Week 07 Answers

پاسخنامه تکالیف هفته ۰۷

پاسخ سؤال اول :

این سؤال برای درک یکی از مهم‌ترین ساختارهای داده طراحی شده است؛ ساختاری که هدف آن ارائهٔ دسترسی بسیار سریع به داده‌هاست. مفهوم کلیدی این درس، «نگاشت کلید به یک ایندکس کوچک‌تر» با استفاده از تابع درهم‌سازی است.

الف) هدف اصلی استفاده از درهم‌سازی

هدف اصلی از به‌کارگیری جداول درهم‌سازی، دستیابی به **سرعت بالا در عملیات جستجو، درج و حذف** است؛ به‌طوری که این عملیات به‌طور میانگین دارای پیچیدگی زمانی

$$\dots O(1)$$

باشد. این سرعت نتیجهٔ تبدیل کلیدهای بزرگ یا پیچیده به آدرس‌های عددی کوچک و ثابت است.

ب) ویژگی‌های یک تابع درهم‌سازی مناسب

یک تابع درهم‌سازی خوب باید خصوصیات زیر را داشته باشد:

- **توزیع یکنواخت (Uniform Distribution):**

کلیدهای ورودی را باید تا حد امکان به‌صورت پخش‌شده روی خانه‌های جدول نگاشت کند. این کار احتمال برخورد را به‌طور چشمگیری کاهش می‌دهد.

- **محاسبهٔ سریع و سبک (Fast Computation):**

تابع باید از نظر زمانی ساده و سبک باشد تا محاسبهٔ هش خودش باعث کاهش کارایی ساختار داده نشود.(ویژگی‌های دیگری مثل وابستگی کم به الگوی ورودی نیز معمولاً ذکر می‌شوند.)

ج) برخورد (Collision) و علت اجتناب‌ناپذیر بودن آن

«برخورد» زمانی رخ می‌دهد که **دو کلید متفاوت** توسط تابع هش به **یک خانهٔ یکسان** در جدول نگاشت شوند.

علت اجتناب‌ناپذیر بودن برخورد این است که:

- فضای کلیدها معمولاً **بسیار بزرگ** یا حتی بی‌نهایت است،
- اما اندازهٔ جدول هش **محدود** می‌باشد.

به‌همین دلیل، طبق اصل «لانه‌کبوتری»، حتی بهترین تابع هش هم نمی‌تواند کاملاً از برخورد جلوگیری کند؛ بنابراین سیستم باید مکانیزم‌های مدیریت برخورد (مثل Chain کردن یا Open Addressing) داشته باشد.

پاسخ سوال دوم :

این مسئله برای درک نحوه‌ی عملکرد واقعی جداول درهم‌سازی در سناریوی ذخیره‌سازی داده‌ها طراحی شده است، به‌ویژه وقتی برخوردهای متعدد رخ می‌دهد. هدف این تمرین، بررسی نحوه‌ی شکل‌گیری زنجیره‌ها در جدول و تأثیر تابع درهم‌سازی بر توزیع داده‌هاست.

ویژگی‌های سیستم درهم‌سازی:

- اندازه‌ی جدول: 9
- تابع درهم‌سازی:

$$h(k) = (11k + 4) \bmod 9$$

با ساده‌سازی داریم:

چون $11 \bmod 9 = 2$

پس:

$$h(k) = (2k + 4) \bmod 9$$

- روش رفع تداخل: **زنجیره‌ای (Chaining)**

در این روش، تمام عناصر دارای هش برابر، در قالب یک لیست پیوندی یا زنجیره در همان خانه ذخیره می‌شوند.

مرحله‌ی محاسبات نگاشت شناسه‌ها (Hashing Process):

شناسه(k)	محاسبه‌ی h(k)	خانه‌ی نهایی	توضیح
67	(2×67 + 4) mod 9 = 138 mod 9 = **3**	خانه ۳	اولین عضو زنجیره
13	(2×13 + 4) mod 9 = 30 mod 9 = **3**	خانه ۳	برخورد با 67 → افزودن به زنجیره
49	(2×49 + 4) mod 9 = 102 mod 9 = **3**	خانه ۳	برخورد دیگر → افزودن به زنجیره
24	(2×24 + 4) mod 9 = 52 mod 9 = **7**	خانه ۷	بدون برخورد
40	(2×40 + 4) mod 9 = 84 mod 9 = **3**	خانه ۳	افزودن به زنجیرهٔ موجود
33	(2×33 + 4) mod 9 = 70 mod 9 = **7**	خانه ۷	برخورد با 24 → افزودن به زنجیره
58	(2×58 + 4) mod 9 = 120 mod 9 = **3**	خانه ۳	افزودن به زنجیرهٔ طولانی خانه ۳

