પ્રશ્ન 1(a) [3 ગુણ]

Time Complexity માટે best case, worst case અને average case વ્યાખ્યાયિત કરો.

જવાબ:

ટેબલ: Time Complexity Cases

કેસનો પ્રકાર	વ્યાખ્યા	ઉદાહરણ
Best Case	એલ્ગોરિધમ execution માટે લઘુત્તમ સમય	Linear search માં એલિમેન્ટ પહેલી પોઝિશન પર મળે
Worst Case	એલ્ગોરિધમ execution માટે મહત્તમ સમય	Linear search માં એલિમેન્ટ છેલ્લી પોઝિશન પર મળે
Average Case	સામાન્ય input scenarios માટે અપેક્ષિત સમય	Linear search માં એલિમેન્ટ મધ્યમાં મળે

- **Best Case**: આદર્શ input conditions સાથે એલ્ગોરિધમ optimal પ્રદર્શન આપે
- **Worst Case**: પ્રતિકૂળ input સાથે એલ્ગોરિધમ મહત્તમ સમય લે
- **Average Case**: બધા શક્ય inputs માં execution time ની ગાણિતિક અપેક્ષા

મેમરી ટ્રીક: "BWA - Best, Worst, Average"

પ્રશ્ન 1(b) [4 ગુણ]

OOP માં Class અને Object શું છે? યોગ્ય ઉદાહરણ આપો.

જવાબ:

ટેબલ: Class vs Object

પાસું	Class	Object
વ્યાખ્યા	Objects બનાવવા માટે blueprint/template	Class નું instance
મેમરી	કોઈ મેમરી allocate નથી થતી	બનાવવામાં આવે ત્યારે મેમરી allocate થાય
ઉદાહરણ	Car (template)	my_car = Car()

```
# Class definition
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print(f"Name: {self.name}, Age: {self.age}")

# Object creation
student1 = Student("John", 20)
student1.display()
```

- **Class**: Attributes અને methods વ્યાખ્યાયિત કરતું template
- **Object**: વાસ્તવિક values સાથેનું instance

મેમરી ટ્રીક: "Class = Cookie Cutter, Object = વાસ્તવિક Cookie"

પ્રશ્ન 1(c) [7 ગુણ]

Simple nested loop અને numpy module નો ઉપયોગ કરીને બે matrix multiplication માટે પ્રોગ્રામ લખો.

જવાબ:

```
# Method 1: Simple Nested Loop નો ઉપયોગ
def matrix_multiply_nested(A, B):
    rows_A, cols_A = len(A), len(A[0])
    rows_B, cols_B = len(B), len(B[0])

    # Result matrix initialize કરો
    result = [[0 for _ in range(cols_B)] for _ in range(rows_A)]

    # Matrix multiplication
    for i in range(rows_A):
        for j in range(cols_B):
            for k in range(cols_A):
                result[i][j] += A[i][k] * B[k][j]

    return result

# Method 2: NumPy નો ઉપયોગ
import numpy as np

def matrix_multiply_numpy(A, B):
    A_np = np.array(A)
    B_np = np.array(B)
    return np.dot(A_np, B_np)

# ઉદાહરણ
A = [[1, 2], [3, 4]]
B = [[5, 6], [7, 8]]

print("Nested Loop Result:", matrix_multiply_nested(A, B))
print("NumPy Result:", matrix_multiply_numpy(A, B))
```

- **Nested Loop**: Row, column અને multiplication માટે ત્રણ loops
- **NumPy**: કાર્યક્ષમ multiplication માટે built-in dot() function

મેમરી ટ્રીક: "Row × Column = Result"

પ્રશ્ન 1(c) OR [7 ગુણ]

Array ના basic operations માટે એક પ્રોગ્રામ લખો.

જવાબ:

```
import array

# Array બનાવો
arr = array.array('i', [1, 2, 3, 4, 5])

def array_operations():
    print("મૂળ array:", arr)

    # Element insert કરો
    arr.insert(2, 10)
    print("insert(2, 10) પછી:", arr)

    # Element append કરો
    arr.append(6)
    print("append(6) પછી:", arr)

    # Element remove કરો
    arr.remove(10)
    print("remove(10) પછી:", arr)

    # Element pop કરો
    popped = arr.pop()
    print(f"Pop કરેલું element: {popped}, Array: {arr}")

    # Element શોધો
    index = arr.index(3)
    print(f"3 નું index: {index}")

    # Occurrences ગણો
    count = arr.count(2)
    print(f"2 નું count: {count}")

array_operations()
```

ટેબલ: Array Operations

Operation	Method	વર્ણન
Insert	insert(index, value)	ચોક્કસ position પર element ઉમેરવું
Append	append(value)	છેડે element ઉમેરવું
Remove	remove(value)	પહેલું occurrence દૂર કરવું
Pop	pop()	છેલ્લું element દૂર કરીને return કરવું

મેમરી ટ્રીક: "IARP - Insert, Append, Remove, Pop"

પ્રશ્ન 2(a) [3 ગુણ]

Big 'O' Notation સમજાવો.

