elements: [1, 2, 3]
>>> q.enqueue(4)
Queue overflow!
>>> q.dequeue()
1
>>> q.print()
Elements: [2, 3]
>>> q.dequeue()
2
>>> q.print()
Elements: [3]
>>> q.dequeue()
3
>>> q.dequeue()
Queue underflow!
```

UnBounded Queue

```python
class Queue:
        array = []

        def __init__(self):
                self.array = []

        def enqueue(self,element):
                self.array.append(element)

        def dequeue(self):
                if len(self.array) == 0:
                        print('Queue empty')
                else:
                        return self.array.pop(0)

        def print(self):
                print('Elements: ',end = " ")
                print(self.array)
```

```
>>> from UnBoundedQueue import Queue
>>> q=Queue()
>>> q.print()
Elements:  []
>>> q.enqueue(1)
>>> q.print()
Elements:  [1]
>>> q.enqueue(2)
>>> q.enqueue(3)
>>> q.print()
Elements:  [1, 2, 3]
>>> q.dequeue()
1
>>> q.print()
Elements:  [2, 3]
>>> q.dequeue()
2
>>> q.dequeue()
3
>>> q.dequeue()
Queue empty
>>> q.print()
Elements:  []
```

LINKED LIST

- Insertion - O(1)
- Deletion - O(1)
- Search - O(n)

LINKED LIST

```python
class Node:
    value = None
    next = None

    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList:
    head = None

    def __init__(self):
        self.head = None

    def addNode(self, value):
        if self.head is None:
            self.head = Node(value)
        else:
            current = self.head
            while current.next is not None:
                current = current.next
            current.next = Node(value)

    def deleteNodeByValue(self, value):
        if self.head.value == value:
            self.head = self.head.next
        else:
            current = self.head
            previous = self.head
            while current is not None and current.value != value :
                previous = current
                current = current.next
            if current is not None:
                previous.next = current.next
            else:
                print('No node found')

    def deleteNodeByPosition(self, position):
        position -= 1 #considering given position starts from 1
        if position == 0:
            self.head = self.head.next
        else:
            i = 0
            current = self.head
            previous = self.head
            while i < position and i != position:
                previous = current
                current = current.next
                i +=1
            if i > position:
                print('Invalid position')
            else:
                previous.next = current.next

    def print(self):
        current = self.head
        while current is not None:
            print(current.value, end =" ")
            current = current.next
        print()
```

```
>>> from LinkedList import LinkedList
>>> ll=LinkedList()
>>> ll.print()

>>> ll.addNode(1)
>>> ll.addNode(2)
>>> ll.addNode(3)
>>> ll.addNode(4)
>>> ll.addNode(5)
>>> ll.print()
1 2 3 4 5
>>> ll.deleteNodeByValue(2)
>>> ll.print()
1 3 4 5
>>> ll.deleteNodeByPosition(3)
>>> ll.print()
1 3 5
>>> ll.deleteNodeByValue(1)
>>> ll.print()
3 5
>>> ll.deleteNodeByPosition(1)
>>> ll.print()
5
```

DOUBLE LINKED LIST

```python
class Node:
    value = None
    previous = None
    next = None

    def __init__(self, value):
        self.value = value
        self.previous = None
        self.next = None

class LinkedList:
    head = None

    def __init__(self):
        self.head = None

    def addNode(self, value):
        if self.head is None:
            self.head = Node(value)
        else:
            current = self.head
            while current.next is not None:
                current = current.next
            new_node = Node(value)
            current.next = new_node
            new_node.previous = current

    def deleteNodeByValue(self, value):
        if self.head.value == value:
            self.head = self.head.next
            self.head.previous = None
        else:
            previous = self.head
            current = self.head
            while current is not None and current.value != value:
                previous = current
                current = current.next
            if current is None:
                print('Node with given value not found!')
            else:
                previous.next = current.next
                current.previous = None
                current.next = None

