

نام و نام خانوادگی	پارمیدا فهندژ
نام درس	ساختمان داده

سوال (۱)

بعد از اجرای این دستورات، دو گرهی جدید «۶» و «۰» به لیست اضافه می‌شوند، به طوری که گرهی ۶ بین head و ۱ قرار می‌گیرد و گرهی ۰ بین ۵ و tail. شکل نهایی لیست می‌شود:

head ↔ 6 ↔ 1 ↔ 2 ↔ 3 ↔ 4 ↔ 5 ↔ 0 ↔ tail

سوال (۲)

الف) لیست پیوندی دوطرفه بهینه (XOR-Linked List) چیست و چگونه کار می‌کند؟

* در یک DLL معمولی، هر گره دو اشاره‌گر دارد:

* prev ← آدرس گرهی قبلی

* next → آدرس گرهی بعدی

* در یک لیست پیوندی دوطرفه بهینه یا XOR-Linked List برای صرفه‌جویی در حافظه، به جای دو اشاره‌گر از یک فیلد واحد به نام npx استفاده می‌کنیم:

$$npx = \text{address}(\text{prev}) \oplus \text{address}(\text{next})$$

یعنی هر گره تنها یک مقدار ذخیره می‌کند که حاصل XOR آدرس گره‌های قبلی و بعدی است.

* **طریقهٔ پیمایش** (مثال پیمایش رو به جلو):

1. فرض کنید قبلاً در گره prev بوده‌ایم و اکنون در گره curr.

2. npx در curr برابر است با $\text{address}(\text{prev}) \oplus \text{address}(\text{next})$.


3. با محاسبه


$$\text{next} = npx(\text{curr}) \oplus \text{address}(\text{prev})$$

آدرس گرهی بعدی (next) به دست می‌آید.

4. سپس $prev \leftarrow curr$ و $curr \leftarrow next$ و همین روند را تکرار می‌کنیم.

* مزایا و ملاحظات

*  * مزیت اصلی: * هر گره به جای دو اشاره‌گر فقط یک اشاره‌گر (یا یک مقدار npx) نگه می‌دارد \Rightarrow کاهش مصرف حافظه

*  * محدودیت: * نیاز به نگهداری آدرس گرهی قبلی هنگام پیمایش، دیباگ دشوارتر، امکان استفاده‌ی هم‌زمان از چند اشاره‌گر خارجی محدودتر

ب) مقایسه پیچیدگی زمانی عملیات‌ها

عملیات	DLL معمولی	DLL بهینه (XOR)	توضیح
پیمایش (حرکت یک گام)	$O(1)$	$O(1)$	فقط یک XOR اضافی روی آدرس‌ها
درج در ابتدای لیست (head)	$O(1)$	$O(1)$	تنظیم اشاره‌گرها / npx
درج در انتهای لیست (tail)	$O(1)$	$O(1)$	مشابه
حذف گره‌ای که اشاره‌گرش داریم	$O(1)$	$O(1)$	فقط به prev و next (یا npx) نیاز دارد
جستجوی عنصر با کلید مشخص	$O(n)$	$O(n)$	باید پیمایش کامل کند
دسترسی تصادفی بر اساس اندیس	$O(n)$	$O(n)$	مانند DLL معمولی

< نکته: در عمل، XOR-Linked List به یک عملیات XOR اضافی در هر گام نیاز دارد اما این هم‌چنان $O(1)$ است.

✨ جمع‌بندی:

* لیست پیوندی دوطرفه بهینه (XOR) حافظه‌ی هر گره را با یکی کردن دو اشاره‌گر صرفه‌جویی می‌کند.

* از نظر **پیچیدگی زمانی**، تمام عملیات اصلی (درج، حذف، پیمایش) هم‌چنان $O(1)$ هستند و جستجو/دسترسی $O(n)$ ، درست مانند DLL معمولی.