شکل نهایی جدول درهم‌سازی (پس از ثبت همه شناسه‌ها):

خانه	زنجیره(Chaining)
0	—
1	—
2	—
3	[67 → 13 → 49 → 40 → 58]
4	—
5	—
6	—
7	[24 → 33]
8	—

تحلیل و نکات فنی:

- **عدم توزیع یکنواخت:**

تابع هش باعث تمرکز داده‌ها در خانه‌ی شماره ۳ شده و احتمالاً انتخاب ضریب ۱۱ چندان ایده‌آل نبوده است.

- **افزایش طول زنجیره:**

برخوردهای زیاد در یک خانه، باعث طولانی شدن زنجیره و افزایش زمان جستجو در آن خانه می‌شود.

- **بهبود بالقوه:**

می‌توان از توابع هش متفاوت (مثلاً مبتنی بر اعداد اول یا مدهای دیگر) برای بهبود توزیع استفاده کرد.

- **مرتبه‌ی زمانی میانگین:**

برای درج یا جستجو در این روش معمولاً $O(1+\alpha)$است،

که αضریب بار (load factor) نام دارد و نشان‌دهنده‌ی نسبت تعداد عناصر به اندازه‌ی جدول است.

پاسخ سوال سوم :

این تمرین برای درک رفتار **Linear Probing** در جداول درهم‌سازی طراحی شده است؛ به‌ویژه زمانی که برخوردهای پشت‌سرهم رخ می‌دهند و پدیده‌ی «خوشه‌سازی خطی (Primary Clustering)» شکل می‌گیرد.

مشخصات سیستم

- اندازه‌ی جدول: 8
- تابع درهم‌سازی:

$$h(x) = x \bmod 8$$

- روش رفع برخورد: **Linear Probing**

- حذف انجام نمی‌شود
- داده‌های ورودی به ترتیب:

19, 27, 36, 10, 64, 45

الف) محاسبات هش و محل نهایی کلیدها

۱) محاسبهٔ خانهٔ اولیه (Initial Slot)

$$19 \bmod 8 = 3$$

$$27 \bmod 8 = 3$$

$$36 \bmod 8 = 4$$

$$10 \bmod 8 = 2$$

$$64 \bmod 8 = 0$$

$$45 \bmod 8 = 5$$

۲) رفع برخورد با Linear Probing

در صورت اشغال خانه، به ترتیب خانه‌های بعدی بررسی می‌شود (با امکان wrap-around).

کلید	h(x)	وضعیت	محل نهایی	توضیح
19	3	آزاد	3	درج مستقیم
27	3	برخورد	4	اولین خانهٔ خالی بعد از ۳
36	4	برخورد	5	خانه ۴ پر است → خانه ۵ خالی
10	2	آزاد	2	درج مستقیم
64	0	آزاد	0	درج مستقیم

خانه ۵ پر است → خانه ۶ خالی	6	برخورد	5	**45**

جدول نهاییHash Table

Index : Value

64 : 0

— : 1

10 : 2

19 : 3

27 : 4

36 : 5

45 : 6

— : 7

ب) برخوردها و تعداد آنها

کلیدهای دارای برخورد:

- کلید **27** با **19** برخورد داشته است.
- کلید **36** با **27** برخورد داشته است.
- کلید **45** با **36** برخورد داشته است.

تعداد برخوردها:

تعداد کل برخوردها = 3

ج) مقایسهٔ رفتار Linear Probing با Chaining

اگر از **Chaining** استفاده می‌شد:

- تمام کلیدهایی که مقدار هش یکسان دارند در **یک لیست زنجیره‌ای** ذخیره می‌شدند.
- **نیازی به جستجوی خانه‌های بعدی نبود.**
- پدیدهٔ **Primary Clustering** ایجاد نمی‌شد؛ چون خانه‌های متوالی اشغال نمی‌شوند.

نتیجهٔ مقایسه:

- **تعداد برخوردهای مفهومی (یعنی هش برابر)** همان مقدار باقی می‌ماند.
- اما **تعداد عملیات اضافی** و **هزینهٔ جستجو/درج** در Chaining کمتر می‌شود.
- روش Chaining در این مثال کارایی بیشتری نسبت به Linear Probing دارد.

جواب سوال چهارم :

فایل A4.py را اجرا کنید.