    def deleteNodeByPosition(self, position):
        position -= 1
        if position == 0:
            self.head = self.head.next
            self.head.next.previous = None
        else:
            i = 0
            previous = self.head
            current = self.head
            while i < position and i != position:
                previous = current
                current = current.next
                i +=1
            if i > position:
                print('Invalid position')
            else:
                current.previous = None
                previous.next = current.next
                current.next = None
    def print(self):
        current = self.head
        while current is not None:
            print(current.value, end =" ")
            current = current.next
        print()
```

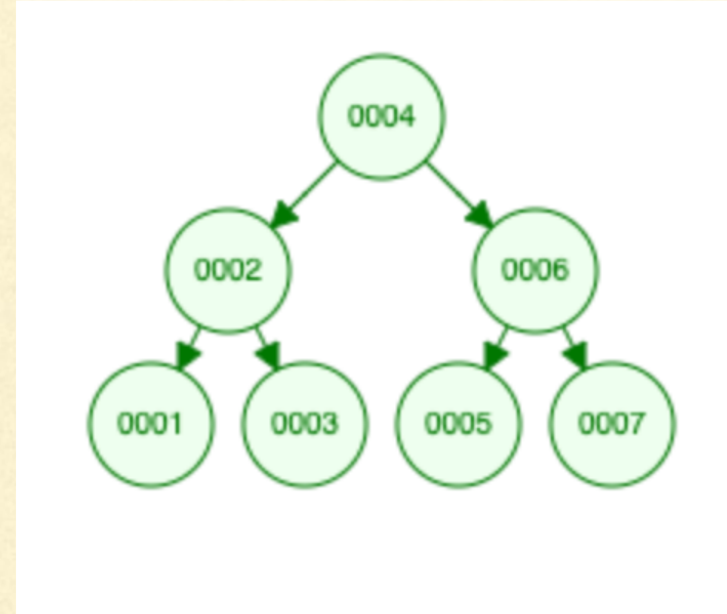
```
>>> from DoubleLinkedList import LinkedList
>>> ll=LinkedList()
>>> ll.addNode(1)
>>> ll.addNode(2)
>>> ll.print()
1 2
>>> ll.addNode(3)
>>> ll.addNode(4)
>>> ll.addNode(5)
>>> ll.print()
1 2 3 4 5
>>> ll.deleteNodeByValue(1)
>>> ll.print()
2 3 4 5
>>> ll.deleteNodeByValue(3)
>>> ll.print()
2 4 5
>>> ll.addNode(6)
>>> ll.addNode(7)
>>> ll.deleteNodeByPosition(1)
>>> ll.print()
4 5 6 7
>>> ll.deleteNodeByPosition(2)
>>> ll.print()
4 6 7
```

BINARY TREE

- Insertion - O(n)
- Deletion - O(n)
- Searching - O(n)

BINARY TREE

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




```python
class Node:
    value = None
    lChild = None
    rChild = None

    def __init__(self, value):
        self.value = value
        self.lChild = None
        self.rChild = None

    def addLChild(self,value):
        self.lChild = Node(value)

    def addRChild(self,value):
        self.rChild = Node(value)

    def print(self, order = "inorder"):
        if order == "inorder":
            if self.lChild is not None:
                self.lChild.print(order)
            print(self.value,end =' ')
            if self.rChild is not None:
                self.rChild.print(order)
        elif order == "preorder":
            print(self.value, end =' ')
            if self.lChild is not None:
                self.lChild.print(order)
            if self.rChild is not None:
                self.rChild.print(order)
        elif order == "postorder":
            if self.lChild is not None:
                self.lChild.print(order)
            if self.rChild is not None:
                self.rChild.print(order)
            print(self.value, end =' ')
```

```
>>> from BinaryTree import Node
>>> r=Node(4)
>>> r.addLChild(2)
>>> r.lChild.addLChild(1)
>>> r.lChild.addRChild(3)
>>> r.addRChild(6)
>>> r.rChild.addLChild(5)
>>> r.rChild.addRChild(7)
>>> r.print()
1 2 3 4 5 6 7 >>>
>>> r.print(order='preorder')
4 2 1 3 6 5 7 >>>
>>> r.print(order='postorder')
1 3 2 5 7 6 4 >>>
```

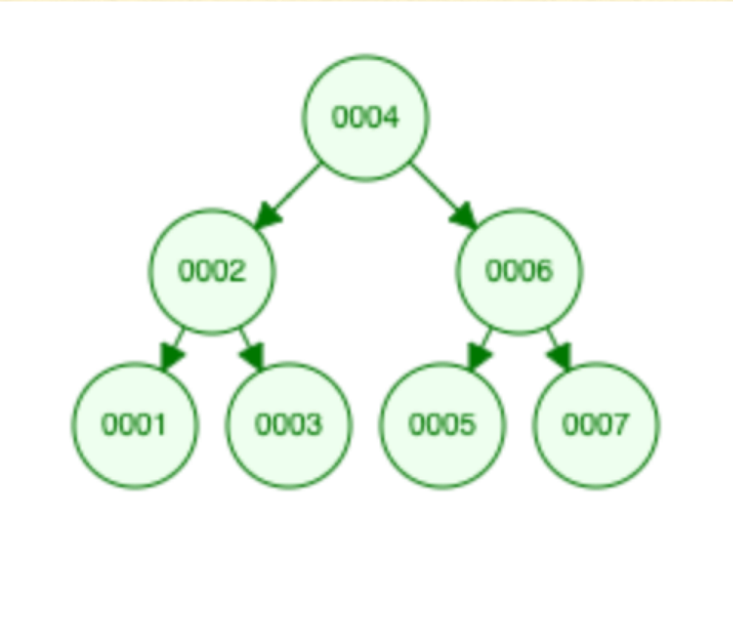
BINARY SEARCH TREE

- Let h - height of BST
- Insertion - O(h)
- Deletion - O(h)
- Searching - O(h)

BINARY SEARCH TREE