در اینجا یک پیاده‌سازی ساده به زبان پایتون داریم که با یک بار پیمایش لیست ($O(n)$) و استفاده از یک مجموعه برای پیگیری مقادیر دیده‌شده ($O(n)$ حافظه) تمام گره‌های تکراری را حذف می‌کند:

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

def remove_duplicates(head: Node) -> Node
    """
    لیست پیوندی head را می‌گیرد و همه تکراری‌ها را (جز اولین ظهور هر مقدار)
    حذف می‌کند و سر لیست منحصربه‌فرد را برمی‌گرداند.
    """
    seen = set
    prev = None
    curr = head

    while curr
        if curr.val in seen
            # اگر این مقدار قبلاً دیده شده، گره را از زنجیره بیرون می‌کشیم
            prev.next = curr.next
        else
            # در غیر این صورت به مجموعه اضافه‌اش می‌کنیم
            seen.add(curr.val)
            prev = curr
            curr = curr.next

    return head

### مثال کاربردی

# ساخت لیست نمونه: 1 -> 3 -> 3 -> 2 -> 2 -> 4 -> 1
vals = [1, 3, 3, 2, 2, 4, 1]
nodes = [Node(v) for v in vals]
for i in range(len(nodes)-1)
```

```
nodes[i].next = nodes[i+1]
head = nodes[0]
```

```
# حذف تکراری‌ها
head = remove_duplicates(head)
```

```
# پیمایش و چاپ نتیجه
curr = head
while curr:
    print(curr.val, end=" → " if curr.next else " → None\n")
    curr = curr.next
```

خروجی:

None → 4 → 2 → 3 → 1

< پیچیدگی زمانی: $O(n)$
< پیچیدگی فضایی: $O(n)$ به خاطر نگهداری مجموعهٔ مقادیر دیده‌شده

در صورتی که بخواهید بدون حافظهٔ اضافی کار کنید (مثلاً $O(1)$ فضا)، می‌توانید از دو حلقه‌ی تو در تو استفاده کنید تا هر بار برای هر گره، در بقیه لیست جستجو کنید (پیچیدگی زمانی $O(n^2)$).

((سوال ۵))

در این پیاده‌سازی از دو پشته (stack) برای نگهداری تاریخچهٔ عملیات «undo» و «redo» استفاده می‌کنیم. هر بار که عملیاتی روی لیست (درج یا حذف گره) انجام می‌دهیم، معکوس آن را در پشتهٔ undo نگه می‌داریم و پشتهٔ redo را خالی می‌کنیم.

```
from collections import namedtuple
```

```
# تعریف گره لیست
class Node
```

```

        :def __init__(self, val)
            self.val = val
            self.prev = None
            self.next = None

undo/redo یک عملیات برای #
        'delete' یا 'op': 'insert #
        node: مرجع به گره درگیر #
        prev, next: گره‌های قبل و بعد آن در زمان عملیات
Action = namedtuple('Action', ['op', 'node', 'prev', 'next'])

class UndoableDoublyLinkedList
    :def __init__(self)
        sentinel head/tail برای ساده شدن درج/حذف لبه‌ها
        self.head = Node(None)
        self.tail = Node(None)
        self.head.next = self.tail
        self.tail.prev = self.head

        [] = self.undo_stack # پشته عملیات قابل بازگشت
        [] = self.redo_stack # پشته عملیات قابل تکرار

    :def _link_between(self, node, prev, nxt)
        """گره node را بین prev و nxt درج می‌کند."""
        node.prev, node.next = prev, nxt
        prev.next, nxt.prev = node, node

    :def insert_after(self, ref_node, new_node)
        """
        گره new_node را بعد از ref_node درج می‌کند
        و یک Action معکوس (حذف new_node) را در undo_stack می‌گذارد.
        """
        # 1 درج واقعی
        self._link_between(new_node, ref_node, ref_node.next)
        # 2 ثبت معکوس در undo

```

```

        )self.undo_stack.append
        ,Action(op='delete', node=new_node
        ((prev=ref_node, next=ref_node.next
        # پس از هر عملیات جدید، redo پاک می‌شود
        ()self.redo_stack.clear

```

```

        :def delete_node(self, node)
            """
            گره node را از لیست حذف می‌کند
            و یک Action معکوس (درج node) را در undo_stack می‌گذارد.
            """

```

```

        prev, nxt = node.prev, node.next
        prev.next, nxt.prev = nxt, prev
        # ثبت معکوس در undo
        )self.undo_stack.append
        (Action(op='insert', node=node, prev=prev, next=nxt)
        ()self.redo_stack.clear

```

