પ્રશ્ન 1(a) [3 ગુણ]

પાયથોનમાં સેટ ડેટા સ્ટ્રક્ચર સમજાવો?

ઉત્તર:

સેટ એ પાયથોનમાં અનન્ય એલિમેન્ટ્સનો અક્રમ સંગ્રહ છે. સેટ્સ mutable છે પરંતુ તેમાં ફક્ત immutable એલિમેન્ટ્સ જ હોય છે.

મુખ્ય લક્ષણો:

લક્ષણ	વર્ણન
અનન્ય એલિમેન્ટ્સ	કોઈ ડુપ્લિકેટ વેલ્યુઝ મંજૂર નથી
અક્રમ	કોઈ ઇન્ડેક્સિંગ અથવા સ્લાઇસિંગ નથી
Mutable	એલિમેન્ટ્સ ઉમેરી/દૂર કરી શકાય છે
Iterable	એલિમેન્ટ્સમાં લૂપ ચલાવી શકાય છે

મૂળભૂત ઓપરેશન્સ:

```
# સેટ બનાવો
my_set = {1, 2, 3, 4}
# એલિમેન્ટ ઉમેરો
my_set.add(5)
# એલિમેન્ટ દૂર કરો
my_set.remove(2)
```

મેમરી ટ્રીક: "સેટ્સ એ અનન્ય અક્રમ સંગ્રહો છે"

પ્રશ્ન 1(b) [4 ગુણ]

પાયથોનમાં Tuple ની વ્યાખ્યા આપો? પાયથોનમાં Tuple data structure ના operations સમજાવો.

ઉત્તર:

Tuple એ આઇટમ્સનો ક્રમિત સંગ્રહ છે જે immutable છે (બનાવ્યા પછી બદલી શકાતું નથી).

Tuple વ્યાખ્યા:

- **ક્રમિત**: એલિમેન્ટ્સનો નિશ્ચિત ક્રમ
- **Immutable**: બનાવ્યા પછી બદલી શકાતું નથી
- **ડુપ્લિકેટ્સ મંજૂર**: સમાન વેલ્યુઝ આવી શકે છે
- **ઇન્ડેક્સ્ડ**: ઇન્ડેક્સ વાપરીને એક્સેસ કરી શકાય છે

Tuple ઓપરેશન્સ:

ઓપરેશન	ઉદાહરણ	વર્ણન
બનાવવું	`t = (1, 2, 3)`	Tuple બનાવો
ઇન્ડેક્સિંગ	`t[0]`	પ્રથમ એલિમેન્ટ એક્સેસ કરો
સ્લાઇસિંગ	`t[1:3]`	સબસેટ મેળવો
લેન્થ	`len(t)`	એલિમેન્ટ્સ ગણો
જોડાણ	`t1 + t2`	Tuples જોડો

```
# ઉદાહરણ ઓપરેશન્સ
tup = (10, 20, 30, 40)
print(tup[1])      # આઉટપુટ: 20
print(tup[1:3])    # આઉટપુટ: (20, 30)
```

મેમરી ટ્રીક: "Tuples એ Immutable ક્રમિત સંગ્રહો છે"

પ્રશ્ન 1(c) [7 ગુણ]

પાયથોનમાં કન્સ્ટ્રક્ટરના પ્રકારો સમજાવો? Static methodનો ઉપયોગ કરીને બે સંખ્યાઓના ગુણાકાર માટે પાયથોન પ્રોગ્રામ લખો.

ઉત્તર:

કન્સ્ટ્રક્ટરના પ્રકારો:

કન્સ્ટ્રક્ટર પ્રકાર	વર્ણન	ઉપયોગ
Default Constructor	કોઈ પેરામીટર નથી	`__init__(self)`
Parameterized Constructor	પેરામીટર લે છે	`__init__(self, params)`
Non-parameterized Constructor	ફક્ત self પેરામીટર	મૂળભૂત પ્રારંભિકરણ

Static Method પ્રોગ્રામ:

```
class Calculator:
    def __init__(self):
        pass

    @staticmethod
    def multiply(num1, num2):
        return num1 * num2

# ઉપયોગ
result = Calculator.multiply(5, 3)
print(f"ગુણાકાર: {result}")  # આઉટપુટ: 15
```

મુખ્ય મુદ્દાઓ:

- **Static methods**: ઑબ્જેક્ટ ઇન્સ્ટન્સની જરૂર નથી
- **@staticmethod decorator**: Static method દર્શાવે છે
- **કોઈ self પેરામીટર નથી**: ક્લાસ ઇન્સ્ટન્સથી સ્વતંત્ર

મેમરી ટ્રીક: "Static methods સેલ્ફથી અલગ રહે છે"

પ્રશ્ન 1(c) અથવા [7 ગુણ]

Data Encapsulation ની વ્યાખ્યા આપો? પાયથોનમાં વિવિધ પ્રકારની methods ની યાદી આપો. Multilevel inheritance માટે પાયથોન પ્રોગ્રામ લખો.

ઉત્તર:

Data Encapsulation:
ડેટા એન્કેપ્સુલેશન એ ડેટા અને methods ને ક્લાસની અંદર બાંધવાની અને કેટલાક ઘટકોની સીધી પહોંચ મર્યાદિત કરવાની વિભાવના છે.