જવાબ:

ટેબલ: Big O Complexity

Notation	નામ	ઉદાહરણ
O(1)	Constant	Array access
O(n)	Linear	Linear search
O(n²)	Quadratic	Bubble sort
O(log n)	Logarithmic	Binary search

- **Big O**: એલ્ગોરિધમની time complexity ની upper bound વર્ણવે છે
- **હેતુ**: વિવિધ એલ્ગોરિધમની કાર્યક્ષમતાની તુલના કરવી
- **ધ્યાન**: Worst-case scenario analysis પર

મેમરી ટ્રીક: "Big O = વૃદ્ધિના Big Order"

પ્રશ્ન 2(b) [4 ગુણ]

Class method અને static method વચ્ચે તફાવત લખી સમજાવો.

જવાબ:

ટેબલ: Method Types Comparison

પાસું	Class Method	Static Method
Decorator	@classmethod	@staticmethod
પહેલું Parameter	cls (class reference)	કોઈ ખાસ parameter નહીં
Access	Class variables ને access કરી શકે	Class/instance variables ને access કરી શકતું નથી
ઉપયોગ	Alternative constructors	Utility functions

```
class MyClass:
    class_var = "હું class variable છું"

    @classmethod
    def class_method(cls):
        return f"Class method accessing: {cls.class_var}"

    @staticmethod
    def static_method():
        return "Static method - કોઈ class access નથી"

# ઉપયોગ
```

```
print(MyClass.class_method())
print(MyClass.static_method())
```

મેમરી ટ્રીક: "Class method માં CLS છે, Static method STandalone છે"

પ્રશ્ન 2(c) [7 ગુણ]

Public અને private type derivation નો ઉપયોગ કરીને single level inheritance માટે class બનાવો.

જવાબ:

```
# Base class
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand          # Public attribute
        self._model = model         # Protected attribute
        self.__year = 2023          # Private attribute

    def start_engine(self):
        return f"{self.brand} engine શરૂ થયું"

    def _display_model(self):       # Protected method
        return f"Model: {self._model}"

    def __private_method(self):     # Private method
        return f"Year: {self.__year}"

# Derived class (Single level inheritance)
class Car(Vehicle):
    def __init__(self, brand, model, doors):
        super().__init__(brand, model)
        self.doors = doors

    def car_info(self):
        # Public અને protected members ને access કરી શકે
        return f"Car: {self.brand}, {self._display_model()}, Doors: {self.doors}"

    def demonstrate_access(self):
        print("Public access:", self.brand)
        print("Protected access:", self._model)
        # print("Private access:", self.__year)  # આ error આપશે

# ઉપયોગ
my_car = Car("Toyota", "Camry", 4)
print(my_car.car_info())
print(my_car.start_engine())
my_car.demonstrate_access()
```

- **Public**: બધે accessible (brand)
- **Protected**: Class અને subclasses માં accessible (_model)

- **Private**: માત્ર સમાન class માં accessible (__year)

મેમરી ટ્રીક: "Public = બધા, Protected = કુટુંબ, Private = વ્યક્તિગત"

પ્રશ્ન 2(a) OR [3 ગુણ]

Constructor ને ઉદાહરણ સાથે સમજાવો.

જવાબ:

ટેબલ: Constructor Types

પ્રકાર	Method	હેતુ
Default	`__init__(self)`	Default values સાથે initialize
Parameterized	`__init__(self, params)`	Custom values સાથે initialize

```
class Student:
    def __init__(self, name="અજ્ઞાત", age=18):  # Constructor
        self.name = name
        self.age = age
        print(f"Student {name} બનાવ્યો")

    def display(self):
        print(f"નામ: {self.name}, ઉંમર: {self.age}")

# Object creation automatically constructor ને call કરે છે
s1 = Student("Alice", 20)
s2 = Student()  # Default values ઉપયોગ કરે
```

- **Constructor**: Object બનાવવામાં આવે ત્યારે call થતી special method
- **હેતુ**: Object attributes ને initialize કરવા
- **Automatic**: Object creation દરમિયાન automatically call થાય

મેમરી ટ્રીક: "Constructor = Object નું જન્મ પ્રમાણપત્ર"

પ્રશ્ન 2(b) OR [4 ગુણ]

Polymorphism દર્શાવવા માટે એક પ્રોગ્રામ લખો.

જવાબ:

```
# Base class
class Animal:
    def make_sound(self):
        pass

# Derived classes
class Dog(Animal):
    def make_sound(self):
```

```
        return "ભોંભોં!"

class Cat(Animal):
    def make_sound(self):
        return "મિયાઉં!"

class Cow(Animal):
    def make_sound(self):
        return "હંભા!"

# Polymorphism demonstration
def animal_sound(animal):
    return animal.make_sound()

# Objects બનાવવા
animals = [Dog(), Cat(), Cow()]

# સમાન method call, અલગ behavior
for animal in animals:
    print(f"{animal.__class__.__name__}: {animal_sound(animal)}")
```

ટેબલ: Polymorphism ના ફાયદા

ફાયદો	વર્ણન
લવચીકતા	સમાન interface, અલગ implementations
જાળવણી	નવા types ઉમેરવા સહેલા
Code Reuse	વિવિધ objects માટે સામાન્ય interface

મેમરી ટ્રીક: "Poly = ઘણા, Morph = સ્વરૂપો"

પ્રશ્ન 2(c) OR [7 ગુણ]

Multiple અને hierarchical inheritance નો ઉપયોગ કરી પાયથોન પ્રોગ્રામ લખો.