```

        :def undo(self)
            """آخرین تغییر را برمی‌گرداند (undo)."""
            :if not self.undo_stack
                print("Nothing to undo")
                return
            ()action = self.undo_stack.pop
            # معکوس کردن
            :if action.op == 'insert
                # عمل insert را undo می‌کنیم یعنی delete
                self._link_between(action.node, action.prev, action.next)
                # حالا معکوس این undo را برای redo ذخیره می‌کنیم
                ,redo_action = Action('delete
                    ,action.node
                    ,action.prev
                    (action.next
                'else: # action.op == 'delete
                # عمل delete را undo می‌کنیم یعنی دوباره insert

```

```

        action.prev.next = action.node
        action.next.prev = action.node
        action.node.prev, action.node.next = action.prev, action.next
        , 'redo_action = Action('insert
            , action.node
            , action.prev
            (action.next

self.redo_stack.append(redo_action)

```

```

        :def redo(self)
        """آخرین undo شده را دوباره اجرا می کند (redo)."""
        :if not self.redo_stack
        print("Nothing to redo")
        return

        ()action = self.redo_stack.pop
        redo_stack # دقیقاً مثل undo اما این بار روی redo_stack
        :if action.op == 'insert
        self._link_between(action.node, action.prev, action.next)
        ,opposite = Action('delete', action.node
            (action.prev, action.next
            'else: # action.op == 'delete
        action.prev.next, action.next.prev = action.next, action.prev
        ,opposite = Action('insert', action.node
            (action.prev, action.next

        self.undo_stack.append(opposite)

```

```

        :def traverse(self)
        """ابزار کمکی برای دیده بانیش لیست."""
        cur = self.head.next
        [] = vals
        :while cur is not self.tail

```

```

vals.append(cur.val)
cur = cur.next
return vals

```

----- مثال استفاده ----- #

```
()lst = UndoableDoublyLinkedList
```

درج چند گره

```

n1 = Node(1)
n2 = Node(2)
n3 = Node(3)

```

لیست: 1 lst.insert_after(lst.head, n1)

لیست: 1 ↔ 2 lst.insert_after(n1, n2)

لیست: 1 ↔ 2 ↔ 3 lst.insert_after(n2, n3)

```
print(lst.traverse()) # → [1, 2, 3]
```

حذف گرهی 2

```

lst.delete_node(n2)
print(lst.traverse()) # → [1, 3]

```

undo (بازگشت حذف گرهی 2)

```

()lst.undo
print(lst.traverse()) # → [1, 2, 3]

```

redo (دوباره حذف گرهی 2)

```

()lst.redo
print(lst.traverse()) # → [1, 3]

```

1. هر بار که insert_after یا delete_node فراخوانی می‌شود:

* خود عمل را انجام می‌دهیم.

* معکوس آن عمل (یعنی اگر درج بود یک حذف، و اگر حذف بود یک درج) را در undo_stack ذخیره می‌کنیم.
* پشته redo_stack را پاک می‌کنیم (زیرا بعد از یک تغییر جدید، مسیر redo قبلی بی‌معنی است).

2. undo()

* یک Action از undo_stack بیرون می‌کشد.
* معکوس آن را روی لیست اجرا می‌کند.
* یک Action جدید (معکوس معکوس) در redo_stack می‌گذارد تا بتوان بعداً با redo() آن را دوباره تکرار کرد.

3. redo()

* مثل undo اما روی redo_stack کار می‌کند و نتیجه را دوباره در undo_stack ذخیره می‌کند.

با این مکانیسم شما همیشه می‌توانید «قدم عقب» (undo) و «قدم جلو» (redo) را تا هر عمقی که در پشته‌ها دارید انجام بدهید.

سوال (۶)

در ادامه یک پیاده‌سازی ساده به زبان پایتون می‌بینید که دقیقاً خواسته‌های مسئله (۱-۴) را برآورده می‌کند. در این پیاده‌سازی:

* هر واگن یک شیء Node است با فیلدهای carriage_id, passenger_count, cargo_weight و اشاره‌گر next.
* کلاس Train فهرست پیوندی را نگه می‌دارد و متدهای درج، حذف، یافتن سنگین‌ترین واگن و معکوس کردن ترتیب واگن‌ها را پیاده می‌کند.