Methods ના પ્રકારો:

Method પ્રકાર	પહોંચ સ્તર	ઉદાહરણ
Public	બધે પહોંચી શકાય છે	`method()`
Protected	ક્લાસ અને સબક્લાસ	`_method()`
Private	ફક્ત ક્લાસની અંદર	`__method()`
Static	ક્લાસ લેવલ	`@staticmethod`
Class	ક્લાસ અને સબક્લાસીઝ	`@classmethod`

Multilevel Inheritance પ્રોગ્રામ:

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print(f"{self.name} અવાજ કરે છે")

class Mammal(Animal):
    def __init__(self, name, warm_blooded):
        super().__init__(name)
        self.warm_blooded = warm_blooded

class Dog(Mammal):
    def __init__(self, name, breed):
        super().__init__(name, True)
        self.breed = breed

    def bark(self):
```

```
        print(f"{self.name} ભસે છે")

# ઉપયોગ
dog = Dog("બડી", "ગોલ્ડન રિટ્રીવર")
dog.speak()  # Animal ક્લાસથી
dog.bark()   # Dog ક્લાસથી
```

મેમરી ટ્રીક: "એન્કેપ્સુલેશન આંતરિક વિગતો છુપાવે છે"

પ્રશ્ન 2(a) [3 ગુણ]

Simple Queue અને Circular Queue વચ્ચેનો તફાવત આપો.

ઉત્તર:

Queue સરખામણી:

લક્ષણ	Simple Queue	Circular Queue
સ્ટ્રક્ચર	રેખીય ગોઠવણી	વર્તુળાકાર ગોઠવણી
મેમરી ઉપયોગ	બગાડજનક (ખાલી જગ્યાઓ)	કાર્યક્ષમ (જગ્યા પુનઃઉપયોગ)
Rear Pointer	રેખીય રીતે આગળ વધે છે	ફરીથી આગળ વળે છે
Front Pointer	રેખીય રીતે આગળ વધે છે	ફરીથી આગળ વળે છે
જગ્યા ઉપયોગ	નબળો	ઉત્તમ

મુખ્ય તફાવતો:

- **Simple Queue**: Front અને rear ફક્ત એક દિશામાં જાય છે
- **Circular Queue**: Rear ફરીથી front સ્થાને જોડાય છે
- **કાર્યક્ષમતા**: Circular queue મેમરી વેસ્ટેજ ટાળે છે

મેમરી ટ્રીક: "Circular Queues વર્તુળ પૂર્ણ કરે છે"

પ્રશ્ન 2(b) [4 ગુણ]

પાયથોનમાં પોલીમોર્ફિઝમ ઉદાહરણ સાથે સમજાવો.

ઉત્તર:

પોલીમોર્ફિઝમ એટલે "અનેક સ્વરૂપો" - સમાન method નામ અલગ અલગ ક્લાસમાં અલગ રીતે વર્તે છે.

પોલીમોર્ફિઝમના પ્રકારો:

પ્રકાર	વર્ણન	અમલીકરણ
Method Overriding	ચાઇલ્ડ ક્લાસ પેરેન્ટ method ફરીથી વ્યાખ્યાયિત કરે છે	Inheritance
Duck Typing	અલગ ક્લાસમાં સમાન method	Interface સમાનતા
Operator Overloading	સમાન ઓપરેટર અલગ વર્તન	Magic methods

ઉદાહરણ:

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

class Dog(Animal):
    def make_sound(self):
        return "ભૌં ભૌં!"

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

# પોલીમોર્ફિક વર્તન
animals = [Dog(), Cat()]
for animal in animals:
    print(animal.make_sound())
```

મેમરી ટ્રીક: "પોલીમોર્ફિઝમ અનેક વ્યક્તિત્વ પ્રદાન કરે છે"

પ્રશ્ન 2(c) [7 ગુણ]

વ્યાખ્યા આપો. a). Infix b).Postfix સ્ટેકનો ઉપયોગ કરીને આપેલ Infix expression ને Postfix expression માં ફેરવો. A+(B*C/D)

ઉત્તર:

વ્યાખ્યાઓ:

Expression પ્રકાર	વર્ણન	ઉદાહરણ
Infix	ઓપરેટર ઓપરેન્ડ્સ વચ્ચે	`A + B`
Postfix	ઓપરેટર ઓપરેન્ડ્સ પછી	`A B +`

રૂપાંતરણ એલ્ગોરિધમ:

1. Infix expression ને ડાબેથી જમણે સ્કેન કરો
2. જો operand છે, આઉટપુટમાં ઉમેરો
3. જો operator છે, સ્ટેક ટોપ સાથે precedence સરખાવો

4. વધુ precedence → સ્ટેકમાં push કરો
5. ઓછી/સમાન precedence → pop કરીને આઉટપુટમાં ઉમેરો

પગલાં પ્રમાણે રૂપાંતરણ: A+(B*C/D)

```
ઇનપુટ: A+(B*C/D)

પગલું | સિમ્બલ | સ્ટેક | આઉટપુટ
------|--------|-------|--------
1     | A      | []    | A
2     | +      | [+]   | A
3     | (      | [+,(] | A
4     | B      | [+,(] | AB
5     | *      | [+,(,*] | AB
6     | C      | [+,(,*] | ABC
7     | /      | [+,(,/] | ABC*
8     | D      | [+,(,/] | ABC*D
9     | )      | [+]   | ABC*D/
10    | અંત    | []    | ABC*D/+
```

અંતિમ જવાબ: ABC*D/+

મેમરી ટ્રીક: "સ્ટેક ઓપરેટર્સને વ્યૂહાત્મક રીતે સંગ્રહિત કરે છે"

પ્રશ્ન 2(a) અથવા [3 ગુણ]

Queue ના ગેરફાયદા સમજાવો.

