دانشکده‌ی علوم ریاضی

ساختمان داده‌ها — ۱۵ آذر ۱۳۹۳

جلسه‌ی ۱۷: درهم‌سازی

مدرّس: دکتر شهرام خزائی — *نگارنده: مجتبی تفاق*

۱ مقدمه

در بسیاری از کاربردها، برای داده‌ساختار مورد نیاز کافی است تا کارکردهای یک فرهنگ لغت عادی را انجام دهد:

- اضافه کردن یک داده‌ی جدید،
- حذف یک داده‌ی موجود،
- جستجوی یک داده.

تا کنون داده‌ساختارهای زیر را جهت ذخیره اطلاعات دیده‌ایم:

داده‌ساختار	درج	جست‌وجو	حذف
آرایه‌ی نامرتب	۱	n	n
آرایه‌ی مرتب	n	$\log(n)$	n
لیست پیوندی یک‌سویه	۱	n	n
لیست پیوندی دوسویه	۱	n	۱
هرم بیشینه (کمینه)	$\log(n)$	$\log(n)$	$\log(n)$

در این جلسه می‌خواهیم داده‌ساختاری را معرفی کنیم، که این اعمال را در زمان $\Theta(1)$ انجام دهد.

۲ آدرس‌دهی مستقیم

فرض کنید می‌خواهیم اطلاعات ۱۰۰۰۰ دانشجو را ذخیره کنیم. همچنین فرض کنید شماره‌های دانشجویی، اعدادی بین ۱ تا یک میلیون هستند. برای این کار می‌توانیم از یک ”جدول آدرس‌دهی مستقیم“[۱] استفاده کنیم، به این ترتیب که آرایه‌ای مانند T با یک میلیون خانه و برای هر شماره‌ی دانشجویی یک خانه از T را در نظر می‌گیریم. بنابراین یک آرایه‌ی خیلی بزرگ خواهیم داشت، که در آن از شماره‌ی دانشجویی به عنوان اندیس آرایه استفاده کرده‌ایم و در هر

[۱] direct addressing table

خانه‌ی متناظر با شماره دانشجویی یک دانشجو، اطلاعات و یا اشارگری به اطلاعات آن دانشجو وجود دارد.
این روش وقتی که کلیدها (در این‌جا شماره‌ی دانشجویی) از یک مجموعه‌ی به طور نسبی کوچک آماده باشند، روشی کارآمد است و هدف ما را برآورده می‌کند زیرا که سه عمل بالا را با الگوریتم‌هایی که در پایین می آیند می‌توان در زمان ثابت انجام داد:

DIRECT-ADDRESS-SEARCH (T, k)
$Return\ T[k]$

DIRECT-ADDRESS-INSERT (T, x)
$T[x.key] \longleftarrow x$

DIRECT-ADDRESS-DELETE (T, x)
$T[x.key] \longleftarrow null$

---

۳ چند کاربرد

حال فرض کنید که به جای اطلاعات ۱۰۰۰۰ دانشجو، می‌خواستیم که اطلاعات یک کلاس ۱۰۰ نفره را ذخیره کنیم. دوباره مجبور بودیم آرایه‌ای با همان اندازه‌ی قبلی در نظر بگیریم. همان‌گونه که مشاهده می‌شود، در مواقعی که تعداد داده‌های مورد نظر نسبت به اندازه‌ی مجموعه‌ای که کلیدها از آن می‌آیند کوچک باشد، روش آدرس‌دهی مستقیم کارایی خود را به علت اتلاف حافظه از دست خواهد داد. دقت کنید که کاربردهای این چنینی بسیار زیاد رخ می‌دهند.
به عنوان یک مثال دیگر فرض کنید می‌خواهیم خود یک فرهنگ لغت را پیاده‌سازی کنیم. با احتساب فرم‌های مختلف کلی حدود ۶۰۰۰۰۰ لغت در زبان انگلیسی وجود دارد، اما اگر بخواهیم رشته‌های حروف به طول حداکثر سی را به عنوان کلید برای آدرس‌دهی مستقیم در نظر بگیریم، آن‌گاه به آرایه‌ای به اندازه‌ی $۲۶^{۳۰}$ نیاز داریم!
برای مثال آخر، رهیابی[۲] را در یک شبکه در نظر بگیرید که به طور متوسط در هر لحظه باید ۵۰۰ آدرس را ذخیره کند. تعداد IPهای ممکن برابر $۲^{۳۲}$ است که همان اندازه‌ی مجموعه‌ی کلیدهای ماست. آیا راه‌حلی برای استفاده‌ی بهتر از حافظه در این موارد وجود دارد؟