```
:class Node
```

```
,def __init__(self, carriage_id: int
```

```
,passenger_count: int
```

```
:(cargo_weight: float
```

```
self.carriage_id = carriage_id
```

```
self.passenger_count = passenger_count
```

```
self.cargo_weight = cargo_weight
```

```
self.next = None
```

```
:class Train
```

```
:def __init__(self)
```

```
head points to the first wagon after locomotive #
```

```
self.head = None
```

```
:def insert_after(self, ref_id: int, new_node: Node) -> bool
    """
```

```
    Insert new_node immediately after the wagon whose
    .carriage_id == ref_id. Return True on success, False if not found
    """
```

```
        cur = self.head
        :while cur
        :if cur.carriage_id == ref_id
        new_node.next = cur.next
        cur.next = new_node
        return True
        cur = cur.next
    return False # ref_id not found
```

```
:def delete_by_id(self, carriage_id: int) -> bool
    """
```

```
    .Remove the wagon whose carriage_id == carriage_id
    .Return True on success, False if not found
    """
```

```
        prev = None
        cur = self.head
        :while cur
        :if cur.carriage_id == carriage_id
            :if prev
                prev.next = cur.next
            :else
                deleting the very first wagon #
                self.head = cur.next
                return True
        prev, cur = cur, cur.next
        return False # not found
```

```
:def find_heaviest(self) -> Node
```

```
        """
        .Return the Node with the maximum cargo_weight
        .If list is empty, return None
        """
```

```
        :if not self.head
            return None
        heaviest = self.head
        cur = self.head.next
        :while cur
        :if cur.cargo_weight > heaviest.cargo_weight
            heaviest = cur
            cur = cur.next
        return heaviest
```

```
        :def reverse(self)
            """
            .Reverse the linked list of wagons in-place
            """
```

```
            prev = None
            cur = self.head
            :while cur
                nxt = cur.next
                cur.next = prev
                prev = cur
                cur = nxt
            self.head = prev
```

```
        :def traverse(self) -> list
            """
            .Return a list of carriage_ids from front to back
            """
```

```
            [] = result
            cur = self.head
            :while cur
                result.append(cur.carriage_id)
```

```
cur = cur.next
return result
```

----- Example Usage ----- #

```
build initial train: wagons [101, 102, 103] #
()train = Train
train.head = Node(101, passenger_count=30, cargo_weight=1000.0)
n2 = Node(102, passenger_count=40, cargo_weight=1200.5)
n3 = Node(103, passenger_count=20, cargo_weight= 950.0)
train.head.next = n2
n2.next = n3

print("Initial:", train.traverse())
Initial: [101, 102, 103] → #

insert wagon 104 after 102 (1 #
train.insert_after(102, Node(104, passenger_count=25, cargo_weight=1100.0))
print("After insert:", train.traverse())
After insert: [101, 102, 104, 103] → #

delete wagon 101 (2 #
train.delete_by_id(101)
print("After delete 101:", train.traverse())
After delete 101: [102, 104, 103] → #

find heaviest (3 #
()heavy = train.find_heaviest
print(f"Heaviest wagon: id={heavy.carriage_id}, weight={heavy.cargo_weight}")
Heaviest wagon: id=102, weight=1200.5 → #

reverse for return trip (4 #
()train.reverse
print("Reversed:", train.traverse())
Reversed: [103, 104, 102] → #
```

- `insert_after(ref_id, new_node)` ✓ * درج بعد از شناسه مشخص
- `delete_by_id(id)` ✓ * حذف بر اساس شناسه
- `find_heaviest()` ✓ * بازگرداندن واگن با بیشترین بار
- `reverse()` ✓ * معکوس کردن ترتیب لیست

تمام عملیات درج/حذف/جستجو یک گام ($O(n)$ در بدترین حالت برای پویش تا انتها) و معکوس ($O(n)$) اجرا می‌شوند.

سوال (۱۰)