این تمرین به طراحی و پیاده‌سازی یک **جدول درهم‌سازی (Hash Table)** با ویژگی‌های مقیاس‌پذیری بالا می‌پردازد. هدف اصلی، فراهم کردن امکان ذخیره‌سازی و دسته‌بندی سریع سفارش‌های آنلاین غذا با استفاده از **آرایه‌های پویا** و **مدیریت برخورد (Collision Handling) با روش زنجیره‌ای (Chaining)** است.

مفاهیم کلیدی و معماری سیستم

۱. ساختار دادهٔ اصلی:

- **جدول درهم‌سازی (Hash Table):** ساختاری کلید-مقدار که با استفاده از تابع هش، کلیدها را به ایندکس‌های جدول نگاشت می‌کند.
- **ظرفیت اولیه:** 5

- **اندازهٔ جدول پویا:** در صورت رسیدن تعداد عناصر به ظرفیت جدول، اندازهٔ جدول دو برابر می‌شود تا از افت کارایی جلوگیری شود.

۲. تابع درهم‌سازی رشته‌ای (String Hashing Function):

- **فرمول:**

$$\text{hash(key)} = \sum_{i=0}^{n-1} \text{ASCII}(\text{char}_i) \bmod \text{tableSize}$$

که در آن:

- key: رشتهٔ مربوط به نوع غذا (مثلاً "pizza").
- char_i: کاراکتر i-ام در رشتهٔ کلید.
- ASCII(char): مقدار کد اسکی کاراکتر.
- tableSize: اندازهٔ فعلی جدول درهم‌سازی.
- **هدف:** محاسبهٔ ایندکس جدول برای هر سفارش بر اساس نام غذا. این تابع با جمع مقادیر اسکی کاراکترها و سپس محاسبهٔ باقیماندهٔ تقسیم بر اندازهٔ جدول، تلاش می‌کند تا داده‌ها را به‌طور نسبتاً خوبی توزیع کند.

۳. مدیریت برخورد (Collision Handling):

- **روش: زنجیره‌ای (Chaining)**
- **پیاده‌سازی:** هر خانه (slot) از جدول درهم‌سازی به‌جای نگهداری یک مقدار، یک **لیست (List)** را نگهداری می‌کند.
- **نحوهٔ عملکرد:** اگر چند سفارش (کلید) مقدار هش یکسانی داشته باشند، همگی به لیستِ همان خانه اضافه می‌شوند. این روش به جدول اجازه می‌دهد تا تعداد نامحدودی عنصر را ذخیره کند (البته با افت احتمالی کارایی در صورت طولانی شدن زنجیره‌ها).

- **قابلیت تکرار:** سفارش‌های تکراری مجاز هستند و هر کدام به‌صورت یک عنصر جداگانه در زنجیره ذخیره می‌شوند.

۴. مقیاس‌پذیری خودکار (Rehashing & Automatic Resizing):

- **شرط فعال‌سازی:** زمانی که تعداد کل سفارش‌ها (count) برابر یا بیشتر از اندازهٔ جدول (size) شود.
- **مراحل:**

1. **دو برابر کردن اندازهٔ جدول:** self.size *= 2
2. **ایجاد جدول جدید:** جدولی با اندازهٔ دو برابر و خانه‌های خالی ایجاد می‌شود.
3. **هش مجدد (Rehashing):** تمام سفارش‌های موجود از جدول قدیمی، مجدداً با استفاده از تابع هش جدید (که tableSize آن دو برابر شده) هش شده و در جدول جدید درج می‌شوند.

- **اهمیت:** این مکانیزم تضمین می‌کند که حتی با افزایش چشمگیر تعداد سفارش‌ها، متوسط زمان عملیات درج و جستجو نزدیک به $O(1)$باقی بماند.

نمونهٔ ورودی و خروجی (Input 1)

ورودی:

pizza, burger, salad

فرآیند پیاده‌سازی:

1. **جدول اولیه:** table = [[], [], [], [], []] ,count = 0 ,size = 5
2. **درج "pizza":**

- hash("pizza") = (112+105+122+122+97) % 5 = 558 % 5 = 3
- table[3].append("pizza")

- count = 1, table = [[], [], [], ['pizza'], []]

3. **درج "burger":**

- hash("burger") = (98+117+114+103+101+114) % 5 = 647 % 5 = 2
- table[2].append("burger")
- count = 2, table = [[], [], ['burger'], ['pizza'], []]

4. **درج "salad":**

- hash("salad") = (115+97+108+97+100) % 5 = 517 % 5 = 2
- table[2].append("salad")
- count = 3, table = [[], [], ['burger', 'salad'], ['pizza'], []]

در این مرحله count (3) کمتر از size (5) است، لذا Rehash انجام نمی‌شود.