[۲] router

۴ درهم‌سازی

در این بخش، با داده‌ساختاری به نام جدول درهم‌سازی [۳] و روش درهم‌سازی آشنا می‌شویم. در جدول‌های درهم‌سازی سه عمل مورد نظر در بالا بسته به شرایط مختلف، زمان‌های متفاوتی می‌برند ولی با انتخاب مناسب روش، می‌توان میانگین این زمان را در حد $\Theta(۱)$ پایین آورد.
ایده‌ی کلی بر این اساس است که به جای در نظر گرفتن خود کلیدها، برای آدرس‌دهی از تابعی از کلیدها استفاده و هر داده را در اندیس متناظر با مقدار آن تابع ذخیره کنیم. حال اگر برد این تابع از مرتبه‌ی تعداد داده‌ها باشد و این تابع روی کلیدهای مورد نظر به صورت "تقریبا" یک‌به‌یک عمل کند، تمامی اطلاعات را می‌توان در آرایه‌ای به اندازه‌ی داده‌ها ذخیره کرد که مطلوب‌ترین حالت ممکن است.
فرض می‌کنیم مجموعه‌ی داده‌ها حداکثر شامل m عنصر است. کلید عنصر x را $x.k$ می‌نامیم که کلیدها متمایز و عناصری از مجموعه‌ای به‌نام K هستند. همچنین می‌دانیم که $K \subseteq U$ در آن U مجموعه‌ی جهانی مقادیر ممکن کلیدهاست. همان‌طور که اشاره شد، تابع درهم‌ساز از فضای کلیدهای ممکن به فضای اندیس‌های موجود است. یعنی:

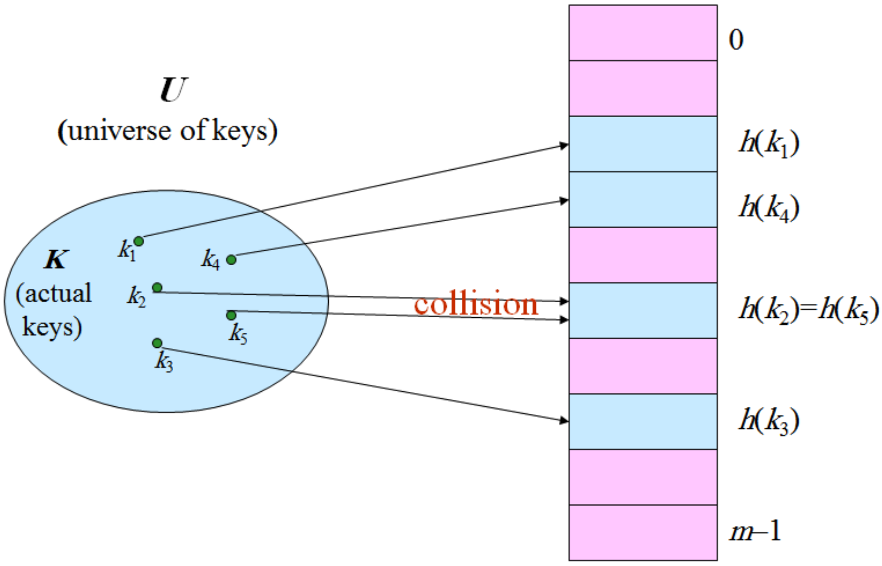
$$h : U \longrightarrow \{۰, ۱, \cdots, m - ۱\}$$

اگر تابع h روی مجموعه‌ی K یک‌به‌یک باشد آن‌گاه داده‌ی x را در $h(x.k)$ ذخیره می‌کنیم. اما اگر دو کلید متفاوت به یک اندیس مشابه نگاشته شوند، چه باید کرد؟ به این مشکل برخورد [۴] می‌گوییم که باید آن‌را به روشی برطرف کنیم.