```
import hashlib
```

```
import json
```

```
import time
```

```
from typing import Any, List, Optional
```

```
:class Block
```

```
,def __init__(self
```

```
,transaction_data: Any
```

```
:("" = previous_hash: str
```

```
self.transaction_data = transaction_data
```

```
()self.timestamp = time.time
```

```
self.previous_hash = previous_hash
```

```
()self.current_hash = self.compute_hash
```

```
self.next: Optional["Block"] = None
```

```
:def compute_hash(self) -> str
```

```
"""
```

```
.Compute SHA-256 over the block's contents
```

```
We serialize transaction_data as JSON so that dicts/lists
```

```
.always hash in a consistent order
```

```
"""
```

```
})block_string = json.dumps
```

```
,transaction_data": self.transaction_data"
```

```

        ,timestamp": self.timestamp"
        previous_hash": self.previous_hash"
        )sort_keys=True).encode,{
    ()return hashlib.sha256(block_string).hexdigest

class Blockchain
    def __init__(self)
        create the genesis block #
genesis = Block(transaction_data="Genesis Block", previous_hash="0")
        self.head = genesis # first block
        self.tail = genesis # last block for O(1) append

    def add_block(self, transaction_data: Any) -> Block
        """
        .Append a new block containing transaction_data
        """
        new_block = Block(transaction_data, previous_hash=self.tail.current_hash)
        self.tail.next = new_block
        self.tail = new_block
        return new_block

    def is_chain_valid(self) -> bool
        """
        Verify that each block's previous_hash matches the
        actual hash of the previous block, and that each block's
        .stored current_hash is correct
        """
        cur = self.head
        while cur.next
            recompute hash and compare #
            :()if cur.current_hash != cur.compute_hash
                return False
            :if cur.next.previous_hash != cur.current_hash
                return False
            cur = cur.next

```

```

        check last block too #
return (cur.current_hash == cur.compute_hash())

def traverse(self) -> List[Block]
    """
    .Return the entire chain as a list, from genesis to tail
    """
    blocks = []
    cur = self.head
    while cur:
        blocks.append(cur)
        cur = cur.next
    return blocks

----- Example Usage ----- #

if __name__ == "__main__":
    chain = Blockchain()

    Append some blocks #
    chain.add_block({"from": "Alice", "to": "Bob", "amount": 50})
    chain.add_block({"from": "Bob", "to": "Carol", "amount": 25})
    chain.add_block({"from": "Carol", "to": "Dave", "amount": 12.5})

    Traverse and print #
    for idx, blk in enumerate(chain.traverse()):
        print(f"Block {idx}")
        print(f"  Timestamp: {blk.timestamp}")
        print(f"  Data: {blk.transaction_data}")
        print(f"  Prev Hash: {blk.previous_hash}")
        print(f"  Curr Hash: {blk.current_hash}\n")

    Validate #
    print("Chain valid?", chain.is_chain_valid())

```

توضیحات

* هر Block در لحظه ساخت:

```
1. timestamp = time.time()
2. previous_hash از بلاک آخر زنجیره می‌آید
3. current_hash = SHA-256(transaction_data ↑ timestamp ↑ previous_hash)
```

* کلاس Blockchain:

* با یک بلاک «خاستگاه» (genesis) شروع می‌شود که `previous_hash = "0"`.

* `add_block(...)` یک بلاک جدید می‌سازد، به یار اشاره تک‌جهتی می‌دهد و آن را دم لیست می‌چسباند.

* `is_chain_valid()` اطمینان می‌دهد که زنجیره تغییرناپذیر است (هر دو شرط مقداردهی مجدد هش و اتصال بلاک‌ها).

* این مدل بسیار ساده است و برای آموزش مناسب است. در محیط واقعی باید مکانیزم‌های پیچیده‌تری مثل ماینینگ، Proof-of-Work/Stake، و دفتر کل توزیع‌شده اضافه شود.

(سوال ۸)

در ادامه یک پیاده‌سازی ساده و کامل با پایتون می‌بینید که:

1. درخت دروس را با کلاس `TreeNode` می‌سازد (هر گره یک درس با کد یا نام مشخص دارد و لیستی از فرزندان = پیش‌نیازها).
2. کلاس `CourseTree` متدهای پیمایش `Postorder`, `Preorder`, `Inorder`، محاسبه اندازه زیردرخت و جست‌وجو بر اساس برچسب را دارد.
3. مثال عملی با چند درس و پیش‌نیاز نمایش می‌دهد.