ઉત્તર:

Queue ગેરફાયદાઓ:

ગેરફાયદો	વર્ણન	અસર
મેમરી વેસ્ટેજ	ખાલી જગ્યાઓ પુનઃઉપયોગ નથી	નબળો જગ્યા ઉપયોગ
નિયત કદ	મર્યાદિત ક્ષમતા	Overflow સમસ્યાઓ
રેન્ડમ એક્સેસ નથી	ફક્ત front/rear એક્સેસ	મર્યાદિત લવચીકતા

મુખ્ય સમસ્યાઓ:

- **રેખીય Queue**: આગળની જગ્યાઓ અનુપયોગી બને છે
- **Insertion/Deletion**: ફક્ત ચોક્કસ છેડાઓથી
- **શોધ ઓપરેશન્સ**: શોધવા માટે કાર્યક્ષમ નથી

મેમરી ટ્રીક: "Queues શાંતિથી ક્વિર્ક્સ સાથે કતાર લગાવે છે"

પ્રશ્ન 2(b) અથવા [4 ગુણ]

પાયથોનમાં Abstract class ની વ્યાખ્યા આપો? પાયથોનમાં abstract method નું declaration સમજાવો?

ઉત્તર:

Abstract Class:
એક ક્લાસ જેનું instantiation કરી શકાતું નથી અને જેમાં એક અથવા વધુ abstract methods હોય છે જે સબક્લાસીઝ દ્વારા અમલ કરવા જોઈએ.

Abstract Method Declaration:

ઘટક	હેતુ	સિન્ટેક્સ
ABC Module	Abstract base class પ્રદાન કરે છે	`from abc import ABC`
@abstractmethod	Abstract methods માટે decorator	`@abstractmethod`
અમલીકરણ	સબક્લાસમાં ફરજિયાત override	આવશ્યક

ઉદાહરણ:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)
```

મેમરી ટ્રીક: "Abstract classes ફક્ત બ્લૂપ્રિન્ટ છે"

પ્રશ્ન 2(c) અથવા [7 ગુણ]

Infix to postfix માટેનો અલ્ગોરિધમ લખો. નીચેની પોસ્ટફિક્સ એક્સપ્રેશન મૂલ્યાંકન કરો. 5 6 2 + * 12 4 / -

ઉત્તર:

Infix to Postfix અલ્ગોરિધમ:

1. ખાલી સ્ટેક અને આઉટપુટ સ્ટ્રિંગ પ્રારંભ કરો
2. ડાબેથી જમણે infix expression સ્કેન કરો
3. જો operand છે → આઉટપુટમાં ઉમેરો
4. જો '(' છે → સ્ટેકમાં push કરો
5. જો ')' છે → '(' સુધી pop કરો
6. જો operator છે → વધુ/સમાન precedence operators pop કરો
7. વર્તમાન operator સ્ટેકમાં push કરો
8. બાકીના operators pop કરો

Postfix મૂલ્યાંકન: 5 6 2 + * 12 4 / -

```
એક્સપ્રેશન: 5 6 2 + * 12 4 / -

પગલું | ટોકન | સ્ટેક | ઓપરેશન
------|-------|-------|----------
1     | 5     | [5]   | Operand push કરો
2     | 6     | [5,6] | Operand push કરો
3     | 2     | [5,6,2] | Operand push કરો
4     | +     | [5,8] | Pop 2,6 → 6+2=8
5     | *     | [40]  | Pop 8,5 → 5*8=40
6     | 12    | [40,12] | Operand push કરો
7     | 4     | [40,12,4] | Operand push કરો
8     | /     | [40,3] | Pop 4,12 → 12/4=3
9     | -     | [37]  | Pop 3,40 → 40-3=37
```

અંતિમ પરિણામ: 37

મેમરી ટ્રીક: "Postfix પ્રોસેસિંગ યુગ્મો થોક્કસ રીતે Pop કરે છે"

પ્રશ્ન 3(a) [3 ગુણ]

સિંગલ લિંક લિસ્ટમાં નોડને traverse કરવા માટે અલ્ગોરિધમ લખો.

ઉત્તર:

Traversal અલ્ગોરિધમ:

```python
def traverse_linked_list(head):
    current = head
    while current is not None:
        print(current.data)
        current = current.next
```

અલ્ગોરિધમ પગલાં:

પગલું	ક્રિયા	હેતુ
1	Head નોડથી શરૂ કરો	Traversal પ્રારંભ કરો
2	ચકાસો કે current ≠ NULL	ચાલુ રાખવાની શરત
3	વર્તમાન નોડ પ્રોસેસ કરો	ઓપરેશન કરો
4	આગલા નોડ પર જાઓ	Pointer આગળ વધારો
5	અંત સુધી પુનરાવર્તન કરો	સંપૂર્ણ traversal

મેમરી ટ્રીક: "Traverse ટેઇલ સુધી પહોંચે છે"

પ્રશ્ન 3(b) [4 ગુણ]

લિસ્ટનો ઉપયોગ કરીને Queueના Dequeue ઓપરેશન માટેનો અલ્ગોરિધમ લખો.

ઉત્તર:

Dequeue અલ્ગોરિધમ:

```
def dequeue(queue):
    if len(queue) == 0:
        print("Queue ખાલી છે")
        return None
    else:
        element = queue.pop(0)
        return element
```

અલ્ગોરિધમ પગલાં:

પગલું	શરત	ક્રિયા
1	ખાલી ચકાસો	જો queue ખાલી છે
2	Underflow હેન્ડલ કરો	એરર મેસેજ દર્શાવો
3	એલિમેન્ટ દૂર કરો	આગળનું એલિમેન્ટ ડિલીટ કરો
4	એલિમેન્ટ પરત કરો	દૂર કરેલી વેલ્યુ પરત કરો
5	સ્ટ્રક્ચર અપડેટ કરો	Queue pointers એડજસ્ટ કરો

Time Complexity: O(n) લિસ્ટ shifting ને કારણે

મેમરી ટ્રીક: "Dequeue આગળના દરવાજેથી ડિલીટ કરે છે"

પ્રશ્ન 3(c) [7 ગુણ]

Double linked list ની વ્યાખ્યા આપો. લિંક લિસ્ટના મુખ્ય ઓપરેશનની નોંધણી કરો. Single Linked list મા શરૂઆતમાં નોડ દાખલ કરવા માટેનો અલ્ગોરિધમ લખો.

ઉત્તર:

Double Linked List:
એક રેખીય ડેટા સ્ટ્રક્ચર જ્યાં દરેક નોડમાં ડેટા અને બે pointers હોય છે - એક આગલા નોડ તરફ અને બીજો પાછલા નોડ તરફ.

લિંક લિસ્ટના મુખ્ય ઓપરેશન્સ:

ઓપરેશન	વર્ણન	Time Complexity
Insertion	નવો નોડ ઉમેરો	O(1) શરૂઆતમાં
Deletion	નોડ દૂર કરો	O(1) જો નોડ ખબર છે
Traversal	બધા નોડ્સની મુલાકાત લો	O(n)
Search	ચોક્કસ નોડ શોધો	O(n)
Update	નોડ ડેટા બદલો	O(1) જો નોડ ખબર છે

શરૂઆતમાં Insert અલ્ગોરિધમ:

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

def insert_at_beginning(head, data):
    new_node = Node(data)
    new_node.next = head
    head = new_node
    return head
```

અલ્ગોરિધમ પગલાં:

1. આપેલ ડેટા સાથે નવો નોડ બનાવો
2. નવા નોડનો next વર્તમાન head પર સેટ કરો
3. Head ને નવા નોડ તરફ અપડેટ કરો
4. નવો head પરત કરો

મેમરી ટ્રીક: "શરૂઆતમાં Insert બેહતર લિસ્ટ બનાવે છે"

પ્રશ્ન 3(a) અથવા [3 ગુણ]

Single Linked List ની એપ્લિકેશન સમજાવો.

ઉત્તર:

Single Linked List એપ્લિકેશન્સ:

એપ્લિકેશન	ઉપયોગ કેસ	ફાયદો
Dynamic Memory	વેરિયેબલ સાઇઝ ડેટા	મેમરી કાર્યક્ષમ
Stack Implementation	LIFO ઓપરેશન્સ	સરળ push/pop
Queue Implementation	FIFO ઓપરેશન્સ	Dynamic sizing
Music Playlist	સિક્વેન્શિયલ પ્લેબેક	સરળ નેવિગેશન
Browser History	પેજ નેવિગેશન	ફોરવર્ડ traversal
Polynomial Representation	ગાણિતિક ઓપરેશન્સ	Coefficient સ્ટોરેજ

મુખ્ય ફાયદાઓ:

- **Dynamic Size**: રનટાઇમ દરમિયાન વધે/ઘટે છે
- **મેમરી કાર્યક્ષમતા**: જરૂર પ્રમાણે allocate કરે છે
- **Insertion/Deletion**: કોઈપણ સ્થાને કાર્યક્ષમ

મેમરી ટ્રીક: "Linked Lists અનેક એપ્લિકેશન્સને લિંક કરે છે"

પ્રશ્ન 3(b) અથવા [4 ગુણ]

લિસ્ટનો ઉપયોગ કરીને સ્ટેકના PUSH ઓપરેશન માટે અલ્ગોરિધમ લખો.

ઉત્તર:

PUSH અલ્ગોરિધમ:

```
def push(stack, element):
    stack.append(element)
    print(f"સ્ટેકમાં {element} push કર્યું")
```

અલ્ગોરિધમ પગલાં:

પગલું	ક્રિયા	વર્ણન
1	ક્ષમતા ચકાસો	સ્ટેક ભરાયો નથી તે ચકાસો
2	એલિમેન્ટ ઉમેરો	લિસ્ટના અંતે append કરો
3	ટોપ અપડેટ કરો	ટોપ છેલ્લા એલિમેન્ટ તરફ પોઇન્ટ કરે છે
4	ઓપરેશન કન્ફર્મ કરો	સફળતાનો મેસેજ દર્શાવો

વિગતવાર અલ્ગોરિધમ:

1. સ્ટેક અને push કરવાનું એલિમેન્ટ સ્વીકારો
2. સ્ટેકની ક્ષમતા ચકાસો (fixed size માટે)
3. append() વાપરીને લિસ્ટના અંતે એલિમેન્ટ ઉમેરો
4. લિસ્ટ આપોઆપ મેમરી allocation હેન્ડલ કરે છે
5. સફળતાની સ્થિતિ પરત કરો

Time Complexity: O(1) - Constant time ઓપરેશન

મેમરી ટ્રીક: "PUSH સ્ટેક શિખર પર મૂકે છે"

પ્રશ્ન 3(c) અથવા [7 ગુણ]

Linked list ના ફાયદા સમજાવો. Single linked list માંથી last નોડ કાઢી નાખવા માટે અલ્ગોરિધમ લખો.