જવાબ:

```
# Multiple Inheritance
class Teacher:
    def __init__(self, subject):
        self.subject = subject

    def teach(self):
        return f"{self.subject} શીખવાડવું"

class Researcher:
    def __init__(self, field):
        self.field = field
```

```
    def research(self):
        return f"{self.field} માં સંશોધન કરવું"

# Multiple inheritance
class Professor(Teacher, Researcher):
    def __init__(self, name, subject, field):
        self.name = name
        Teacher.__init__(self, subject)
        Researcher.__init__(self, field)

    def profile(self):
        return f"પ્રોફેસર {self.name}: {self.teach()} અને {self.research()}"

# Hierarchical Inheritance
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

    def start(self):
        return f"{self.brand} શરૂ થયું"

class Car(Vehicle):
    def drive(self):
        return f"{self.brand} કાર ચાલે છે"

class Bike(Vehicle):
    def ride(self):
        return f"{self.brand} બાઇક ચાલે છે"

# ઉપયોગ
prof = Professor("સ્મિથ", "પાયથોન", "AI")
print(prof.profile())

car = Car("Honda")
bike = Bike("Yamaha")
print(car.drive())
print(bike.ride())
```

આકૃતિ:

```
Multiple Inheritance:    Hierarchical Inheritance:
   Teacher  Researcher          Vehicle
      \        /                 /    \
       \      /                Car    Bike
      Professor
```

મેમરી ટ્રીક: "Multiple = ઘણા માતા-પિતા, Hierarchical = વૃક્ષ માળખું"

પ્રશ્ન 3(a) [3 ગુણ]

Stack પર Push અને Pop operations સમજાવો.

જવાબ:

ટેબલ: Stack Operations

Operation	વર્ણન	Time Complexity
Push	ટોચ પર element ઉમેરવું	O(1)
Pop	ટોચેથી element દૂર કરવું	O(1)
Peek/Top	ટોચનું element જોવું	O(1)
isEmpty	Stack ખાલી છે કે નહીં તપાસવું	O(1)

```
stack = []

# Push operation
stack.append(10)  # 10 Push કરો
stack.append(20)  # 20 Push કરો
print("Push પછી:", stack)  # [10, 20]

# Pop operation
item = stack.pop()  # 20 Pop કરો
print(f"Pop કર્યું: {item}, Stack: {stack}")  # [10]
```

- **LIFO**: Last In, First Out સિદ્ધાંત
- **ટોચ**: Operations માટે માત્ર accessible element

મેમરી ટ્રીક: "Stack = થાળીઓનો ઢગલો - છેલ્લી થાળી અંદર, પહેલી થાળી બહાર"

પ્રશ્ન 3(b) [4 ગુણ]

Queue ના Enqueue અને Dequeue operations સમજાવો.

જવાબ:

ટેબલ: Queue Operations

Operation	વર્ણન	સ્થાન	Time Complexity
Enqueue	Element ઉમેરવું	પાછળ	O(1)
Dequeue	Element દૂર કરવું	આગળ	O(1)
Front	આગળનું element જોવું	આગળ	O(1)
Rear	પાછળનું element જોવું	પાછળ	O(1)

```
from collections import deque

queue = deque()

# Enqueue operation
queue.append(10)  # 10 Enqueue કરો
queue.append(20)  # 20 Enqueue કરો
print("Enqueue પછી:", list(queue))  # [10, 20]

# Dequeue operation
item = queue.popleft()  # 10 Dequeue કરો
print(f"Dequeue કર્યું: {item}, Queue: {list(queue)}")  # [20]
```

- **FIFO**: First In, First Out સિદ્ધાંત
- **બે છેડા**: આગળ removal માટે, પાછળ insertion માટે

મેમરી ટ્રીક: "Queue = દુકાનમાં લાઇન - પહેલો વ્યક્તિ અંદર, પહેલો વ્યક્તિ બહાર"

પ્રશ્ન 3(c) [7 ગુણ]

Stack ની વિવિધ applications સમજાવો.

