

به نام خدا

ساختمان داده

آرش شفيعی



درهم سازی

- الگوریتم‌ها ممکن است نیاز داشته باشند عملیات متعددی بر روی مجموعه‌های پویا انجام دهند. بسیاری از الگوریتم‌ها نیاز دارند تنها عملیات درج، حذف و جستجو انجام دهند. مجموعه‌ای پویا که این عملیات را پشتیبانی کند دیکشنری¹ نامیده می‌شود.
- برای مثال در یک کامپایلر جدولی به نام جدول علائم وجود دارد که متغیرها و ویژگی‌های آنها را ذخیره می‌کند. برای ذخیره‌سازی متغیرهای یک برنامه و ویژگی‌های آن متغیرها از دیکشنری استفاده می‌کنیم.

¹ dictionary

- جدول درهم سازی¹ یک ساختار داده کارا برای پیاده سازی دیکشنری است.
- اگرچه پیچیدگی زمانی جستجوی یک عنصر در جدول درهم سازی می تواند در بدترین حالت مانند جستجو در یک لیست پیوندی یعنی $\Theta(n)$ باشد، اما با تعدادی پیش فرض می توان جستجوی یک عنصر در جدول درهم سازی را در زمان $O(1)$ انجام داد.
- نوع داده ای دیکشنری در زبان پایتون توسط جداول درهم سازی پیاده سازی شده است.
- درواقع جدول درهم سازی مفهوم آرایه ها را تعمیم می دهد تا جستجو در مجموعه پویا همانند دسترسی به یک عنصر در آرایه در زمان $O(1)$ انجام شود.
- ایده اصلی جدول درهم سازی این است که اندیس یک مکان در آرایه، که حاوی یک مقدار کلید است، از طریق مقدار کلید آن با استفاده از روشی که توضیح خواهیم داد، محاسبه می شود.

¹ hash table

- آدرس دهی مستقیم¹ یک روش ساده است که وقتی مجموعه مرجع کلیدها (U) کوچک باشد می تواند مورد استفاده قرار بگیرد.

¹ direct addressing

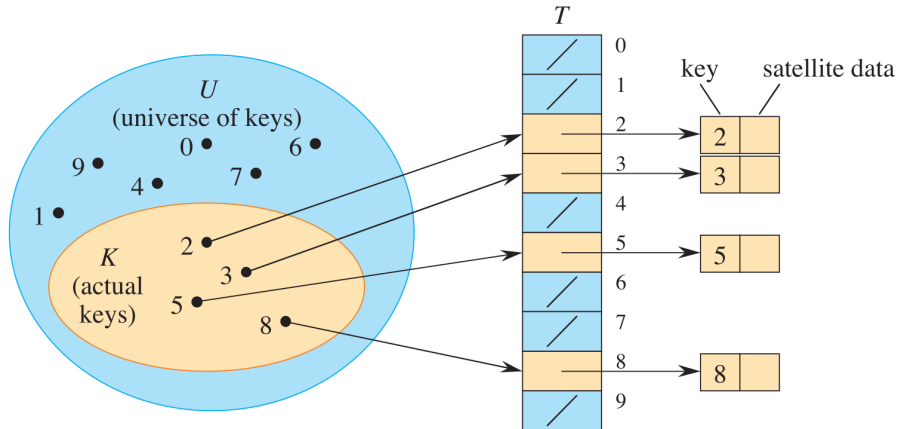
- فرض کنید برنامه‌ای می‌خواهد از یک مجموعهٔ پویا استفاده کند که کلیدهای عناصر آن یکتا بوده و زیر مجموعه‌ای از مجموعهٔ مرجع $U = \{0, 1, \dots, m-1\}$ ¹ هستند به طوری که m عدد بسیار بزرگی نیست.
- برای ذخیره‌سازی این مجموعه پویا، می‌توانیم از یک آرایه استفاده کنیم که به آن جدول آدرس دهی مستقیم گفته می‌شود. این آرایه را با $T[0 : m-1]$ نمایش می‌دهیم. هریک از مکان‌های آن (که به آن اسلات² نیز گفته گفته می‌شود) به یکی از کلیدها در مجموعهٔ مرجع U اختصاص دارد.

