جزوه جلسه چهاردهم داده ساختارها و الگوریتم

۱۸ آبان ۱۴۰۰

فهرست مطالب

۱ درهم سازی یا Hash

جدول درهم سازی یکی از پرکاربردترین داده ساختار ها در علوم و مهندسی کامپیوتر است. این داده ساختار در پایتون به نام Dictionary و در جاوا به نام Hash Map شناخته میشود. به دلیل پرکاربرد بودن جدول درهم سازی، خود مسئله درهم سازی نیز پرکاربرد و مهم است.

۱.۱ کاربرد های درهم سازی

۱. زبان های برنامه نویسی: هم در نوشتن کامپایلر (برای مثال keyword های یک زبان را نمیتوان به عنوان اسم متغیر انتخاب کرد و این محدودیت توسط جدول درهم سازی اعمال میشود) و هم به عنوان داده ساختار آماده در خود زبان

۲. پایگاه های داده یا DataBase: در دیتابیس های جدید مانند SQL که داده ها به صورت جدولی ذخیره سازی میشوند، با در دست داشتن یک کلید میتوان به داده های متناطر با شئ آن کلید دسترسی پیدا کرد

۳. مسیریابی شبکه توسط router: مسیر یابی برای یک IP خاص توسط الگوریتم های درهم سازی انجام میگیرد

۴. سرورهای شبکه: عملیات اختصاص port برای یک برنامه توسط جدول درهم سازی انجام میگیرد

۵. حافظه مجازی: هنگام پر شدن حافظه RAM، قسمت هایی از آن رو Hard Drive نوشته میشوند که این عملیات به کمک درهم سازی انجام میگیرد

۶. جست و جوی رشته و زیر رشته در سرویس هایی مانند Google و Grep

۷. پیدا کردن تشابه DNA

۸. همگام سازی فایل ها و شاخه ها (Branch): در فضاهای ابری مانند Dropbox و دستورهایی مانند rsync در Linux، تغییر یک فایل رو یک کامپیوتر موجب تغییر فایل در دیگر کامپیوتر های همگام میشود

۹. رمزنگاری یا Cryptography

۱۰. زنجیره بلوکی یا Block Chain

۲.۱ توضیحات مسئله درهم سازی یا interface

دنبال داده ساختاری هستیم که بر روی تعدادی شئ که هرکدام دارای یک کلید منحصر به فرد هستند عملیات زیر را انجام دهد.
۱. insert: یک کلید میگیرد و آنرا در داده ساختار درج میکند. به دلیل منحصر بودن کلیدها، درصورت وجود یک شئ با کلید مدنظر، مقدار آن در داده ساختار آپدیت میشود؛ یعنی مقدار جدید جایگزین مقدار قبلی میشود.
۲. delete: یک کلیذ میگیرد و درصورت وجود آن در داده ساختار، آنرا حذف میکند.
۳. search: کلید را به عنوان ورودی میگیرد و در صورت وجود کلید در داده ساختار، شئ متناظر با آنرا برمیگرداند.
از دانسته های قبلی میدانیم بهترین داده ساختار برای پیاده سازی این عملیات درخت های دودویی جست و جوی متوازن (مانند AVL یا R-B) است که هر کدام از عملیات بالا را در زمان $O(logn)$ انجام میدهند. اما میتوان این زمان را به $O(1)$ در حالت میانگین کاهش داد.

۳.۱ چند نکته درمورد درهم سازی و Dictionary پایتون

درهم سازی در مدل مقایسه بیان نمیشود، زیرا در اینصورت مرتبه زمانی کمتر از $logn$ نمیشد.
همچنین در واسط درهم سازی نیازی به find _next()/prev() نداریم. البته در مدل word RAM دو عملیات ذکر شده را میتوان در زمانی کمتر از $O(1)$ پیاده سازی کرد.
جدول درهم سازی در پایتون Dictionary نام دارد که عملیات زیر روی آن اجرا میشود:

d = {'key1': 5, 'key2': 10}
d['key1'] → 5 , d[5] → KeyError
'key2' in d → True , 10 in d → False
d.item() → returns all dictionary (key and value)

۴.۱ پیاده سازی جدول درهم سازی به روش Direct Access Array

در این شیوه پیاده سازی، از کلید ها به عنوان اندیس های آرایه برای ذخیره سازی اشیا استفاده میشود. در این شیوه پیاده سازی، عملیات سه گانه ذکر شده (insert، delete و search) هرسه در زمان $O(1)$ انجام میشوند.
اما از محدودیت های این روش میتوان به:
۱. حافظه زیاد: هنگامی که کلیدها بزرگ شوند، اندازه آرایه نیز بزرگتر میشود
۲. cast کردن کلیدها به عدد: از انجا که کلید ها به عنوان اندیس نیز استفاده میشوند

باید مقادیر صحیح و نامنفی داشته باشند.
راه حل مشکل اول Hash و راه حل مشکل دوم Pre Hahs نام دارد.

۱.۴.۱ پیش درهم سازی یا **Pre Hash**

از آنجا که تمام اطلاعات در کامپیوتر به صورت صفر و یک نگه داری میشوند، پس میتوان برای هر داده یک نمایش عددی نیز داشت. تابع Pre Hash در زبان های مختلف پیاده سازی شده است (تابع hash در پایتون) که ورودی آن یک کلید و خروجی آن یک عدد صحیح و نامنفی یکتا است. همچنین کلید ها نیز باید غیر قابل تغییر (immutable) باشند.

۲.۴.۱ درهم سازی یا **Hash**

ایده حل مشکل ذکر شده، تقلیل مجموعه کلید هاست. اگر مجموعه کلید های اولیه را u بنامیم، میخواهیم u را به یک مجموعه کوچکتر با m عضو کاهش دهیم.
h را تابع درهم سازی مینامیم و داریم:

$h : u \to \{0, 1, 2, ..., m-2, m-1\}$

در حالت ایده آل داریم:

$m \approx n(= size(u))$

پس با حل دو مشکل ذکر شده، در پیاده سازی به روش Direct Access Array مراحل زیر را طی میکنیم: (x یک کلید از u میباشد)

$x \xrightarrow{PreHash} x' \xrightarrow{Hash} h(x') \to x \text{ is in index } h(x') \text{ of array}$

حال حالتی را در نظر میگیریم که دو کلید x' و y' بعد از اعمال تابع h، خروجی یکسان داشته باشند ($h(x') = h(y')$) یعنی هردوی کلیدها در یک خانه از آرایه دخیره بشوند. این حالت برخورد یا Collision نام دارد.
ساده ترین راه حل برای این مشکل، استفاده از زنجیر (chain یا linked list) است؛ یعنی هر خانه از آرایه یک زنجیر است که اعضای آن، عناصر با $h(x)$ های یکسان هستند.
درنهایت یا حل این مشکل مرتبه زمانی عملیات مربوط را ارائه میکنیم:

1. insert: $O(1)$

(در صورتی که نیاز به بررسی عدم وجود عنصر با کلیذ تکراری داشته باشیم مرتبه زمانی به $O(n)$ میرسد.)

2. delete: $O(n)$
3. search: $O(n)$

در بدترین حالت، همه اعضا در یک زنجیر ذخیره میشوند و مرتبه زمانی حذف و جست و جو به $O(n)$ میرسد.