જવાબ:

ટેબલ: Stack Applications

Application	વર્ણન	ઉદાહરણ
Expression Evaluation	Infix ને postfix માં રૂપાંતર	(a+b)c → *ab+c*
Function Calls	Function call sequence manage કરવું	Recursion handling
Undo Operations	તાજેતરની ક્રિયાઓ ઉલટાવવી	Text editor undo
Browser History	Pages દ્વારા પાછળ navigate કરવું	Back button
Parentheses Matching	Balanced brackets ચકાસવા	{[()]} validation

```
# ઉદાહરણ: Parentheses matching
def is_balanced(expression):
    stack = []
    pairs = {'(': ')', '[': ']', '{': '}'}

    for char in expression:
        if char in pairs:  # Opening bracket
            stack.append(char)
        elif char in pairs.values():  # Closing bracket
            if not stack:
                return False
            if pairs[stack.pop()] != char:
                return False
```

```
    return len(stack) == 0

# ટેસ્ટ
print(is_balanced("({[]})"))  # True
print(is_balanced("({[}])"))  # False
```

- **Memory Management**: Programming માં function call stack
- **Backtracking**: Maze solving, game algorithms
- **Compiler Design**: Syntax analysis અને parsing

મેમરી ટ્રીક: "Stack Applications = UFPB (Undo, Function, Parentheses, Browser)"

પ્રશ્ન 3(a) OR [3 ગુણ]

Single Queue ની મર્યાદાઓની યાદી બનાવો.

જવાબ:

ટેબલ: Single Queue Limitations

મર્યાદા	વર્ણન	સમસ્યા
Memory Wastage	આગળની જગ્યા અનુપયોગી બને છે	અકાર્યક્ષમ memory ઉપયોગ
Fixed Size	Dynamically resize કરી શકાતું નથી	જગ્યાની મર્યાદા
False Overflow	આગળની જગ્યા ખાલી હોવા છતાં queue ભરેલી લાગે	અકાળે capacity limit
No Reuse	Dequeue કરેલી positions ફરીથી વાપરી શકાતી નથી	Linear space utilization

```
Single Queue Problem:
Front               Rear
  ↓                  ↓
[X][X][3][4][5][ ][ ][ ]
 ↑  ↑
બગાડેલી જગ્યા (ફરીથી વાપરી શકાતી નથી)
```

- **Linear Implementation**: Dequeue કરેલી જગ્યા utilize કરી શકાતી નથી
- **Static Array**: Fixed size allocation

મેમરી ટ્રીક: "Single Queue = એક બાજુનો રસ્તો (પાછા ફરી શકાતા નથી)"

પ્રશ્ન 3(b) OR [4 ગુણ]

Circular અને simple queues નો તફાવત લખી સમજાવો.

જવાબ:

ટેબલ: Queue Types Comparison

પાસું	Simple Queue	Circular Queue
Memory Usage	Linear, wasteful	Circular, efficient
Space Reuse	Dequeue કરેલી જગ્યા ફરીથી વાપરાતી નથી	બધી positions ફરીથી વાપરે છે
Overflow	False overflow શક્ય	માત્ર true overflow
Implementation	Front અને rear pointers	Modulo સાથે front અને rear

```
Simple Queue:          Circular Queue:
[X][X][3][4][ ][ ]    [5][6][3][4]
Front→    Rear→        ↑Rear Front→
(બગાડેલી જગ્યા)       (જગ્યા ફરીથી વાપરાય)
```

```python
# Circular Queue Implementation
class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.front = -1
        self.rear = -1

    def enqueue(self, item):
        if (self.rear + 1) % self.size == self.front:
            print("Queue ભરાઈ ગયું")
            return
        if self.front == -1:
            self.front = 0
        self.rear = (self.rear + 1) % self.size
        self.queue[self.rear] = item

    def dequeue(self):
        if self.front == -1:
            print("Queue ખાલી છે")
            return None
        item = self.queue[self.front]
        if self.front == self.rear:
            self.front = self.rear = -1
        else:
            self.front = (self.front + 1) % self.size
        return item
```

મેમરી ટ્રીક: "Circular = Ring Road (સતત), Simple = મૃત અંતનો રસ્તો"

પ્રશ્ન 3(c) OR [7 ગુણ]

નીચેની infix expression ને postfix માં રૂપાંતર કરો: (a * b) * (c ^ (d + e) – f)

જવાબ:

કોષ્ટક: Operator Precedence

Operator	Precedence	Associativity
^	3	Right to Left
*, /	2	Left to Right
+, -	1	Left to Right

Step-by-step conversion:

```
Expression: (a * b) * (c ^ (d + e) – f)

Step 1: (a * b) → ab*
Step 2: (d + e) → de+
Step 3: c ^ (de+) → c de+ ^
Step 4: (c de+ ^) – f → c de+ ^ f –
Step 5: (ab*) * (c de+ ^ f –) → ab* c de+ ^ f – *

અંતિમ જવાબ: ab*cde+^f-*
```

Algorithm:

1. **Operand**: Output માં ઉમેરો
2. **'('**: Stack માં push કરો
3. **')'**: '(' આવે ત્યાં સુધી pop કરો
4. **Operator**: Higher/equal precedence pop કરો, પછી push કરો
5. **અંત**: બાકીના બધા operators pop કરો

```python
def infix_to_postfix(expression):
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    stack = []
    output = []

    for char in expression:
        if char.isalnum():
            output.append(char)
        elif char == '(':
            stack.append(char)
        elif char == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()  # '(' દૂર કરો
        elif char in precedence:
            while (stack and stack[-1] != '(' and
                   stack[-1] in precedence and
                   precedence[stack[-1]] >= precedence[char]):
                output.append(stack.pop())
```

```
            stack.append(char)

    while stack:
        output.append(stack.pop())

    return ''.join(output)

# ટેસ્ટ
result = infix_to_postfix("(a*b)*(c^(d+e)-f)")
print("Postfix:", result)  # ab*cde+^f-*
```

મેમરી ટ્રીક: "Precedence માટે PEMDAS, Operators માટે Stack"

પ્રશ્ન 4(a) [3 ગુણ]

Linked List ના પ્રકારો સમજાવો.