۵ برخورد

شکل ۱ یك جدول درهم‌سازی با مجموعه‌ی جهانی U، مجموعه‌ی کلیدهای K و نیز تابع h را برای پنج کلید نشان می‌دهد. در این‌جا کلیدهای $k_۵$ و $k_۲$ برخورد دارند چون تحت h به یک درایه نگاشته می‌شوند. برای رفع مشکل برخورد به دو راه‌حل اشاره می‌کنیم. روش اول "روش زنجیره‌ای" [۵] و روش دوم "آدرس دهی باز" [۶] نامیده می‌شود. در ادامه به توصیف این دو روش خواهیم پرداخت.

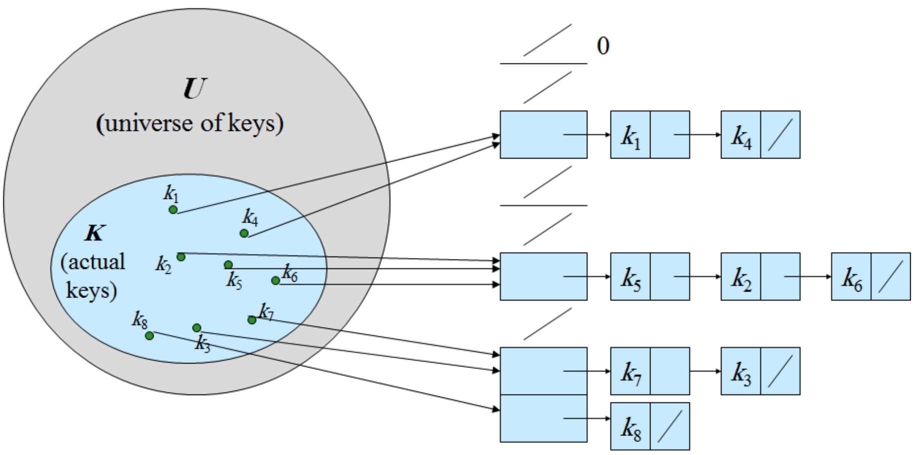
[۳] hashing table
[۴] collosion
[۵] chaind hashing
[۶] open addressing



شکل ۱: یک جدول درهم‌سازی با تابع درهم‌سازی h.

۶ روش ذنجیره‌ای

یکی از راه‌حل‌های رفع مشکل برخورد، ذخیره‌ی اندیس‌هایی که برخورد کرده‌اند در یک لیست پیوندی است. درایه‌های خالی جدول هم لیست‌های تهی ($null$) هستند. به چنین جدولی "جدول درهم‌سازی با روش زنجیره‌ای" می‌گوییم. شکل ۲ مثالی از این روش است.



شکل ۲: روش زنجیره‌ای برای حل برخورد.

در زیر الگوریتم‌های متناظر با سه عمل مورد نظر در این روش آمده است:

Chained-Hash-Insert (T, x)
insert x at the head of the list $T[h(x.k)]$

Chained-Hash-Search (T, k)
search for an element with key k in list $T[h(k)]$

Chained-Hash-Delete (T, x)
delete x from list $T[h(x.k)]$

اگر تابع درهم‌ساز تقریبا یک‌به‌یک باشد، تعداد برخوردها کم خواهد بود و هر سه عمل درج، حذف، و جست‌وجو در $\Theta(1)$ انجام خواهند شد. در غیر این صورت، بعضی از کلیدها یک لیست طولانی خواهند داشت. با این‌که زمان درج $\Theta(1)$ است (چون می‌توانیم به ابتدای لیست اضافه کنیم)، ولی حذف و جستجو کند خواهند بود و در بدترین حالت ممکن است در $\Theta(n)$ انجام شوند که در این‌جا n طول لیست پیوندی مورد نظر است.