```python
class Node:
        def __init__(self, value):
                self.left = None
                self.right = None
                self.value = value

        def addNode(self, value):
                if self.value :
                        if value < self.value:
                                if self.left is None:
                                        self.left = Node(value)
                                else:
                                        self.left.addNode(value)
                        elif value > self.value:
                                if self.right is None:
                                        self.right = Node(value)
                                else:
                                        self.right.addNode(value)
                        else:
                                self.value = value
        def print(self,order = "inorder"):
                if order == "inorder":
                        if self.left is not None:
                                self.left.print(order)
                        print(self.value, end = " ")
                        if self.right is not None:
                                self.right.print(order)
                elif order == "preorder":
                        print(self.value, end=" ")
                        if self.left is not None:
                                self.left.print(order)
                        if self.right is not None:
                                self.right.print(order)
                elif order == "postorder":
                        if self.left is not None:
                                self.left.print(order)
                        if self.right is not None:
                                self.right.print(order)
                        print(self.value, end =" ")
```

SIMULATION



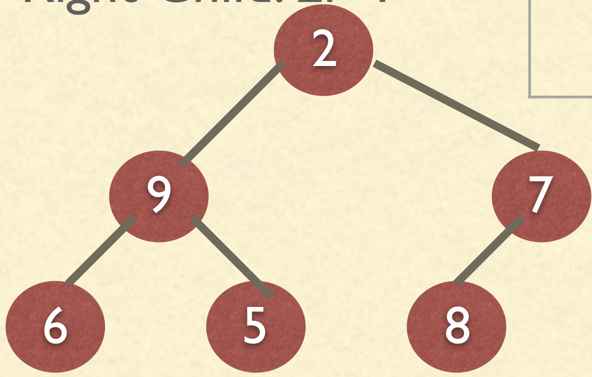

```
>>> from BinarySearchTree import Node
>>> root=Node(4)
>>> root.addNode(2)
>>> root.addNode(1)
>>> root.addNode(3)
>>> root.addNode(6)
>>> root.addNode(5)
>>> root.addNode(7)
>>> root.print()
1 2 3 4 5 6 7 >>>
>>> root.print(order='preorder')
4 2 1 3 6 5 7 >>>
>>> root.print(order='postorder')
1 3 2 5 7 6 4 >>>
```

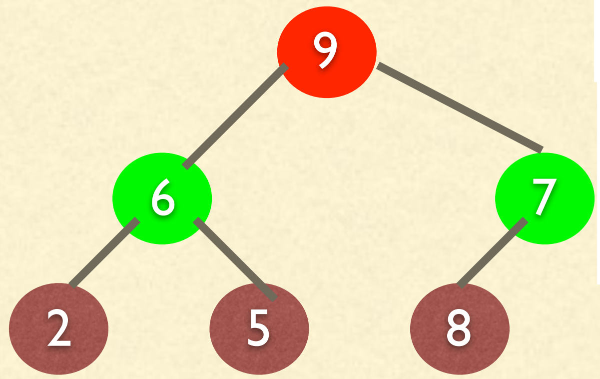
SIMULATION

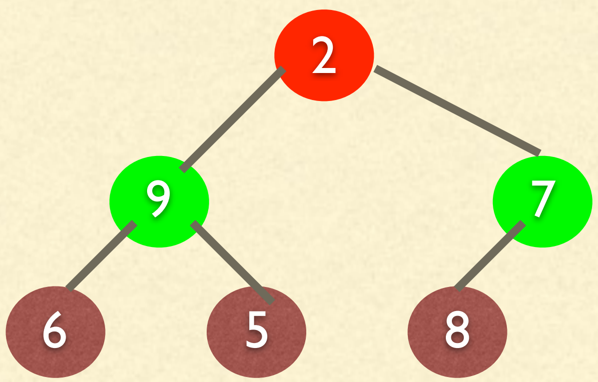
Insertion - O(n log n)
Deletion - O(n log n)
Find Max - O(1)
Extract Max - O(n log n)

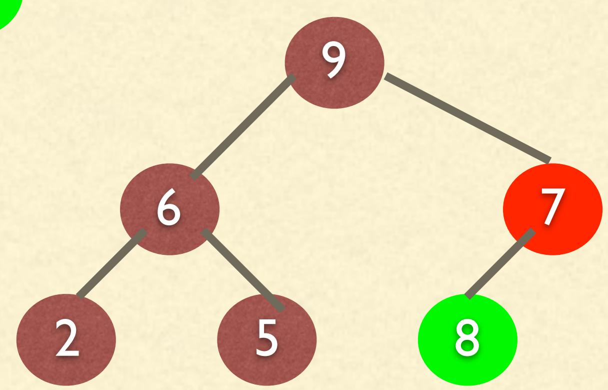
Left Child: 2i
Right Child: 2i+1

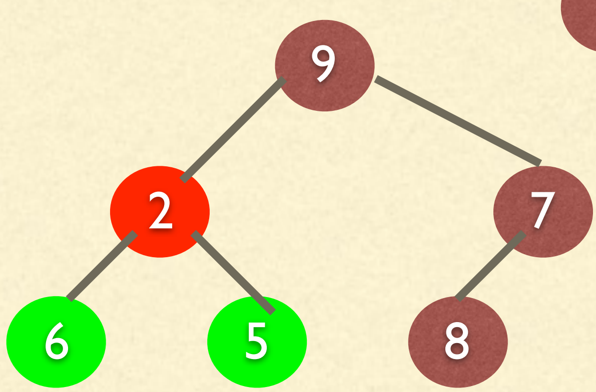
1	2	3	4	5	6
2	9	7	6	5	8











BUILD-MAX-HEAP(A)