જવાબ:

ટેબલ: Linked List Types

પ્રકાર	વર્ણન	મુખ્ય વિશેષતા
Singly Linked	Next node નો એક pointer	માત્ર આગળ traversal
Doubly Linked	Next અને previous નાં pointers	બંને દિશામાં traversal
Circular Linked	છેલ્લો node પહેલાને point કરે	કોઈ NULL pointer નથી
Doubly Circular	Doubly + Circular features	બંને દિશા + circular

```
Singly:     [A]→[B]→[C]→NULL

Doubly:     NULL←[A]⇄[B]⇄[C]→NULL

Circular:   [A]→[B]→[C]
             ↑_________|

Doubly Circular: [A]⇄[B]⇄[C]
                  ↑_________|
```

- **Memory**: દરેક node માં data અને pointer(s) હોય છે
- **Dynamic**: Runtime દરમિયાન size બદલાઈ શકે છે

મેમરી ટ્રીક: "SDCD - Singly, Doubly, Circular, Doubly-Circular"

પ્રશ્ન 4(b) [4 ગુણ]

Circular linked list અને singly linked list નો તફાવત લખી સમજાવો.

જવાબ:

ટેબલ: Singly vs Circular Linked List

પાસું	Singly Linked List	Circular Linked List
છેલ્લો Node	NULL ને point કરે	પહેલા node ને point કરે
Traversal	NULL પર અંત આવે	સતત loop
Memory	છેલ્લો node NULL store કરે	કોઈ NULL pointer નથી
Detection	NULL માટે check કરો	શરૂઆતના node માટે check કરો

```
# Singly Linked List Node
class SinglyNode:
    def __init__(self, data):
        self.data = data
        self.next = None

# Circular Linked List Node
class CircularNode:
    def __init__(self, data):
        self.data = data
        self.next = None

def traverse_singly(head):
    current = head
    while current:  # NULL પર અટકે
        print(current.data)
        current = current.next

def traverse_circular(head):
    if not head:
        return
    current = head
    while True:
        print(current.data)
        current = current.next
        if current == head:  # શરૂઆતમાં પાછા
            break
```

આકૃતિ:

```
Singly:    [1]→[2]→[3]→NULL

Circular:  [1]→[2]→[3]
            ↑_____|
```

મેમરી ટ્રીક: "Singly = મૃત અંત, Circular = Race Track"

પ્રશ્ન 4(c) [7 ગુણ]

Singly linked list માં નીચેની કામગીરી કરવા માટે એક પ્રોગ્રામનો અમલ કરો:
a. Singly linked list ની શરૂઆતમાં node દાખલ કરો.
b. Singly linked list ના અંતે node દાખલ કરો.

જવાબ:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def insert_at_beginning(self, data):
        """શરૂઆતમાં node insert કરો"""
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
        print(f"{data} શરૂઆતમાં insert કર્યું")

    def insert_at_end(self, data):
        """અંતે node insert કરો"""
        new_node = Node(data)

        if not self.head:  # ખાલી list
            self.head = new_node
            print(f"{data} અંતે insert કર્યું (પહેલો node)")
            return

        # છેલ્લા node સુધી traverse કરો
        current = self.head
        while current.next:
            current = current.next

        current.next = new_node
        print(f"{data} અંતે insert કર્યું")

    def display(self):
        """Linked list દર્શાવો"""
        if not self.head:
            print("List ખાલી છે")
            return

        current = self.head
        elements = []
        while current:
            elements.append(str(current.data))
            current = current.next

        print(" → ".join(elements) + " → NULL")
```

```
# ઉપયોગનું ઉદાહરણ
sll = SinglyLinkedList()

# શરૂઆતમાં insert કરો
sll.insert_at_beginning(10)
sll.insert_at_beginning(20)
sll.display()  # 20 → 10 → NULL

# અંતે insert કરો
sll.insert_at_end(30)
sll.insert_at_end(40)
sll.display()  # 20 → 10 → 30 → 40 → NULL
```

ટેબલ: Insertion Operations

Operation	Time Complexity	પગલાં
શરૂઆત	O(1)	1. Node બનાવો 2. Head ને point કરો 3. Head update કરો
અંત	O(n)	1. Node બનાવો 2. અંત સુધી traverse કરો 3. છેલ્લો node link કરો

મેમરી ટ્રીક: "શરૂઆત = ઝડપથી (O(1)), અંત = પ્રવાસ (O(n))"

પ્રશ્ન 4(a) OR [3 ગુણ]

Doubly linked list સમજાવો.

જવાબ:

ટેબલ: Doubly Linked List Features

વિશેષતા	વર્ણન
બે Pointers	દરેક node માં prev અને next
Bidirectional	આગળ અને પાછળ બંને તરફ traverse કરી શકાય
Memory	prev pointer માટે વધારાની જગ્યા
લવચીકતા	ગમે ત્યાં insertion/deletion સહેલું