۷ عامل بار

میزان برخورد مستقیما به تعداد اعداد ورودی بستگی ندارد و به "عامل بار"[۷] جدول مربوط است. عامل بار، که با نماد α نشان می‌دهیم، برابر است با نسبت تعداد داده‌های ورودی (تعداد کلیدها) به اندازه‌ی آرایه:

$$\alpha = \frac{n}{m}$$

عامل بار را می‌توان به این صورت تفسیر کرد که در m خانه‌ی حافظه‌ی با n داده‌ی ورودی به طور متوسط α داده در هر خانه قرار می‌گیرد.
برای به دست آوردن زمان متوسط سه عمل بالا فرض می‌کنیم تابع h برای قرار دادن هر کلید در حافظه، یکی از m خانه‌ی موجود را به صورت تصادفی انتخاب می‌کند، یعنی احتمال انتخاب شدن هر خانه برابر با $\frac{1}{m}$ است. در ضمن فرض می‌کنیم کلیدها مستقل از هم در خانه‌های جدول ذخیره می‌شوند. به این فرض "درهم‌سازی یک‌نواخت ساده"[۸] می‌گوییم، زیرا هر کلید شانس یکسانی برای ذخیره شدن در هر خانه‌ی جدول دارد. توجه کنید که این فرض صحیحی نیست، چون h یک تابع غیرتصادفی است و این فرض صرفا جهت ساده‌سازی انجام می‌شود.
حال برای مثال میانگین زمان جست‌وجو را محاسبه می‌کنیم. بدترین حالت زمانی اتفاق می‌افتد که کلید k که به دنبال آن هستیم ذخیره نشده باشد. در این صورت باید کل لیست $h(k)$ را پیمایش کنیم. بنابر تعریف، طول این لیست به‌طور متوسط برابر است با عامل بار:

$$E[n_{h(k)}] = \alpha$$

پس زمان جست‌وجو در حالت ناموفق از مرتبه‌ی $\Theta(1+\alpha)$ است. زمان ۱ برای این است که ببینیم کلید در کدام خانه‌ی

---

[۷] load factor
[۸] simple uniform hashing

حافظه است و در زمان α باید آرایه‌ی مربوط به آن خانه را جست‌وجو کنیم. در حالت جست‌وجوی موفق نیز با همین استدلال و با توجه به این که کلید k به طور متوسط در فاصله‌ی $\frac{\alpha}{۲}$ از ابتدای لیست قرار دارد، زمان جست‌وجوی از مرتبه‌ی $\Theta(۱ + \frac{\alpha}{۲})$ است.
در مورد زمان درج و حذف اگر از لیست پیوندی دوسویه استفاده کنیم، هر دو عمل در $\Theta(۱)$ انجام می‌شوند. بنابراین در مجموع اگر $\alpha = O(۱)$ آن‌گاه همان‌طور که انتظار داشتیم هر سه عمل مورد نظر در $\Theta(۱)$ انجام خواهند شد.

۸ تابع درهم‌ساز

در این قسمت می‌خواهیم چند نوع تابع درهم‌ساز مقدماتی را معرفی کنیم. فرض کنید می‌خواهیم برای مثال رهیاب در بالا، یک تابع درهم‌ساز بسازیم. یک پیشنهاد می‌تواند به صورت زیر باشد:

$$h(k) = k \mod ۲۵۶$$

دقت کنید که عمل این تابع روی IPهای مختلف را می‌توان به این صورت نمایش داد:

$$(x_۱, x_۲, x_۳, x_۴) \rightarrow x_۴$$

آیا این تابع مناسب است؟ برای پاسخ دادن به این سوال باید فرض درهم‌سازی یک‌نواخت ساده را بررسی کنیم. در این حالت اگر بدانیم که هشت رقم آخر یک آدرس در مبنای دو توزیع یکنواخت دارند، این تابع در این فرض صدق می‌کند. البته در عمل موضوعات دیگری را نیز باید در نظر گرفت، مانند این که طراحی یک حمله برای چنین تابع ساده‌ای بسیار راحت است که در این‌جا به این مباحث نمی‌پردازیم.
برای کاربردهای ساده، یک تابع مناسب در بسیاری از این موارد تابع زیر است:

$$h(k) = \lfloor m \; \{A.k\} \rfloor$$

در این عبارت m مانند گذشته اندازه‌ی آرایه است. تابع $\{\cdot\}$ همان تابع جز اعشار است، یعنی $\{x\} = x - \lfloor x \rfloor$. برای مقدار ثابت A، انتخاب‌های زیادی می‌توان داشت ولی آن‌چه در عمل بسیار استفاده می‌شود،

$$A = \frac{\sqrt{۵} - ۱}{۲} = ۰/۶۱۸۰۳۳۹۸۸۷\ldots$$

می‌باشد.