```
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

HEAPSORT(A)

```
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

MAX-HEAPIFY(A, i)

```
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

PARENT(i)

```
return ⌊i/2⌋
```

LEFT(i)

```
return 2i
```

RIGHT(i)

```
return 2i + 1
```

```python
from math import floor
def left(i):
        return 2*i
def right(i):
        return  (2*i)+1

def max_heapify(heap_size, arr,i):
        l = left(i)
        r = right(i)
        largest = 0
        if l <= heap_size-1 and arr[l] > arr[i]:
                largest = l
        else:
                largest = i
        if r <= heap_size-1 and arr[r] > arr[l]:
                largest = r
        if largest != i:
                temp = arr[i]
                arr[i] = arr[largest]
                arr[largest] = temp
                max_heapify(heap_size,arr, largest)
h = [2,9,7,6,5,8]
h.insert(0,0)
print(h)
for i in reversed(range(1,floor((len(h)-1)/2)+1)):
        max_heapify(len(h)-1,h,i)
print(h)
heap_size = len(h)-1
for i in reversed(range(2,len(h))):
        temp = h[1]
        h[1] = h[i]
        h[i] = temp
        heap_size -= 1
        max_heapify(heap_size,h,1)
print(h)
```

PS4 - Monk and the power of time

The Monk is trying to explain to its users that even a single unit of time can be extremely important and to demonstrate this particular fact he gives them a challenging task.

There are **N** processes to be completed by you, the chosen one, since you're Monk's favorite student. All the processes have a unique number assigned to them from **1 to N**.

Now, you are given two things:

- The **calling** order in which all the processes are called.
- The **ideal** order in which all the processes should have been executed.

Now, let us demonstrate this by an example. Let's say that there are **3 processes**, the calling order of the processes is: **3 - 2 - 1**. The ideal order is: **1 - 3 - 2**, i.e., process number 3 will only be executed after process number 1 has been completed; process number 2 will only be executed after process number 3 has been executed.

- *Iteration #1:* Since the ideal order has process #1 to be executed firstly, the calling ordered is changed, i.e., the first element has to be pushed to the last place. Changing the position of the element takes 1 unit of time. The new calling order is: 2 - 1 - 3. Time taken in step #1: 1.
- *Iteration #2:* Since the ideal order has process #1 to be executed firstly, the calling ordered has to be changed again, i.e., the first element has to be pushed to the last place. The new calling order is: 1 - 3 - 2. Time taken in step #2: 1.
- *Iteration #3:* Since the first element of the calling order is same as the ideal order, that process will be executed. And it will be thus popped out. Time taken in step #3: 1.
- *Iteration #4:* Since the new first element of the calling order is same as the ideal order, that process will be executed. Time taken in step #4: 1.
- *Iteration #5:* Since the last element of the calling order is same as the ideal order, that process will be executed. Time taken in step #5: 1.

Total time taken: 5 units.

PS: Executing a process takes 1 unit of time. Changing the position takes 1 unit of time.