¹ universal set (universe)

² slot

جداول آدرس دهی مستقیم

- شکل زیر این روش را نمایش می دهد. مکان k به عنصری اشاره می کند که مقدار کلید آن k است. اگر مجموعه هیچ عنصری با کلید k نداشته باشد، آنگاه خواهیم داشت $T[k] = \text{NIL}$.



- همه توابع جستجو Direct-Address-Search و درج Direct-Address-Insert و حذف Direct-Address-Delete در زیر نشان داده شده اند که هر کدام در زمان $O(1)$ انجام می شوند.

Algorithm Direct Address Search

```
function DIRECT-ADDRESS-SEARCH(T,k)
1: return T[k]
```

Algorithm Direct Address Insert

```
function DIRECT-ADDRESS-INSERT(T,x)
1: T[x.key] = x
```

Algorithm Direct Address Delete

– **function** DIRECT-ADDRESS-DELETE(T, x)

1: $T[x.key] = \text{NIL}$

جداول درهم سازی

- اگر مجموعه مرجع U بسیار بزرگ یا نامحدود باشد، ذخیره سازی جدول T به اندازه $|U|$ غیر عملی یا حتی غیرممکن است.
- علاوه بر این، مجموعه k از کلیدهایی که واقعا ذخیره شده اند ممکن است نسبت به مجموعه مرجع U بسیار کوچک باشد و در نتیجه بیشتر فضایی که به T تخصیص داده شده است ممکن است هدر رود.
- وقتی مجموعه k یعنی کلیدهای ذخیره شده در دیکشنری بسیار کوچک تر از مجموعه مرجع U یعنی تمام کلیدهای ممکن باشد، جداول درهم سازی کمک می کنند که فضای بسیار کوچک تری نسبت به جدول آدرس دهی مستقیم اشغال شود.
- پیچیدگی حافظه با استفاده از جداول درهم سازی به $\Theta(|k|)$ کاهش پیدا می کند و پیچیدگی زمانی جستجو همچنان $O(1)$ باقی می ماند.

جداول درهم‌سازی

- در آدرس‌دهی مستقیم یک عنصر با کلید k در مکان k ذخیره می‌شود، اما در جداول درهم‌سازی از یک تابع درهم‌سازی h^1 استفاده می‌کنیم تا مکان کلید k را محاسبه کنیم و بنابراین کلید k در مکان $h(k)$ قرار می‌گیرد. تابع درهم‌سازی h مجموعه مرجع U را به مکان‌های جدول درهم‌سازی $T[0 : m - 1]$ نگاشت می‌کند.

$$h : U \rightarrow \{0, 1, \dots, m - 1\}$$

- در اینجا m اندازه جدول درهم‌سازی است که معمولاً بسیار کوچک‌تر از اندازه U است.

- می‌گوییم عنصر با کلید k به مکان $h(k)$ نگاشت می‌شود ².

- همچنین می‌گوییم $h(k)$ مقدار نگاشت شده یا مقدار هش ³ برای کلید k است.

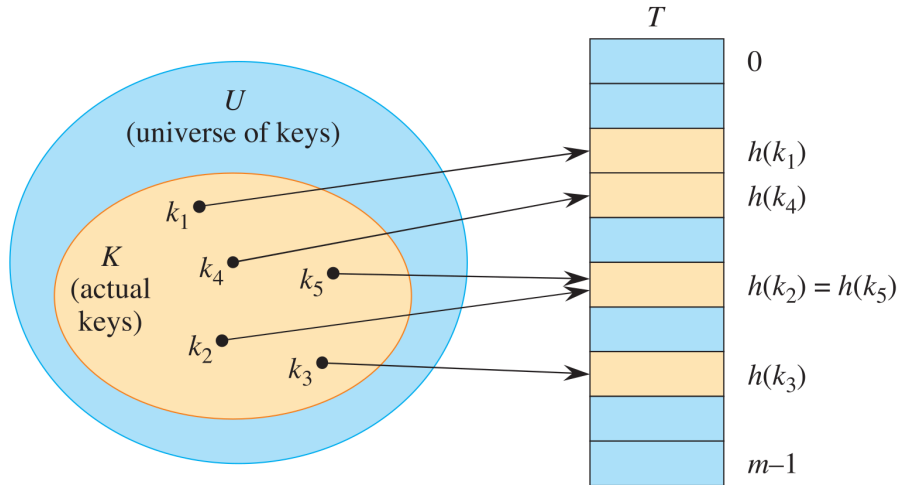
¹ hash function

² k hashes to slot $h(k)$

³ hash value

جداول درهم سازی

- ایده اصلی جدول درهم سازی در زیر نشان داده شده است.