۹ آدرس‌دهی باز

همان‌گونه که گفته شد برای رفع مشکل برخورد، روش دیگری نیز وجود دارد که همان آدرس‌دهی باز است. در این روش برعکس روش قبل تمامی داده‌ها داخل خود آرایه ذخیره می‌شوند. شاید به نظر آید که می‌توان همان لیست‌های پیوندی را درون خود آرایه ذخیره کرد ولی نکته‌ی بعدی آن است که در این روش به اشاره‌گر نیز نیازی نیست، که این خود باعث صرفه‌جویی در حافظه است.

ایده‌ی کلی آن است که اگر برخورد رخ داد، به خانه‌ی بعدی برویم و همین روند را ادامه دهیم تا در نهایت به یک خانه‌ی خالی برسیم و یا این‌که کل آرایه پر شده باشد. ایده‌ی بعدی آن است که این "وارسی"[۹] برای رسیدن به خانه‌ی خالی، به این صورت نیست که در هر مرحله به خانه‌ی بعدی نگاه کنیم و در عوض "دنباله‌ی وارسی"[۱۰] تابعی از کلید مورد نظر است.
برای این منظور باید به ازای هر کلید یک جایگشت از $\{۰, ۱, ۲, \cdots, m - ۱\}$ را به عنوان دنباله‌ی وارسی متناظر داشته باشیم، پس تابع درهم‌ساز را به این طریق گسترش می‌دهیم:

$$h : U \times \{۰, ۱, \cdots, m - ۱\} \longrightarrow \{۰, ۱, \cdots, m - ۱\}$$

با در دست داشتن این تابع، دنباله‌ی وارسی متناظر با هر کلید به این صورت به‌دست می‌آید که:

$$k \to \langle h(k, ۰), h(k, ۱), \cdots, h(k, m - ۱)\rangle$$

حال برای درج داده با کلید k کافی است با همین ترتیب جدول را وارسی و داده را در اولین خانه‌ی خالی ذخیره کنیم. به طریق مشابه، برای جست‌وجوی داده با کلید k دوباره با همین ترتیب خانه‌ها را وارسی می‌کنیم. اگر کلید k را یافتیم که جست‌وجو موفق بوده است و اگر به خانه‌ای خالی رسیدیم، دیگر نیازی به ادامه‌ی وارسی نیست و جست‌وجو ناموفق بوده است. در زیر شبه‌کدهای متناظر با آن‌چه گفته شد، آورده شده است.

```
Hash-Insert (T, k)
i ← 0
repeat
    j ← h(k, i)
    if T[j] = null
        T[j] ← k
        return j
    else i ← i + 1
until i = m
error "hash table overflow"
```

۱۰ الگوریتم حذف

در روش آدرس‌دهی باز الگوریتم حذف نیاز به توجه ویژه‌ای دارد. آن‌چه در ابتدا به ذهن می‌رسد این است که همان الگوریتم جست‌وجو را اجرا و پس از یافتن کلید مورد نظر، آن را حذف کنیم. اما توجه کنید اگر خانه‌ی پاک شده را با *null* علامت‌گذاری کنید، در الگوریتم جست‌وجو و در حالت جست‌وجوی ناموفق دچار مشکل خواهید شد. بنابراین باید از نماد دیگری مانند *deleted* استفاده کرد که در الگوریتم درج دقیقا مانند *null* عمل می‌کند، ولی در الگوریتم

[۹] probe
[۱۰] probe sequence

```
Hash-Search (T, k)
i ← 0
repeat
    j ← h(k, i)
    if T[j] = k
        return j
    i ← i + 1
until T[j] = null or i = m
return null
```

جست‌وجو با رسیدن به آن جست‌وجو متوقف نمی‌شود.
با اندکی دقت، باز مشکل دیگری وجود دارد و آن این که در حالتی که تعداد زیادی داده اضافه و پاک شوند، آن بخش از دنباله‌های وارسی که در الگوریتم جست‌وجو دنبال می‌شود به علت وجود تعداد زیادی $deleted$ مستقل از عامل بار بلند خواهد شد و بنابراین اجرای الگوریتم جست‌وجو به صورت قابل ملاحظه‌ای کند می‌شود. پس با این که دیدیم روش آدرس‌دهی باز نسبت به استفاده از لیست‌های پیوندی مدیریت حافظه‌ی کارآمدتری دارد، در مواردی که داده‌ساختار به صورت مداوم تغییر می‌کند مناسب نیست.