Input format:
The first line a number **N**, denoting the number of processes. The second line contains the calling order of the processes. The third line contains the ideal order of the processes.

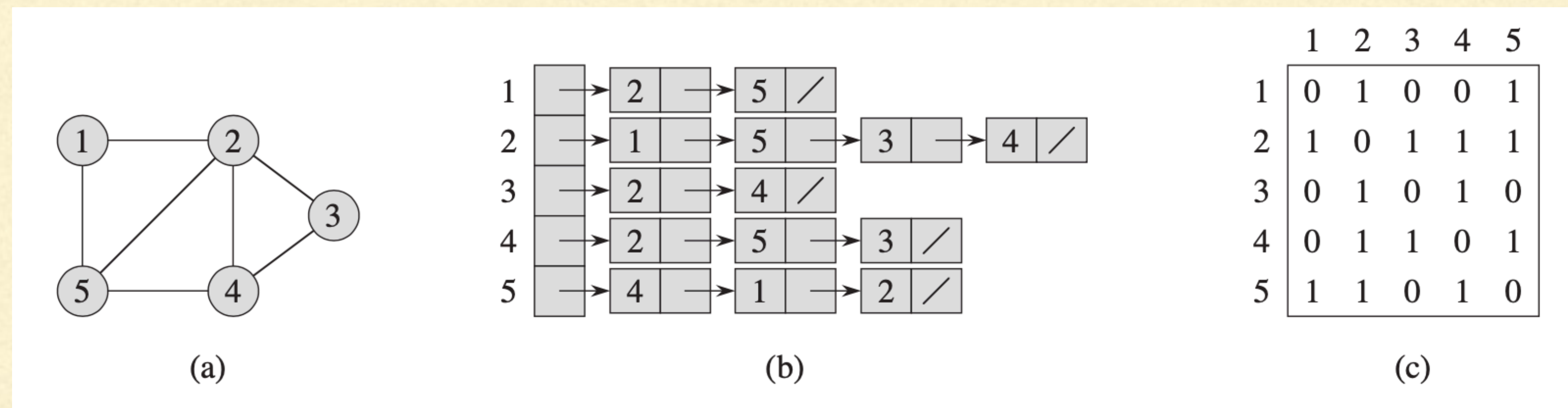
Output format:
Print the total time taken for the entire queue of processes to be executed.

Constraints:
1<=**N**<=100

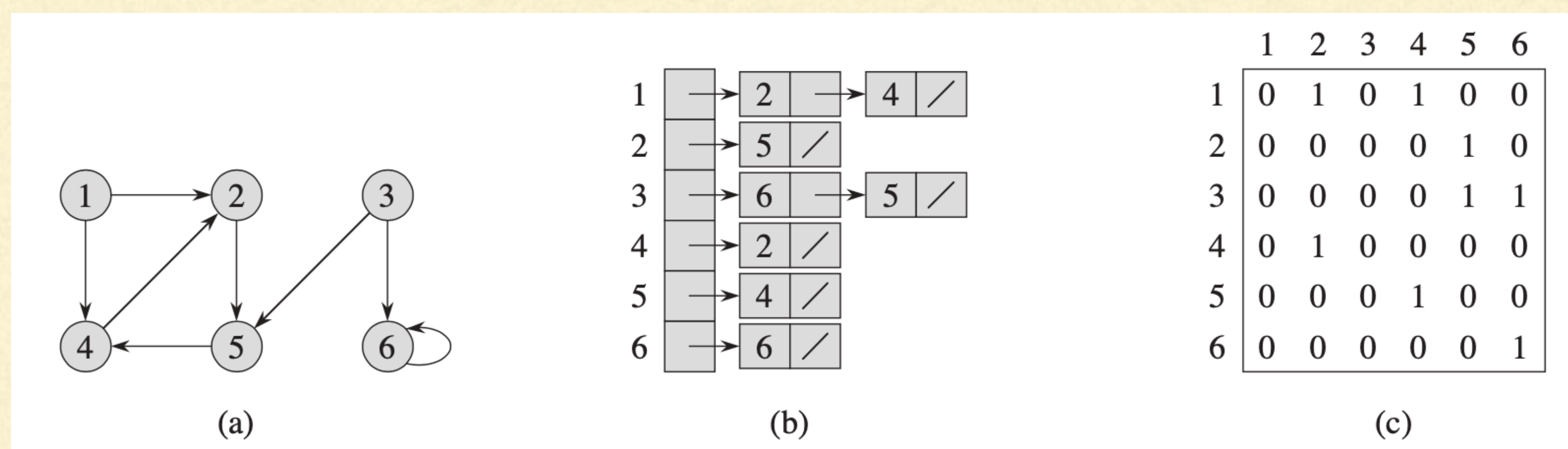
SAMPLE INPUT	SAMPLE OUTPUT
3 3 2 1 1 3 2	5

```python
inputFile = open('InputPS4.txt','r')
fileLines = inputFile.readlines()
n = int(fileLines[0])
calling = [int(i) for i in fileLines[1].split(" ")]
ideal = [int(i) for i in fileLines[2].split(" ")]
i=0
counter=0
while len(calling) != 0:
        cur = calling.pop(0)
        if cur != ideal[i]:
                calling.append(cur)
        else:
                i+=1
        counter+=1
print(counter)
```