```
class DoublyNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

# માળખાનું visualization
# NULL ← [prev|data|next] ⇄ [prev|data|next] → NULL
```

```
Doubly Linked List માળખું:
NULL←[A]⇄[B]⇄[C]→NULL
     prev next
```

- **ફાયદા**: Bidirectional traversal, સહેલું deletion
- **નુકસાન**: prev pointer માટે વધારાની મેમરી

મેમરી ટ્રીક: "Doubly = બે બાજુનો રસ્તો"

પ્રશ્ન 4(b) OR [4 ગુણ]

Linked List ની applications નું વર્ણન કરો.

જવાબ:

ટેબલ: Linked List Applications

Application	Use Case	ફાયદો
Dynamic Arrays	જ્યારે size બદલાતું રહે	કાર્યક્ષમ memory usage
Stack/Queue	LIFO/FIFO operations	Dynamic size
Graphs	Adjacency list representation	Space efficient
Music Playlist	પાછલા/આગલા ગીતો	સહેલું navigation
Browser History	Back/Forward navigation	Dynamic history
Undo Operations	Text editors	કાર્યક્ષમ undo/redo

```python
# ઉદાહરણ: Doubly Linked List વાપરીને Browser History
class Page:
    def __init__(self, url):
        self.url = url
        self.prev = None
        self.next = None

class BrowserHistory:
    def __init__(self):
        self.current = None

    def visit(self, url):
        page = Page(url)
        if self.current:
            self.current.next = page
            page.prev = self.current
        self.current = page

    def back(self):
        if self.current and self.current.prev:
```

```
            self.current = self.current.prev
            return self.current.url
        return "કોઈ પાછલું page નથી"

    def forward(self):
        if self.current and self.current.next:
            self.current = self.current.next
            return self.current.url
        return "કોઈ આગલું page નથી"
```

મેમરી ટ્રીક: "Linked Lists = Dynamic, લવચીક, જોડાયેલ"

પ્રશ્ન 4(c) OR [7 ગુણ]

Merge Sort algorithm નો પ્રોગ્રામ લખી સમજાવો.

જવાબ:

```
def merge_sort(arr):
    """Merge Sort implementation"""
    if len(arr) <= 1:
        return arr

    # Array ને બે ભાગમાં વહેંચો
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # બંને ભાગો recursively sort કરો
    left_sorted = merge_sort(left_half)
    right_sorted = merge_sort(right_half)

    # Sorted ભાગોને merge કરો
    return merge(left_sorted, right_sorted)

def merge(left, right):
    """બે sorted arrays ને merge કરો"""
    result = []
    i = j = 0

    # Elements compare કરીને merge કરો
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # બાકીના elements ઉમેરો
    result.extend(left[i:])
```

```
    result.extend(right[j:])

    return result

# ઉદાહરણ
def demonstrate_merge_sort():
    arr = [64, 34, 25, 12, 22, 11, 90]
    print("મૂળ array:", arr)

    sorted_arr = merge_sort(arr)
    print("Sorted array:", sorted_arr)

demonstrate_merge_sort()
```

ટેબલ: Merge Sort Analysis

પાસું	મૂલ્ય
Time Complexity	O(n log n)
Space Complexity	O(n)
Stability	Stable
પ્રકાર	Divide and Conquer

Algorithm પગલાં:

1. **વિભાજન**: Array ને બે ભાગમાં વહેંચો
2. **જીત**: બંને ભાગો recursively sort કરો
3. **જોડાણ**: Sorted ભાગોને merge કરો

મેમરી ટ્રીક: "Merge Sort = વહેંચો, ગોઠવો, જોડો"

પ્રશ્ન 5(a) [3 ગુણ]

Binary tree ની applications નું વર્ણન કરો.

જવાબ:

ટેબલ: Binary Tree Applications

Application	વર્ણન	ઉદાહરણ
Expression Trees	ગાણિતિક expression representation	(a+b)*c
Decision Trees	AI/ML માં decision making	Classification algorithms
File Systems	Directory structure organization	Folder hierarchy
Database Indexing	કાર્યક્ષમ searching માટે B-trees	Database indices
Huffman Coding	Data compression technique	File compression
Heap Operations	Priority queues implementation	Task scheduling

- **વંશવેલો Data**: Tree-like structures ને કુદરતી રીતે represent કરે
- **કાર્યક્ષમ Search**: Binary search trees O(log n) operations આપે
- **Memory Management**: Compiler design માં syntax trees માટે વપરાય

મેમરી ટ્રીક: "Binary Trees = EDFDHH (Expression, Decision, File, Database, Huffman, Heap)"

પ્રશ્ન 5(b) [4 ગુણ]

ઉદાહરણ સાથે binary tree ની Indegree અને Outdegree સમજાવો.

જવાબ:

ટેબલ: Degree Definitions

શબ્દ	વ્યાખ્યા	Binary Tree Value
Indegree	Node માં આવતા edges ની સંખ્યા	0 (root) અથવા 1 (બાકી)
Outdegree	Node માંથી જતા edges ની સંખ્યા	0, 1, અથવા 2
Degree	Node સાથે જોડાયેલા કુલ edges	Indegree + Outdegree