- تابع درهم سازی محدوده اندیس های آرایه را کاهش می دهد و بنابراین اندازه آرایه مورد نیاز برای ذخیره کاهش پیدا می کند، در نتیجه به جای استفاده از جدولی به اندازه $|U|$ از جدولی به اندازه m استفاده می کنیم.
- یک مثال بسیار ساده از تابع درهم سازی تابع $h(k) = k \bmod m$ است.

- مشکلی که ممکن است در فرایند محاسبه تابع درهم‌سازی به وجود آید این است که دو کلید به یک مکان حافظه نگاشت شوند. به چنین وضعیتی برخورد یا تصادم¹ گفته می‌شود.
- روش‌های کارایی برای حل مشکل تصادم وجود دارد.
- در شرایط ایده‌آل هیچ تصادمی نباید رخ دهد و تلاش می‌کنیم تابع درهم‌سازی را به نحوی انتخاب کنیم که تصادم حداقل شود.

¹ collision

- تابع درهم‌سازی باید قطعی¹ باشد به طوری که به ازای ورودی k همیشه خروجی $h(k)$ را تولید کند.
- از آنجایی که $|U| > m$ بنابراین امکان تصادم وجود دارد. بنابراین اگرچه یک تابع درهم‌سازی خوب تعداد تصادم‌ها را کاهش می‌دهد، اما همچنان امکان تصادم وجود دارد و باید روشی برای برطرف کردن تصادم به کار ببریم.
- در ادامه در مورد توابع درهم‌سازی و همچنین روش‌های برخورد با تصادم صحبت خواهیم کرد.

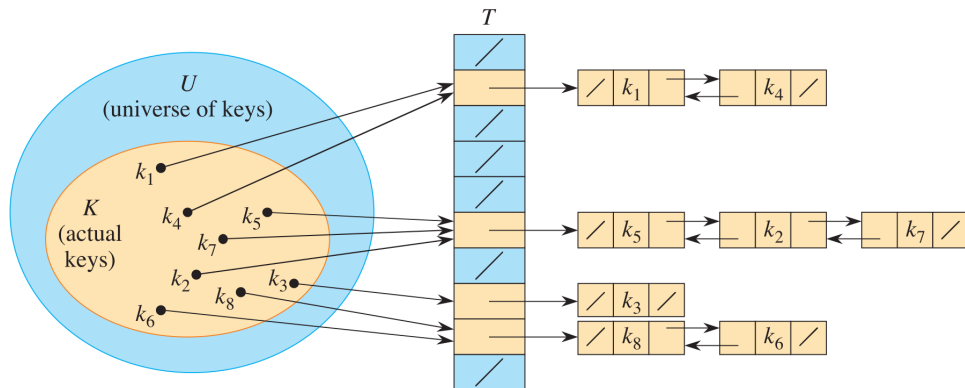
¹ deterministic

- درهم‌سازی مستقل یکنواخت¹ : یک تابع درهم‌سازی ایده‌آل h به ازای هر ورودی k در دامنه U خروجی $h(k)$ را مستقلاً به طور یکنواخت از محدوده $\{0, 1, \dots, m-1\}$ انتخاب می‌کند.
- مستقل بدین معناست که نگاشت یک کلید به کلیدهای دیگر بستگی ندارد و یکنواخت بدین معناست که یک کلید با احتمال یکسان می‌تواند به هر یک از خانه‌های جدول درهم‌سازی نگاشت شود.
- وقتی مقدار $h(k)$ انتخاب شد، هر فراخوانی بعدی h با ورودی k همان مقدار $h(k)$ را تولید می‌کند.
- تابعی که چنین ویژگی‌هایی داشته باشد را یک تابع درهم‌سازی مستقل یکنواخت² می‌نامیم.