GRAPH



Undirected graph



Directed graph

PS28 IPL Bench

Assignment 1 – PS28 - [IPL Bench] - [Weightage 12%]

1. Problem Statement

In this Problem, you have to write an application in Python 3.7 that maps IPL franchises and players as per the below guidelines.

Assume that you are a news reporter and you want to map which players have been associated with a franchise (either in the past or present). For this you need to have some system of storing these players and the franchise they have been with. Assume that you have a list of N franchises and M players. For the sake of this assignment, let us assume that a particular player could be associated with only two franchises at max.

Model the following problem as a graph based problem. Clearly state how the vertices and edges can be modelled such that this graph can be used to answer the following queries efficiently.

1. List the unique franchises and players the reporter has collected in the system.
2. For a particular player, help the reporter recollect the franchises he has represented.
3. For a particular franchise, list the players that have been associated with it (past or present).
4. Identify if two players are franchise buddies. Player A and Player B are considered to be franchise buddies if they have been associated with the same franchise (not necessarily at the same time or in the same year)
5. Can two players A and B be connected such that there exists another player C where A and C are franchise buddies and C and B are franchise buddies.
6. Perform an analysis for the questions above and give the running time in terms of n.

The basic structure of the graph will be:

```
class IPL:
    PlayerTeam=[] #list containing players and teams
    edges=[[],[]] # matrix of edges/ associations
```

```
class IPL:
    PlayerTeam = []
    edges = [[],[]]
    franchises=[]
    players = []

    def __init__(self):
        self.PlayerTeam = []
        self.edges =[[],[]]

    def readInputFile(self, filename):
        f=open(filename,'r')
        playerList = []
        playernames = set()
        for line in f.readlines():
            line = line.replace('\n', '')
            data = line.split('/')
            self.franchises.append(data[0].strip())
            players = [datum.strip() for datum in data[1:]]
            playerList.append(players)
            playernames.update(set(players))
        self.players = list(playernames)
        self.players.sort()

        num_nodes = len(self.franchises) + len(self.players)
        self.edges = [[0]*num_nodes for i in range(num_nodes)]
        for index, franchise in enumerate(self.franchises):
            for player in playerList[index]:
                j = self.players.index(player)
                self.edges[index][j+len(self.franchises)] =1
                self.edges[j+len(self.franchises)][index]=1
```

PS28

```
def displayAll(self):
        print('--------Function displayAll--------')
        print('Total no. of franchises: '+str(len(self.franchises)))
        print('List of franchises: ')
        print("\n".join(self.franchises))
        print('\nList of players: ')
        print("\n".join(self.players))
```

```
def displayFranchises(self,player):
        print('--------Function displayFranchises --------')
        print('Player name: '+player)
        print('List of Franchises: ')
        if not self.players.__contains__(player):
                print('Player not found')
        else:
                playerIndex = self.players.index(player) + len(self.franchises)
                found = False
                for i in range(len(self.franchises)):
                        if self.edges[playerIndex][i] == 1:
                                found = True
                                print(self.franchises[i],end=' ')
                if not found:
                        print('Player not associated with any franchise', end = ' ')
                print()
```

```
def displayPlayers(self,franchise):
        print('--------Function displayPlayers --------')
        print('Franchise name: '+franchise)
        if not self.franchises.__contains__(franchise):
                print('Franchise not found')
        else:
                franchiseIndex = self.franchises.index(franchise)
                found = False
                for i in range(len(self.players)):
                        j = i+len(self.franchises)
                        if self.edges[franchiseIndex][j] == 1:
                                found = True
                                print(self.players[i], end = ' ')
                if not found:
                        print('Franchise not associated with any player',end=' ')
                print()
```