```
:class TreeNode
    :def __init__(self, code: str)
        self.code = code
        [] = self.children: list[TreeNode]

    :def add_prerequisite(self, node: "TreeNode")
        """اضافه کردن یک پیش‌نیاز (فرزند)"""
        self.children.append(node)
```



```

class CourseTree
: def __init__(self, root: TreeNode)
    self.root = root

: def postorder(self, node: TreeNode = None) -> list[str]
    """پیمایش پس‌سرتی: اول همه پیش‌نیازها، سپس خود درس"""
    : if node is None
        node = self.root
        [] = result
        : for child in node.children
            result += self.postorder(child)
        result.append(node.code)
    return result

: def preorder(self, node: TreeNode = None) -> list[str]
    """پیمایش پیش‌سرتی: خود درس، سپس به ترتیب پیش‌نیازها"""
    : if node is None
        node = self.root
        result = [node.code]
        : for child in node.children
            result += self.preorder(child)
    return result

: def inorder(self, node: TreeNode = None) -> list[str]
    """
    پیمایش میانی برای درخت عمومی:
    - اگر فرزند وجود داشت، اول میانی فرزند اول،
    سپس خود گره،
    سپس میانی بقیه فرزندان.
    """
    : if node is None
        node = self.root
        [] = result
        : if node.children

```

```

# میانی روی فرزند اول
result += self.inorder(node.children[0])
result.append(node.code)
# میانی روی بقیه فرزندان
:for child in node.children[1:]
result += self.inorder(child)
return result

```

```

:def get_subtree_size(self, node: TreeNode = None) -> int
    """تعداد گره‌های زیردرخت شامل خود گره"""
    :if node is None
        node = self.root
        count = 1
    :for child in node.children
count += self.get_subtree_size(child)
return count

```

```

:def find_by_label(self, label: str, node: TreeNode = None) -> TreeNode | None
    """
        جست‌وجو به صورت DFS
        اگر گره با آن کد پیدا شد برگردانده شود.
    """
    :if node is None
        node = self.root
    :if node.code == label
        return node
    :for child in node.children
found = self.find_by_label(label, child)
    :if found
        return found
    return None

```

----- # ساخت یک درخت نمونه -----

تعریف گره‌ها

```
db = TreeNode("Database")
ds = TreeNode("Data Structures")
dm = TreeNode("Data Modeling")
se = TreeNode("Software Engineering")
arr = TreeNode("Arrays")
oop = TreeNode("OOP")
sql = TreeNode("SQL")
```

رابطهٔ پیش‌نیازها

```
db.add_prerequisite(ds)
db.add_prerequisite(dm)
db.add_prerequisite(se)
```

```
ds.add_prerequisite(arr)
se.add_prerequisite(oop)
dm.add_prerequisite(sql)
```

ساخت درخت

```
tree = CourseTree(db)
```

تست متدها

```
print("Postorder →", tree.postorder())
```

خروجی: ['Arrays', 'Data Structures', 'SQL', 'Data Modeling', 'OOP', 'Software Engineering', 'Database']

```
print("Preorder →", tree.preorder())
```

خروجی: ['Database', 'Data Structures', 'Arrays', 'Data Modeling', 'SQL', 'Software Engineering', 'OOP']

```
print("Inorder →", tree.inorder())
```

خروجی: ['Arrays', 'Data Structures', 'Database', 'SQL', 'Data Modeling', 'OOP', 'Software Engineering']

```
print("SubtreeSize(db) →", tree.get_subtree_size())  
# خروجی: 7 (تعداد کل گره‌ها)
```

```
node = tree.find_by_label("SQL")  
print("Found SQL? →", node.code if node else "Not found")  
# خروجی: Found SQL?
```

*** Postorder** (فهرست پیشنهادی برای «ابتدا درس‌های پیش‌نیاز، بعد درس اصلی»)
*** Preorder** (ترتیب «از بالا به پایین» برای برنامه‌ریزی تدریس)
*** Inorder** (برای ارزیابی عمق و تعادل عمومی)
*** get_subtree_size** (محاسبه تعداد دروس وابسته)
*** find_by_label** (یافتن گره دلخواه)