```
Binary Tree ઉદાહરણ:
        A (indegree=0, outdegree=2)
       / \
      B   C (indegree=1, outdegree=1)
     /   /
    D   E (indegree=1, outdegree=0)
```

- **Root Node**: હંમેશા indegree = 0
- **Leaf Nodes**: હંમેશા outdegree = 0
- **Internal Nodes**: outdegree = 1 અથવા 2

ટેબલ: ઉદાહરણ Analysis

Node	Indegree	Outdegree	Node Type
A	0	2	Root
B	1	1	Internal
C	1	1	Internal
D	1	0	Leaf
E	1	0	Leaf

મેમરી ટ્રીક: "In = અંદર આવતા, Out = બહાર જતા"

પ્રશ્ન 5(c) [7 ગુણ]

Binary search tree બનાવવા માટે પ્રોગ્રામ લખો.

જવાબ:

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, data):
        """BST માં node insert કરો"""
        if self.root is None:
            self.root = TreeNode(data)
        else:
            self._insert_recursive(self.root, data)

    def _insert_recursive(self, node, data):
        if data < node.data:
            if node.left is None:
                node.left = TreeNode(data)
            else:
                self._insert_recursive(node.left, data)
        elif data > node.data:
            if node.right is None:
                node.right = TreeNode(data)
            else:
                self._insert_recursive(node.right, data)

    def search(self, data):
        """BST માં node શોધો"""
        return self._search_recursive(self.root, data)
```

```
    def _search_recursive(self, node, data):
        if node is None or node.data == data:
            return node

        if data < node.data:
            return self._search_recursive(node.left, data)
        else:
            return self._search_recursive(node.right, data)

    def inorder_traversal(self):
        """Inorder traversal (ડાબે, રૂટ, જમણે)"""
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, node, result):
        if node:
            self._inorder_recursive(node.left, result)
            result.append(node.data)
            self._inorder_recursive(node.right, result)

    def display_tree(self):
        """સાદું tree display"""
        if self.root:
            self._display_recursive(self.root, 0)

    def _display_recursive(self, node, level):
        if node:
            self._display_recursive(node.right, level + 1)
            print("   " * level + str(node.data))
            self._display_recursive(node.left, level + 1)

# ઉદાહરણ
bst = BinarySearchTree()
values = [50, 30, 70, 20, 40, 60, 80]

print("મૂલ્યો insert કરી રહ્યા છીએ:", values)
for value in values:
    bst.insert(value)

print("\nTree structure:")
bst.display_tree()

print("\nInorder traversal:", bst.inorder_traversal())

# Search ઉદાહરણો
print(f"\n40 શોધો: {'મળ્યું' if bst.search(40) else 'મળ્યું નહીં'}")
print(f"90 શોધો: {'મળ્યું' if bst.search(90) else 'મળ્યું નહીં'}")
```

ટેબલ: BST Operations

Operation	Time Complexity	વર્ણન
Insert	O(log n) average, O(n) worst	નવો node ઉમેરો
Search	O(log n) average, O(n) worst	ચોક્કસ node શોધો
Delete	O(log n) average, O(n) worst	Node દૂર કરો
Traversal	O(n)	બધા nodes ની મુલાકાત

મેમરી ટ્રીક: "BST નિયમ = ડાબે < રૂટ < જમણે"

પ્રશ્ન 5(a) OR [3 ગુણ]

Binary tree માં level, degree અને leaf node વ્યાખ્યાયિત કરો.

જવાબ:

ટેબલ: Binary Tree શબ્દો

શબ્દ	વ્યાખ્યા	ઉદાહરણ
Level	Root થી અંતર (root = level 0)	Root=0, Children=1, વગેરે
Degree	Node ના children ની સંખ્યા	0, 1, અથવા 2
Leaf Node	કોઈ children વગરનો node (degree = 0)	Terminal nodes

```
Levels સાથે Binary Tree:
Level 0:        A
               / \
Level 1:      B   C
             /   / \
Level 2:    D   E   F
```

ટેબલ: ઉદાહરણ Analysis

Node	Level	Degree	પ્રકાર
A	0	2	Root
B	1	1	Internal
C	1	2	Internal
D	2	0	Leaf
E	2	0	Leaf
F	2	0	Leaf

- **Height**: Tree માં મહત્તમ level
- **Depth**: એક node માટે level જેટલું જ

મેમરી ટ્રીક: "Level = માળનો નંબર, Degree = બાળકોની ગણતરી, Leaf = કોઈ બાળક નથી"

પ્રશ્ન 5(b) OR [4 ગુણ]

ઉદાહરણ સાથે complete binary tree સમજાવો.