ઉત્તર:

Linked List ફાયદાઓ:

ફાયદો	વર્ણન	લાભ
Dynamic Size	રનટાઇમે સાઇઝ બદલાય છે	મેમરી લવચીક
મેમરી કાર્યક્ષમ	જરૂર પ્રમાણે allocate કરે છે	કોઈ વેસ્ટેજ નથી
સરળ Insertion	ગમે ત્યાં કાર્યક્ષમ રીતે ઉમેરો	O(1) ઓપરેશન
સરળ Deletion	કાર્યક્ષમ રીતે દૂર કરો	O(1) ઓપરેશન
મેમરી Shift નથી	એલિમેન્ટ્સ ખસતા નથી	ઝડપી ઓપરેશન્સ

છેલ્લો નોડ ડિલીટ કરવાનો અલ્ગોરિધમ:

```
def delete_last_node(head):
    # ખાલી લિસ્ટ
    if head is None:
        return None

    # એક જ નોડ
    if head.next is None:
        return None

    # અનેક નોડ્સ
    current = head
    while current.next.next is not None:
        current = current.next

    current.next = None
    return head
```

અલ્ગોરિધમ પગલાં:

1. ખાલી લિસ્ટ કેસ હેન્ડલ કરો
2. એક નોડ કેસ હેન્ડલ કરો
3. છેલ્લાથી બીજા નોડ સુધી traverse કરો
4. છેલ્લાથી બીજા નોડનો next NULL પર સેટ કરો
5. અપડેટેડ head પરત કરો

મેમરી ટ્રીક: "Linked Lists તાર્કિક ફાયદાઓ તરફ દોરી જાય છે"

પ્રશ્ન 4(a) [3 ગુણ]

બબલ સૉર્ટમાટેના અલ્ગોરિધમ લખો.

ઉત્તર:

Bubble Sort અલ્ગોરિધમ:

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

અલ્ગોરિધમ પગલાં:

પગલું	ક્રિયા	હેતુ
1	બાહ્ય લૂપ i=0 થી n-1	પાસની સંખ્યા
2	આંતરિક લૂપ j=0 થી n-i-2	નજીકના એલિમેન્ટ્સ સરખાવો
3	arr[j] અને arr[j+1] સરખાવો	ક્રમ ચકાસો
4	ક્રમ ખોટો હોય તો અદલાબદલી કરો	યોગ્ય સ્થિતિ
5	સૉર્ટ થાય સુધી પુનરાવર્તન કરો	સંપૂર્ણ સૉર્ટિંગ

Time Complexity: O(n²)

મેમરી ટ્રીક: "બબલ્સ ધીમે ધીમે સપાટી પર આવે છે"

પ્રશ્ન 4(b) [4 ગુણ]

Circular linked list ને તેના ફાયદાઓ સાથે સમજાવો.

ઉત્તર:

Circular Linked List:
એક લિંક લિસ્ટ જ્યાં છેલ્લો નોડ પ્રથમ નોડ તરફ પોઇન્ટ કરે છે, વર્તુળાકાર સ્ટ્રક્ચર બનાવે છે.

લક્ષણો:

લક્ષણ	વર્ણન	ફાયદો
વર્તુળાકાર સ્ટ્રક્ચર	છેલ્લો નોડ → પ્રથમ નોડ	સતત traversal
કોઈ NULL Pointers નથી	કોઈ અંત માર્કર નથી	હંમેશા જોડાયેલ
કાર્યક્ષમ Traversal	કોઈપણ નોડથી શરૂ કરી શકાય છે	લવચીક એક્સેસ

ફાયદાઓ:

- **મેમરી કાર્યક્ષમ**: કોઈ NULL pointers નથી
- **વર્તુળાકાર Traversal**: સતત લૂપ કરી શકાય છે
- **Queue Implementation**: કાર્યક્ષમ enqueue/dequeue
- **Round Robin Scheduling**: CPU time sharing
- **Music Player**: સતત playlist looping

```
Circular Linked List સ્ટ્રક્ચર:

[A] → [B] → [C] → [D]
 ↑                 ↓
 ←←←←←←←←←←←←←←←←←
```

મેમરી ટ્રીક: "Circular Lists સતત કનેક્શન્સ બનાવે છે"

પ્રશ્ન 4(c) [7 ગુણ]

યોગ્ય ઉદાહરણ સાથે મર્જ સૉર્ટ સમજાવો.

ઉત્તર:

Merge Sort:
એક divide-and-conquer અલ્ગોરિધમ જે array ને ભાગોમાં વહેંચે છે, તેમને અલગ અલગ સૉર્ટ કરે છે, અને પછી ભેગા કરે છે.