¹ independent uniform hashing

² independent uniform hash function

- رفع تصادم با روش زنجیره‌ای : شکل زیر ایده اصلی روش زنجیره‌ای¹ برای رفع تصادم را نشان می‌دهد.



¹ chaining

- هر مکان غیرتهی در جدول به یک لیست پیوندی اشاره می‌کند و همه عناصری که به یک مکان نگاشت می‌شوند وارد لیست پیوندی آن مکان می‌شوند.
- مکان z یک اشاره‌گر به ابتدای یک لیست پیوندی دارد که همه عناصری که به مکان z نگاشت می‌شوند را نگهداری می‌کند. اگر هیچ عنصری به مکان z نگاشت نشده باشد، مکان z تهی است و مقدار NIL را نگهداری می‌کند.
- اگر از چنین روشی برای رفع تصادم استفاده کنیم عملیات دیکشنری به صورت زیر پیاده‌سازی خواهند شد.

Algorithm Chained Hash Insert

```
function CHAINED-HASH-INSERT(T,x)
1: List-Prepend (T[h(x.key)],x)
```

Algorithm Chained Hash Search

— **function** CHAINED-HASH-SEARCH(T, k)
1: **return** List-Search ($T[h(k)], k$)

Algorithm Chained Hash Delete

function CHAINED-HASH-DELETE(T, x)
1: List-Delete ($T[h(x.key)], x$)

- زمان اجرای عملیات درج در بدترین حالت $O(1)$ است. عملیات درج سریع است زیرا فرض می‌کنیم عنصر x که می‌خواهیم درج کنیم در جدول وجود ندارد. بدون وجود این فرض، قبل از درج باید جستجو کنیم که آیا $x.key$ در لیست پیوندی وجود دارد یا خیر.
- زمان اجرای عملیات جستجو در بدترین حالت متناسب با طول لیست است که این زمان اجرا را دقیق‌تر مورد بررسی قرار خواهیم داد.
- زمان اجرای عملیات حذف $O(1)$ است اگر لیست‌های پیوندی دو طرفه باشند. اگر لیست یک طرفه باشد، برای پیدا کردن عنصر قبلی عنصری که می‌خواهیم حذف کنیم، باید لیست را پیمایش کنیم. از آنجایی که تابع حذف یک اشاره‌گر x به عنصر مورد نظر دریافت می‌کند و نه مقدار کلید آن را، بنابراین نیازی به جستجو نیست.

- تحلیل درهم سازی با روش زنجیره ای : به ازای جدول درهم سازی T شامل m مکان که n عنصر را ذخیره می کند، ضریب بار α^1 برای T را به صورت n/m تعریف می کنیم که تعداد میانگین عناصر ذخیره شده در یک زنجیره است. ضریب بار α می تواند کمتر، مساوی یا بیشتر از یک باشد.
- در بدترین حالت همه n کلید در یک مکان جدول ذخیره می شوند که در واقع لیستی به طول n می سازند. زمان اجرا در بدترین حالت برای جستجو $\Theta(n)$ است به علاوه زمان مورد نیاز برای محاسبه تابع درهم سازی.
- کارایی درهم سازی در حالت میانگین بستگی به این دارد که تابع درهم سازی h چگونه مجموعه n کلید را در m مکان توزیع می کند.

¹ load factor

- فرض کنید به ازای $j = 0, 1, \dots, m-1$ طول لیست $T[j]$ برابر با n_j باشد. بنابراین خواهیم داشت $n = n_0 + n_1 + \dots + n_{m-1}$ و مقدار مورد انتظار¹ برای n_j برابر با $E[n_j] = \alpha = n/m$ است.
- فرض می کنیم زمان $O(1)$ برای محاسبه مقدار هش $h(k)$ کافی است و بنابراین زمان لازم برای جستجوی یک مقدار با کلید k به طول $n_{h(k)}$ از لیست $T[h(k)]$ بستگی دارد.