PS28

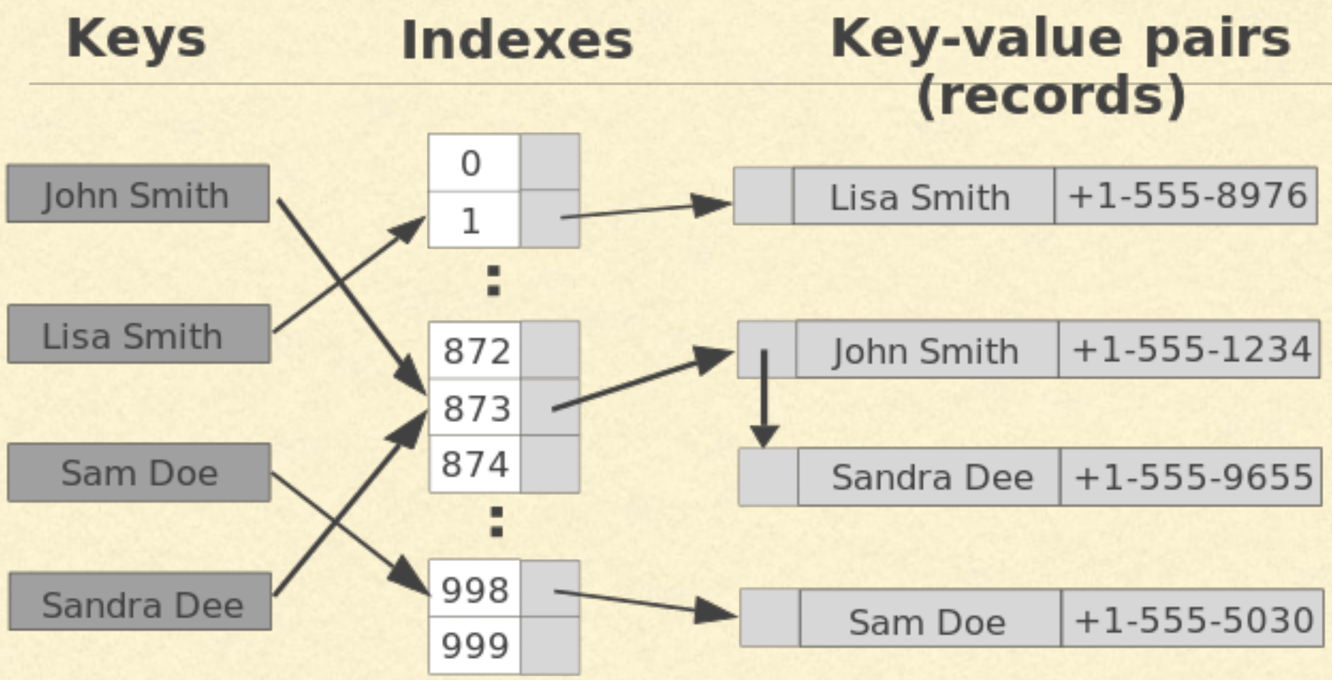
```python
def franchiseBuddies(self,playerA,playerB):
        p1=self.players.index(playerA) + len(self.franchises)
        p2=self.players.index(playerB) + len(self.franchises)
        print('--------Function franchiseBuddies --------')
        print('Player A: '+playerA)
        print('Player B: '+playerB)
        print('Franchise Buddies: ',end="")
        found = False
        for i in range(len(self.franchises)):
                if self.edges[p1][i] == 1 and self.edges[p2][i] == 1:
                        found = True
                        print('Yes, '+self.franchises[i])
        if not found:
                print('No, the players are not franchise buddies')
```