જવાબ:

ટેબલ: Binary Tree પ્રકારો

પ્રકાર	વર્ણન	ગુણધર્મ
Complete	છેલ્લા સિવાય બધા levels ભરેલા, ડાબેથી ભરાય	કાર્યક્ષમ array representation
Full	દરેક node પાસે 0 અથવા 2 children	એક child વાળા nodes નથી
Perfect	બધા levels સંપૂર્ણ ભરેલા	2^h - 1 nodes

```
Complete Binary Tree:
        A
       / \
      B   C
     / \ /
    D  E F

Complete નથી:
        A
       / \
      B   C
       \ / \
        E F G
```

```python
class CompleteBinaryTree:
    def __init__(self):
        self.tree = []

    def insert(self, data):
        """Complete binary tree રીતે insert કરો"""
        self.tree.append(data)

    def get_parent_index(self, i):
        return (i - 1) // 2

    def get_left_child_index(self, i):
        return 2 * i + 1

    def get_right_child_index(self, i):
        return 2 * i + 2
```

```
    def display_level_order(self):
        """Tree ને level દ્વારા દર્શાવો"""
        if not self.tree:
            return

        level = 0
        while (2 ** level) <= len(self.tree):
            start = 2 ** level - 1
            end = min(2 ** (level + 1) - 1, len(self.tree))
            print(f"Level {level}: {self.tree[start:end]}")
            level += 1

# ઉદાહરણ
cbt = CompleteBinaryTree()
for i in [1, 2, 3, 4, 5, 6]:
    cbt.insert(i)

cbt.display_level_order()
```

ગુણધર્મો:

- **Array Representation**: Parent i પર, children 2i+1 અને 2i+2 પર
- **Heap Property**: Heap data structure નો આધાર બને

મેમરી ટ્રીક: "Complete = છેલ્લા સિવાય બધા માળ ભરેલા, ડાબેથી જમણે ભરાય"

પ્રશ્ન 5(c) OR [7 ગુણ]

નીચેના નંબરોના ક્રમ માટે binary search tree (BST) બનાવો: 50, 70, 60, 20, 90, 10, 40, 100

જવાબ:

Step-by-step BST Construction:

```
Insert 50: (Root)
    50

Insert 70: (70 > 50, જમણે જાઓ)
    50
     \
      70

Insert 60: (60 > 50, જમણે જાઓ; 60 < 70, ડાબે જાઓ)
    50
     \
      70
     /
    60

Insert 20: (20 < 50, ડાબે જાઓ)
    50
```

```
   /  \
  20   70
       /
      60

Insert 90: (90 > 50, જમણે; 90 > 70, જમણે)
    50
   /  \
  20   70
       / \
      60  90

Insert 10: (10 < 50, ડાબે; 10 < 20, ડાબે)
    50
   /  \
  20   70
 /    / \
10   60  90

Insert 40: (40 < 50, ડાબે; 40 > 20, જમણે)
    50
   /  \
  20   70
 / \   / \
10  40 60 90

Insert 100: (100 > 50, જમણે; 100 > 70, જમણે; 100 > 90, જમણે)
    50
   /  \
  20   70
 / \   / \
10  40 60 90
             \
              100
```

અંતિમ BST માળખું:

```
        50
       /  \
      20   70
     / \   / \
    10  40 60 90
              \
               100
```

```python
# Construction માટે implementation code
class BST:
    def __init__(self):
        self.root = None

    def insert(self, data):
```

```
        if self.root is None:
            self.root = TreeNode(data)
            print(f"{data} ને root તરીકે insert કર્યું")
        else:
            self._insert(self.root, data)

    def _insert(self, node, data):
        if data < node.data:
            if node.left is None:
                node.left = TreeNode(data)
                print(f"{data} ને {node.data} ના ડાબે insert કર્યું")
            else:
                self._insert(node.left, data)
        else:
            if node.right is None:
                node.right = TreeNode(data)
                print(f"{data} ને {node.data} ના જમણે insert કર્યું")
            else:
                self._insert(node.right, data)

# BST બનાવો
bst = BST()
sequence = [50, 70, 60, 20, 90, 10, 40, 100]

for num in sequence:
    bst.insert(num)
```

ટેબલ: Traversal પરિણામો

Traversal	પરિણામ
Inorder	10, 20, 40, 50, 60, 70, 90, 100
Preorder	50, 20, 10, 40, 70, 60, 90, 100
Postorder	10, 40, 20, 60, 100, 90, 70, 50

ચકાસાયેલા ગુણધર્મો:

- **BST Property**: ડાબું subtree < Root < જમણું subtree
- **Inorder**: Sorted sequence આપે છે

મેમરી ટ્રીક: "BST Construction = તુલના કરો, દિશા પસંદ કરો, Insert કરો"