۱۱ درهم‌سازی یک‌نواخت

برای تحلیل کارایی روش آدرس‌دهی باز به تعمیمی از فرض درهم‌سازی یک‌نواخت ساده نیاز داریم، که آن را فرض "درهم‌سازی یک‌نواخت"[11] می‌نامیم. این فرض دقیقا مانند فرض قبلی است، با این تفاوت که در این حالت تابع درهم‌ساز به ازای هر کلید یک جایگشت تصادفی تولید می‌کند، پس فرض درهم‌سازی یکنواخت به این صورت بیان می‌شود که دنباله‌ی وارسی متناظر با یک کلید مشخص، می‌تواند با احتمال برابری هر یک از $m!$ جایگشت ممکن باشد.
با در نظر گرفتن این فرض، قضیه زیر در مورد زمان اجرای الگوریتم جست‌وجو برقرار است که اثبات آن در کتاب منبع درس آمده است و در این‌جا به آن اشاره نمی‌شود.

قضیه ۱ *برای یک جدول درهم‌سازی با آدرس‌دهی باز که در آن ضریب بار* $\alpha = \frac{n}{m} < ۱$*، امید ریاضی تعداد پیمایش‌ها در یک جستجوی ناموفق حداکثر برابر است با:*

$$\frac{۱}{۱ - \alpha}$$

همچنین با فرض این که هر کلید در جدول با احتمال یکسانی جستجو می‌شود، امید ریاضی تعداد پیمایش‌ها در یک جستجوی موفق حداکثر برابر است با:

$$\frac{۱}{\alpha} \ln(\frac{۱}{۱ - \alpha})$$

[11] uniform hashing

۱۲ ساخت تابع درهم‌ساز از روی تابع درهم‌ساز ساده

در این قسمت می‌خواهیم به طور اجمالی به روش ساخت یک تابع درهم‌ساز برای آدرس‌دهی باز مانند

$$h : U \times \{۰, ۱, \cdots, m - ۱\} \longrightarrow \{۰, ۱, \cdots, m - ۱\}$$

از روی یک تابع درهم‌ساز کمکی مانند

$$h' : U \longrightarrow \{۰, ۱, \cdots, m - ۱\}$$

بپردازیم. به عنوان تلاش اول، تابع زیر را در نظر بگیرید:

$$h(k, i) = h'(k) + i \mod m$$

این تابع در حقیقت همان ساده‌ترین روش موجود است که هر موقع خانه‌ای پر بود، به خانه‌ی بعدی آن برویم.
مشکلی که برای این روش وجود دارد، زمان بالای الگوریتم جست‌وجو است. فرض کنید که i خانه‌ی متوالی پر باشند و خانه‌ی بعدی آن‌ها خالی باشد. آن‌گاه این خانه‌ی خالی در درج بعدی با احتمال $(i + ۱)/m$ پر می‌شود. به این ترتیب دنباله‌های طولانی از خانه‌های اشغال شده بلندتر و بلندتر می‌شوند و زمان جست‌وجو را کندتر و کندتر می‌کنند.
به جای آن روشی را در نظر بگیرید که در هر مرحله به جای خانه‌ی بعد، به چند خانه بعد می‌پریم و طول این پرش برای هر کلید متفاوت و عددی تصادفی است. به این منظور می‌توان از یک تابع درهم‌ساز کمکی دیگری مانند h''، به صورت زیر استفاده نمود:

$$h(k, i) = h'(k) + i \times h''(k) \mod m$$

به سادگی دیده می‌شود که مشکل تجمع خانه‌های اشغال شده که در بالا به آن اشاره شد، در این صورت اصلاح شده به وجود نمی‌آید.