અલ્ગોરિધમ તબક્કાઓ:

તબક્કો	ક્રિયા	વર્ણન
Divide	Array વહેંચો	બે ભાગોમાં વહેંચો
Conquer	Subarrays સૉર્ટ કરો	ભાગોને recursively સૉર્ટ કરો
Combine	પરિણામો મર્જ કરો	સૉર્ટેડ ભાગો મર્જ કરો

ઉદાહરણ: [38, 27, 43, 3, 9, 82, 10]

```
Merge Sort પ્રક્રિયા:

લેવલ 0:  [38, 27, 43, 3, 9, 82, 10]
             ↓
લેવલ 1:  [38, 27, 43, 3] | [9, 82, 10]
             ↓                  ↓
લેવલ 2:  [38, 27] [43, 3] | [9, 82] [10]
             ↓       ↓         ↓      ↓
લેવલ 3:  [38][27] [43][3] | [9][82] [10]
             ↓       ↓         ↓      ↓
મર્જ:     [27, 38] [3, 43] | [9, 82] [10]
             ↓                  ↓
          [3, 27, 38, 43] | [9, 10, 82]
             ↓
          [3, 9, 10, 27, 38, 43, 82]
```

Time Complexity: O(n log n)
Space Complexity: O(n)

મેમરી ટ્રીક: "Merge Sort વ્યવસ્થિત રીતે સેગમેન્ટ્સને મર્જ કરે છે"

પ્રશ્ન 4(a) અથવા [3 ગુણ]

Selection sort માટેના અલ્ગોરિધમ લખો.

ઉત્તર:

Selection Sort અલ્ગોરિધમ:

```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
```

અલ્ગોરિધમ પગલાં:

પગલું	ક્રિયા	હેતુ
1	મિનિમમ એલિમેન્ટ શોધો	સૌથી નાનું લોકેટ કરો
2	પ્રથમ સ્થાને અદલાબદલી કરો	સૉર્ટેડ સ્થાને મૂકો
3	આગલી પોઝિશન પર જાઓ	બાઉન્ડ્રી આગળ વધારો
4	બાકીના માટે પુનરાવર્તન કરો	સૉર્ટિંગ ચાલુ રાખો
5	પૂર્ણ થયા પર સમાપ્ત કરો	અલ્ગોરિધમ ફિનિશ કરો

Time Complexity: O(n²)

મેમરી ટ્રીક: "Selection Sort સફળતાપૂર્વક સૌથી નાનું સિલેક્ટ કરે છે"

પ્રશ્ન 4(b) અથવા [4 ગુણ]

Double linked list ને તેના ફાયદાઓ સાથે સમજાવો.

ઉત્તર:

Double Linked List:
એક લિંક લિસ્ટ જ્યાં દરેક નોડમાં ડેટા અને બે pointers હોય છે - next અને previous.

સ્ટ્રક્ચર:

ઘટક	હેતુ	દિશા
ડેટા	માહિતી સ્ટોર કરો	-
Next Pointer	આગલા નોડ તરફ પોઇન્ટ કરે છે	આગળ
Previous Pointer	પાછલા નોડ તરફ પોઇન્ટ કરે છે	પાછળ

ફાયદાઓ:

- **બાયડાયરેક્શનલ Traversal**: આગળ અને પાછળ બંને દિશામાં ચાલી શકાય છે
- **સરળ Deletion**: પાછલો નોડ જાણ્યા વગર ડિલીટ કરી શકાય છે
- **કાર્યક્ષમ Insertion**: કોઈપણ સ્થાને સરળતાથી insert કરી શકાય છે
- **બહેતર Navigation**: બંને દિશામાં ચાલી શકાય છે

```
Double Linked List સ્ટ્રક્ચર:

NULL ← [prev|data|next] ⇄ [prev|data|next] ⇄ [prev|data|next] → NULL
```

એપ્લિકેશન્સ:

- **Browser navigation** (back/forward બટન્સ)

- **Music player** (previous/next ગીત)
- **Undo/Redo operations**

મેમરી ટ્રીક: "Double Links બેવડી દિશા પ્રદાન કરે છે"

પ્રશ્ન 4(c) અથવા [7 ગુણ]

Insertion સૉર્ટ સમજાવો. Insertion સૉર્ટનો ઉપયોગ કરીને નીચેના નંબરોનો ટ્રેસ આપો: 25, 15,30,9,99,20,26

ઉત્તર:

Insertion Sort:
એક સમયે એક એલિમેન્ટ દ્વારા સૉર્ટેડ array બનાવે છે દરેક એલિમેન્ટને તેની યોગ્ય પોઝિશનમાં insert કરીને.

અલ્ગોરિધમ કન્સેપ્ટ:

- **સૉર્ટેડ ભાગ**: વર્તમાન એલિમેન્ટની ડાબી બાજુ
- **અનસૉર્ટેડ ભાગ**: વર્તમાન એલિમેન્ટની જમણી બાજુ
- **Insert સ્ટ્રેટેજી**: વર્તમાન એલિમેન્ટને સૉર્ટેડ ભાગમાં યોગ્ય સ્થાને મૂકો

[25, 15, 30, 9, 99, 20, 26] નો ટ્રેસ:

પાસ	વર્તમાન	Array સ્થિતિ	સરખામણીઓ	ક્રિયા
પ્રારંભિક	-	[25, 15, 30, 9, 99, 20, 26]	-	શરૂ
1	15	[15, 25, 30, 9, 99, 20, 26]	15 < 25	15 ને 25 પહેલાં insert કરો
2	30	[15, 25, 30, 9, 99, 20, 26]	30 > 25	30 ને જગ્યાએ રાખો
3	9	[9, 15, 25, 30, 99, 20, 26]	9 < બધા	શરૂઆતમાં insert કરો
4	99	[9, 15, 25, 30, 99, 20, 26]	99 > 30	99 ને જગ્યાએ રાખો
5	20	[9, 15, 20, 25, 30, 99, 26]	15 અને 25 વચ્ચે insert	શિફ્ટ કરીને insert કરો
6	26	[9, 15, 20, 25, 26, 30, 99]	25 અને 30 વચ્ચે insert	અંતિમ સ્થિતિ

અંતિમ સૉર્ટેડ Array: [9, 15, 20, 25, 26, 30, 99]

Time Complexity: O(n²) worst case, **O(n)** best case

મેમરી ટ્રીક: "Insertion વધતા ક્રમમાં Insert કરે છે"

પ્રશ્ન 5(a) [3 ગુણ]

બાઈનરી ટ્રીની એપ્લિકેશન સમજાવો.

ઉત્તર:

Binary Tree એપ્લિકેશન્સ:

એપ્લિકેશન	ઉપયોગ કેસ	ફાયદો
Expression Trees	ગાણિતિક expressions	સરળ evaluation
Binary Search Trees	Searching/Sorting	O(log n) operations
Heap Trees	Priority queues	કાર્યક્ષમ min/max
File Systems	Directory structure	હાયરાર્કિકલ ઓર્ગેનાઇઝેશન
Decision Trees	AI/ML algorithms	Classification
Huffman Coding	Data compression	Optimal encoding

મુખ્ય ફાયદાઓ:

- **હાયરાર્કિકલ સ્ટ્રક્ચર**: કુદરતી tree ઓર્ગેનાઇઝેશન
- **કાર્યક્ષમ ઓપરેશન્સ**: Search, insert, delete
- **Recursive Processing**: અમલ કરવામાં સરળ

મેમરી ટ્રીક: "Binary Trees અનેક એપ્લિકેશન્સમાં શાખા કરે છે"

પ્રશ્ન 5(b) [4 ગુણ]

લિસ્ટનો ઉપયોગ કરીને Binary search માટેનો અલ્ગોરિધમ લખો.

ઉત્તર:

Binary Search અલ્ગોરિધમ:

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1  # એલિમેન્ટ મળ્યું નથી
```

અલ્ગોરિધમ પગલાં:

પગલું	ક્રિયા	હેતુ
1	left=0, right=n-1 સેટ કરો	બાઉન્ડ્રીઝ પ્રારંભ કરો
2	Mid કેલ્ક્યુલેટ કરો	મધ્ય એલિમેન્ટ શોધો
3	Target ને mid સાથે સરખાવો	દિશા નક્કી કરો
4	બાઉન્ડ્રીઝ અપડેટ કરો	શોધ જગ્યા સાંકડી કરો
5	મળે સુધી પુનરાવર્તન કરો	શોધ ચાલુ રાખો

પૂર્વશરત: Array સૉર્ટેડ હોવું જોઈએ
Time Complexity: O(log n)

મેમરી ટ્રીક: "Binary Search ઝડપથી શોધવા માટે બાઇસેક્ટ કરે છે"

પ્રશ્ન 5(c) [7 ગુણ]

Tree ની વ્યાખ્યા આપો. Tree ની યાદી બનાવો. પાયથોનનો ઉપયોગ કરીને બાઈનરી સર્ચ ટ્રીમાં નોડ દાખલ કરવા માટે અલ્ગોરિધમ લખો.

ઉત્તર:

Tree વ્યાખ્યા:
એક હાયરાર્કિકલ ડેટા સ્ટ્રક્ચર જેમાં edges દ્વારા જોડાયેલા nodes હોય છે, એક root node સાથે અને કોઈ cycles ન હોય.

Tree ના પ્રકારો:

Tree પ્રકાર	વર્ણન	વિશેષ ગુણધર્મ
Binary Tree	નોડ દીઠ વધુમાં વધુ 2 બાળકો	ડાબું અને જમણું બાળક
Binary Search Tree	ક્રમિત binary tree	ડાબું < Root < જમણું
Complete Binary Tree	છેલ્લા સિવાય બધા લેવલ ભરેલા	કાર્યક્ષમ heap
Full Binary Tree	બધા nodes ને 0 અથવા 2 બાળકો	કોઈ એક બાળક નથી
AVL Tree	સ્વ-સંતુલિત BST	Height balanced
Red-Black Tree	સ્વ-સંતુલિત BST	રંગ ગુણધર્મો

BST Insertion અલ્ગોરિધમ:

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

```
def insert_bst(root, data):
    if root is None:
        return TreeNode(data)

    if data < root.data:
        root.left = insert_bst(root.left, data)
    elif data > root.data:
        root.right = insert_bst(root.right, data)

    return root
```

અલ્ગોરિધમ પગલાં:

1. જો tree ખાલી છે, root node બનાવો
2. જો data < root.data, ડાબા subtree માં insert કરો
3. જો data > root.data, જમણા subtree માં insert કરો
4. જો data = root.data, ignore કરો (duplicates નથી)
5. અપડેટેડ root પરત કરો

મેમરી ટ્રીક: "Trees સંરચિત ઓર્ગેનાઇઝેશન સાથે વધે છે"

પ્રશ્ન 5(a) અથવા [3 ગુણ]

ટ્રીના ઇન-ઓર્ડર ટ્રાવર્સલનો અલ્ગોરિધમ લખો.

ઉત્તર:

In-order Traversal અલ્ગોરિધમ:

```
def inorder_traversal(root):
    if root is not None:
        inorder_traversal(root.left)    # ડાબું
        print(root.data)                # Root
        inorder_traversal(root.right)   # જમણું
```

અલ્ગોરિધમ પગલાં:

પગલું	ક્રિયા	ક્રમ
1	ડાબા subtree ને traverse કરો	Recursive call
2	Root node ની મુલાકાત લો	ડેટા પ્રોસેસ કરો
3	જમણા subtree ને traverse કરો	Recursive call

Traversal ક્રમ: ડાબું → Root → જમણું

ગુણધર્મો:

- **BST ગુણધર્મ**: In-order સૉર્ટેડ sequence આપે છે
- **Time Complexity**: O(n)
- **Space Complexity**: O(h) જ્યાં h એ height છે

ઉદાહરણ Tree પરિણામ:

```
    Tree:      50
              /  \
            30    70
           / \   / \
         20  40 60  80

In-order: 20, 30, 40, 50, 60, 70, 80
```

મેમરી ટ્રીક: "In-order: ડાબું, Root, જમણું"

પ્રશ્ન 5(b) અથવા [4 ગુણ]

Search ની વ્યાખ્યા આપો? લિસ્ટનો ઉપયોગ કરીને Linear search માટેનો અલ્ગોરિધમ લખો.

ઉત્તર:

Search વ્યાખ્યા:
ડેટા સ્ટ્રક્ચરમાં ચોક્કસ એલિમેન્ટ શોધવાની અથવા એલિમેન્ટ અસ્તિત્વમાં છે કે નહીં તે ચકાસવાની પ્રક્રિયા.

Linear Search અલ્ગોરિધમ:

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # મળ્યું તો index પરત કરો
    return -1  # ન મળ્યું તો -1 પરત કરો
```

અલ્ગોરિધમ લક્ષણો:

લક્ષણ	વર્ણન	મૂલ્ય
પદ્ધતિ	ક્રમિક ચકાસણી	એલિમેન્ટ દર એલિમેન્ટ
Time Complexity	O(n)	રેખીય સમય
Space Complexity	O(1)	Constant space
ડેટા આવશ્યકતા	કોઈપણ ક્રમ	અનસૉર્ટેડ ડેટા ચાલે છે

અલ્ગોરિધમ પગલાં:

1. પ્રથમ એલિમેન્ટથી શરૂ કરો
2. દરેક એલિમેન્ટને target સાથે સરખાવો

3. જો મેચ મળે, index પરત કરો
4. જો અંત પહોંચે, -1 પરત કરો

મેમરી ટ્રીક: "Linear Search રેખીય રીતે લિસ્ટ્સમાં જુએ છે"

પ્રશ્ન 5(c) અથવા [7 ગુણ]

વ્યાખ્યા આપો: a) પાથ b). લીફ નોડ. નીચે આપેલ માહિતી ઉપરથી binary search tree બનાવો. 60, 40, 37,31,59,21,65,30

ઉત્તર:

વ્યાખ્યાઓ:

શબ્દ	વ્યાખ્યા	લક્ષણો
Path	એક નોડથી બીજા નોડ સુધીના nodes ની શ્રેણી	edges દ્વારા જોડાયેલ
Leaf Node	કોઈ બાળકો ન હોય તેવો નોડ	કોઈ ડાબું કે જમણું બાળક નથી

BST બનાવટ માટે: 60, 40, 37, 31, 59, 21, 65, 30

પગલાં પ્રમાણે બનાવટ:

```
પગલું 1: 60 Insert કરો (Root)
        60

પગલું 2: 40 Insert કરો (40 < 60, ડાબે જાઓ)
        60
       /
      40

પગલું 3: 37 Insert કરો (37 < 60, ડાબે; 37 < 40, ડાબે)
        60
       /
      40
     /
    37

પગલું 4: 31 Insert કરો (31 < 60, ડાબે; 31 < 40, ડાબે; 31 < 37, ડાબે)
        60
       /
      40
     /
    37
   /
  31

પગલું 5: 59 Insert કરો (59 < 60, ડાબે; 59 > 40, જમણે)
        60
       /
```

```
      40
     / \
    37  59
   /
  31

પગલું 6: 21 Insert કરો (21 < 60, ડાબે; 21 < 40, ડાબે; 21 < 37, ડાબે; 21 < 31, ડાબે)
        60
       /
      40
     / \
    37  59
   /
  31
 /
21

પગલું 7: 65 Insert કરો (65 > 60, જમણે જાઓ)
        60
       / \
      40  65
     / \
    37  59
   /
  31
 /
21

પગલું 8: 30 Insert કરો (30 < 60, ડાબે; 30 < 40, ડાબે; 30 < 37, ડાબે; 30 < 31, ડાબે; 30 > 21, જમણે)
        60
       / \
      40  65
     / \
    37  59
   /
  31
 / \
21  30
```

અંતિમ BST સ્ટ્રક્ચર:

લેવલ	Nodes	પ્રકાર
0	60	Root
1	40, 65	Internal
2	37, 59	Internal, Internal
3	31	Internal
4	21	Internal
5	30	Leaf

Leaf Nodes: 30, 59, 65

મેમરી ટ્રીક: "BST બિલ્ડિંગ Binary Search Tree નિયમોને અનુસરે છે"