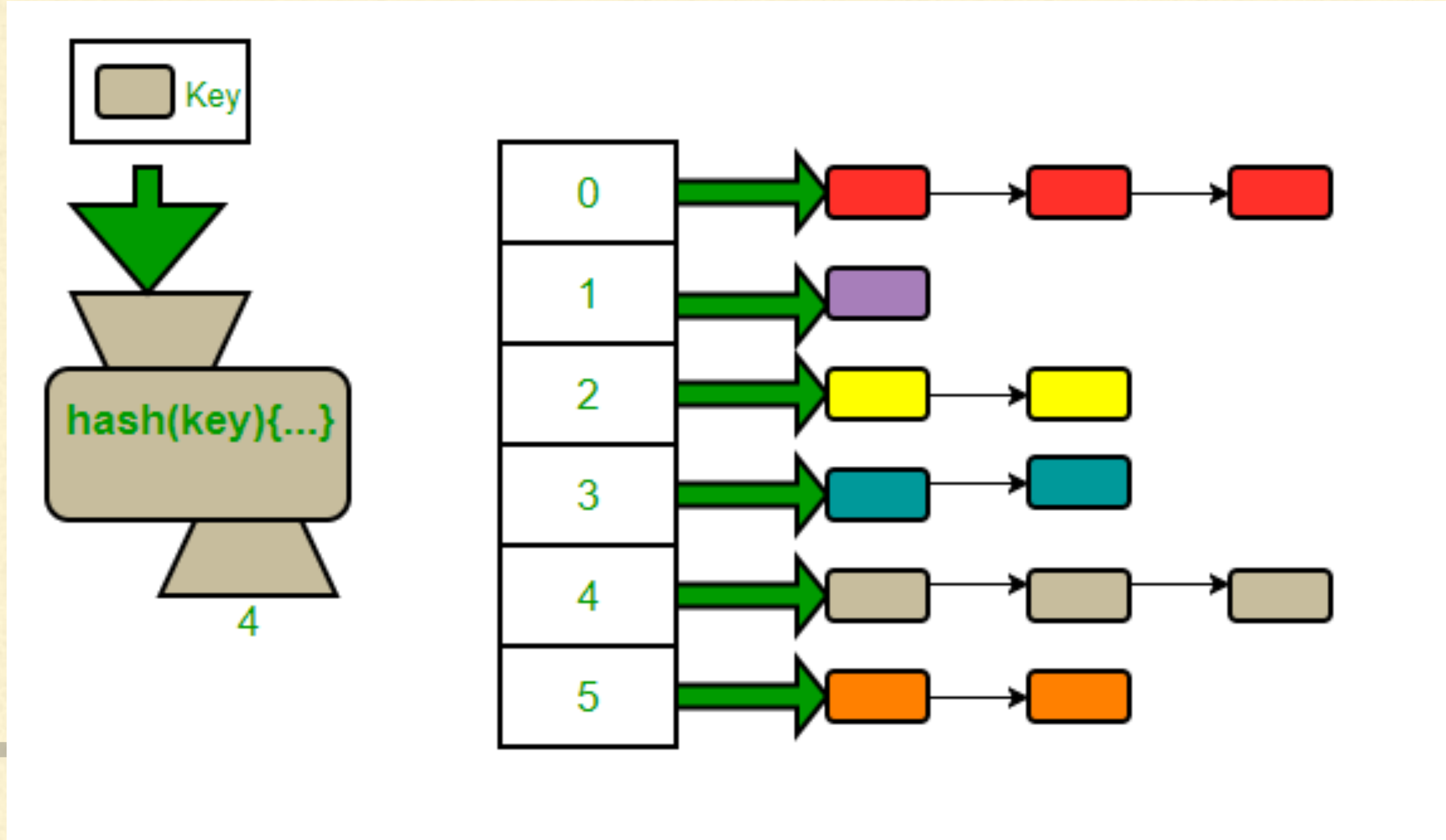
```python
def printPath(self,nodes, parent, i):
        if parent[i] == -1:
                print(nodes[i],end=' > ')
                return
        self.printPath(nodes, parent, parent[i])
        print(nodes[i],end=' > ')

def findPlayerConnect(self, playerA, playerB):
        print('--------Function findPlayerConnect -------')
        print('Player A: '+playerA)
        print('Player B: '+playerB)
        p1 = len(self.franchises) + self.players.index(playerA)
        p2 = len(self.franchises) + self.players.index(playerB)
        parent, dist = dijkstra(self.edges, p1)
        nodes = self.franchises
        nodes.extend(self.players)
        if dist[p2] == 99999:
                print('Related: No')
        else:
                print('Related: Yes,', end=' ')
                self.printPath(nodes, parent, p2)
                print()
```

PS28 Dijkstra's Algorithm

```python
def minDistance(dist, visited):
        min = 999999
        for i in range(len(dist)):
                if dist[i] < min and visited[i] == False:
                        min = dist[i]
                        min_index = i
        return min_index

def printPath(parent,i):
        if parent[i] == -1:
                print(l[i],end =' ')
                return
        printPath(parent, parent[i])
        print(l[i], end =' ')

def dijkstra(adj, src):
        dist = [99999] * len(adj)
        dist[src] = 0
        visited = [False] * len(adj)
        parent = [-1] * len(adj)

        for i in range(len(adj)):
                u = minDistance(dist, visited)
                visited[u] = True
                for v in range(len(adj)):
                        if adj[u][v] !=0 and visited[v] == False and dist[v] > dist[u] + adj[u][v]:
                                dist[v] = dist[u] + adj[u][v]
                                parent[v] = u
        return parent, dist
```

HASH TABLE


Keys
Indexes
Key-value pairs (records)
John Smith
Lisa Smith
Sam Doe
Sandra Dee
0
1
872
873
874
998
999
Lisa Smith
+1-555-8976
John Smith
+1-555-1234
Sandra Dee
+1-555-9655
Sam Doe
+1-555-5030


Key
hash(key){...}
4
0
1
2
3
4
5