خروجی نهایی:

خانه 0: []

خانه 1: []

خانه 2: ['burger', 'salad']

خانه 3 : ['pizza']

خانه 4: []

توضیحات تکمیلی:

- **کارایی:** با استفاده از این ساختار، عملیات درج و جستجو به‌طور متوسط در زمان $O(1)$انجام می‌شود، زیرا تابع هش مقادیر را به‌خوبی توزیع می‌کند و اندازهٔ زنجیره‌ها کوتاه می‌ماند. در بدترین حالت (که همهٔ کلیدها به یک خانه هش شوند)، زمان عملیات به $O(n)$می‌رسد، اما با مکانیزم Rehash، این حالت به‌ندرت اتفاق می‌افتد.

- **کاربرد:** این پیاده‌سازی اساس بسیاری از سیستم‌های ذخیره‌سازی دادهٔ سریع و مقیاس‌پذیر است، مانند کش‌ها (caches)، پایگاه‌های دادهٔ NoSQL، و دیکشنری‌های زبان‌های برنامه‌نویسی.

پاسخ سوال پنجم :

۱. توضیحات فنی سوال و هدف آموزشی:

این مسئله با هدف آموزش پیاده‌سازی یک ساختار داده‌ی کارآمد، یعنی جدول هش (Hash Table) با قابلیت تغییر اندازه پویا (Dynamic Resizing) و مدیریت برخورد به روش زنجیره‌ای (Separate Chaining)، طراحی شده است. در این پیاده‌سازی، شناسه‌ها (Ticket IDs) به صورت رشته‌ای هستند و برای درج آن‌ها در جدول هش، از یک تابع هش سفارشی بر اساس مجموع کدهای ASCII و عملگر mod استفاده می‌شود. قابلیت Rehash (هش مجدد) با دو برابر شدن ظرفیت جدول، تضمین می‌کند که جدول هش حتی با افزایش تعداد عناصر، کارایی خود را حفظ کند. تاکید بر این است که از ساختارهای آماده پایتون مانند dict استفاده نشود تا مفاهیم پیاده‌سازی ساختمان داده به صورت عملی تمرین شوند.

۲. ساختمان داده‌ها و متغیرهای کلیدی:

- DynamicHashTable (کلاس اصلی): ساختار جدول هش را پیاده‌سازی می‌کند.
- size: ظرفیت *فعلی* جدول هش (تعداد سطل‌ها). این مقدار هنگام Rehash دو برابر می‌شود.
- count: تعداد کل عناصری که در حال حاضر در جدول هش ذخیره شده‌اند.
- table: لیستی از لیست‌ها (لیست‌های زنجیره). هر عنصر این لیست، نمایانگر یک "سطل" (bucket) در جدول هش است و خود یک لیست پایتون است که شناسه‌های هش شده را (به ترتیب ورود) در خود نگه می‌دارد.
- _hash(key): تابع هش سفارشی که یک کلید متنی (key) را می‌گیرد و با جمع کردن کدهای ASCII حروف کلید و سپس محاسبه باقی‌مانده تقسیم بر self.size فعلی، اندیس مناسب را تعیین می‌کند.

- insert(key): متد اصلی درج کلید در جدول هش. ابتدا بررسی می‌کند که آیا self.count برابر یا بیشتر از self.size شده است یا خیر. اگر چنین باشد، _rehash() فراخوانی می‌شود. سپس کلید با استفاده از _insert_without_rehash در سطل مربوطه درج شده و self.count افزایش می‌یابد.
- _rehash(): متد مسئول تغییر اندازه جدول هش. self.size دو برابر می‌شود. یک جدول جدید (self.table) با اندازه‌ی دو برابر ایجاد می‌شود. سپس تمام کلیدهای موجود در جدول قدیمی، مجدداً با استفاده از تابع هش و اندازه‌ی جدید (self.size)، هش شده و در جدول جدید درج می‌شوند. self.count نیز در این متد صفر شده و سپس با درج مجدد عناصر، دوباره مقداردهی می‌شود.
- _insert_without_rehash(key): متد داخلی برای درج کلید بدون بررسی شرط Rehash، که عمدتاً توسط rehash_ استفاده می‌شود.
- Display(): متد نمایش وضعیت نهایی جدول هش، شامل خانه‌ها و محتویات آن‌ها.

۳. منطق الگوریتم و مراحل اجرا:

کد با ایجاد یک DynamicHashTable با ظرفیت اولیه ۵ عمل می‌کند. دنباله‌ی شناسه‌های متنی ui, api, timeout, payment, login, bug, crash, login به ترتیب درج می‌شوند:

- درج ui (count=1, size=5): هش محاسبه شده و درج می‌شود.
- درج api (count=2, size=5): هش محاسبه شده و درج می‌شود.
- درج timeout (count=3, size=5): هش محاسبه شده و درج می‌شود.
- درج payment (count=4, size=5): هش محاسبه شده و درج می‌شود.
- درج login (count=5, size=5): در این مرحله، self.size =< self.count برقرار است. بنابراین، _rehash() فراخوانی می‌شود:
- self.size به ۱۰ تغییر می‌کند.
- یک جدول جدید با ۱۰ سطل ایجاد می‌شود.

- کلیدهای قبلی (ui, api, timeout, payment, login) با تابع هش جدید (mod 10) دوباره هش شده و در سطل‌های مربوطه در جدول ۱۰ تایی درج می‌شوند. self.count دوباره مقداردهی می‌شود (۵).
- درج bug (count=6, size=10): هش محاسبه شده و درج می‌شود.
- درج crash (count=7, size=10): هش محاسبه شده و درج می‌شود.
- درج login (مجدد) (count=8, size=10): این کلید در خانه‌ای که تابع هش آن را تعیین می‌کند، درج می‌شود.

نکته مهم در مورد خروجی نمونه: تابع هش تعریف شده در سوال (∑ASCII(key))modtableSize) و اجرای کد شما، خروجی متفاوتی نسبت به خروجی نمونه ارائه شده در سوال اصلی تولید می‌کند. کد شما تابع هش را به درستی پیاده‌سازی کرده و خروجی آن بر اساس این تابع است. (مثال: برای timeout با size=5، مجموع ASCII برابر ۶۶۸ است که 668 % 5 = 3 می‌شود، پس باید در خانه ۳ قرار گیرد، نه خانه ۵ همانطور که در خروجی نمونه آمده است.)

۴. تحلیل پیچیدگی زمانی:

- درج (Insert):
- بهترین حالت: اگر هیچ برخوردی رخ ندهد و Rehash لازم نباشد، پیچیدگی O(1) است.
- بدترین حالت: اگر Rehash لازم شود، تمام n عنصر باید مجدداً هش شوند که پیچیدگی O(n) خواهد داشت. اما با توجه به اینکه Rehash به صورت دوره‌ای (هر بار با دو برابر شدن اندازه) اتفاق می‌افتد، پیچیدگی درج میانگین (Amortized Time Complexity) همچنان O(1)باقی می‌ماند.
- تابع هش (hash_): پیچیدگی تابع هش به طول رشته‌ی کلید بستگی دارد. اگر طول متوسط کلیدها L باشد، پیچیدگی O(L) است.
- مدیریت برخورد (Separate Chaining): در بدترین حالت که تمام کلیدها در یک سطل قرار گیرند (که با Rehash مناسب و تابع هش خوب، احتمال آن کم است)، عملیات جستجو یا درج

ممکن است به O(n) برسد. اما در حالت متوسط، طول زنجیره‌ها کوتاه نگه داشته شده و عملکرد نزدیک به O(1) حفظ می‌شود.

۵. نکات مفید:

- انتخاب تابع هش: تابع هش استفاده شده در این مثال (مجموع ASCII و باقی‌مانده) ساده است اما برای داده‌های واقعی و برای توزیع بهتر و کاهش برخورد، ممکن است نیاز به توابع پیچیده‌تر باشد.
- ظرفیت (Load Factor) Rehash: نسبت تعداد عناصر به ظرفیت جدول (count / size) معیار مهمی برای تعیین زمان Rehash است. در این کد، Rehash زمانی اتفاق می‌افتد که count برابر size شود (Load Factor = 1). در عمل، ممکن است Rehash در Load Factor پایین‌تر (مثلاً 0.7) انجام شود تا از طولانی شدن زنجیره‌ها جلوگیری شود و عملکرد O(1) بهتر حفظ گردد.
- مقیاس‌پذیری: استفاده از Rehash تضمین می‌کند که با افزایش حجم داده‌ها، کارایی جدول هش افت قابل توجهی نکند و این ساختار برای کاربردهای عملی بسیار مناسب است.