¹ expected value

جداول درهم سازی

- قضیه : در یک جدول درهم سازی که تصادم با زنجیرسازی رفع می شود، یک جستجوی ناموفق به طور متوسط در زمان $\Theta(1 + \alpha)$ انجام می شود.
- اثبات : با فرض بر این که درهم سازی مستقل و یکنواخت است، هر کلید k که هنوز در جدول ذخیره نشده است، می تواند با احتمال یکنواخت در هر یک از m مکان جدول ذخیره شود. زمان مورد نیاز برای یافتن کلیدی که در جدول وجود ندارد یعنی برای جستجوی ناموفق کلید k برابر با زمان مورد انتظار آن $E[n_{h(k)}] = \alpha$ است.
- بنابراین تعداد عناصر بررسی شده در یک جستجوی ناموفق برابر با α است و کل زمان مورد نیاز (شامل زمان محاسبه $h(k)$) برابر با $\Theta(1 + \alpha)$ است.

جداول درهم سازی

- بنابراین جستجو در جدول درهم سازی که در آن تصادم با زنجیر سازی رفع می شود، در بدترین حالت برابر با $\Theta(1 + \alpha)$ است.
- این تحلیل بدین معناست که اگر تعداد عناصر در جدول متناسب با تعداد مکان های جدول درهم سازی باشد، داریم $n = O(m)$ و $\alpha = n/m = O(m)/m = O(1)$.
- بنابراین جستجو به طور میانگین در زمان ثابت انجام می شود. از آنجایی که درج در بدترین حالت در زمان $O(1)$ و حذف در بدترین حالت در زمان $O(1)$ انجام می شود، وقتی لیست دو طرفه باشد، بنابراین همه عملیات دیکشنری به طور متوسط در زمان $O(1)$ انجام می شوند.
- تحلیل ارائه شده تنها وقتی معتبر است که درهم سازی یکنواخت و مستقل باشد. یکنواخت بودن بدین معناست که کلیدها با احتمال برابر در هریک از m مکان نگاشت می شود و مستقل بودن بدین معناست که هر دو کلید متمایز با احتمال $1/m$ برخورد می کنند و وابستگی بین کلیدها در محاسبه مقدار درهم سازی وجود ندارد.

- برای اینکه درهم‌سازی خوبی داشته باشیم نیاز به یک تابع درهم‌سازی خوب داریم.
- علاوه بر اینکه تابع باید به طور کارا قابل محاسبه باشد، بررسی می‌کنیم یک تابع خوب چه ویژگی‌هایی باید داشته باشد و چگونه طراحی می‌شود.
- در این قسمت ابتدا دو درهم‌سازی با روش تقسیم و روش ضرب را معرفی می‌کنیم.
- یک روش کارا این است که خانواده‌ای از توابع درهم‌سازی داشته باشیم و در زمان اجرا به طور تصادفی از یکی از توابع خانواده استفاده کنیم. به این روش درهم‌سازی تصادفی گفته می‌شود. یکی از انواع درهم‌سازی تصادفی، درهم‌سازی سراسری است که به معرفی آن خواهیم پرداخت.

- تابع درهم‌سازی خوب : یک تابع درهم‌سازی خوب بر اساس فرض درهم‌سازی یکنواخت مستقل عمل می‌کند بدین معنا که در آن هرکلید با احتمال یکسان به هرکدام از m مکان جدول نگاشت می‌شود مستقل از این که بقیه کلیدها به چه مکانی نگاشت شده‌اند.
- اگر بدانید کلیدها اعداد تصادفی حقیقی k هستند که به طور یکنواخت و مستقل در محدوده $0 \leq k < 1$ توزیع شده‌اند، آنگاه تابع درهم‌سازی $h(k) = [km]$ یک تابع یکنواخت و مستقل خواهد بود.
- اما معمولاً دسترسی به توزیع احتمالی کلیدهای ورودی نداریم.

- در عمل، یک تابع درهم‌سازی برای نوع داده‌های زیر طراحی می‌شود:

(۱) کلیدهای که اعداد صحیح غیر منفی کوچک هستند و در W بیت جای می‌گیرند. معمولاً مقدار W برابر با ۳۲ یا ۶۴ است.

(۲) کلیدهایی که یک وکتور کوچک از اعداد صحیح غیرمنفی هستند که هرکدام اندازه محدودی دارند. برای مثال، هر عنصر وکتور می‌تواند ۸ بیت باشد. یک وکتور را در این حالت یک رشته نیز می‌نامیم.

- درهم سازی ایستا¹ از یک تابع درهم سازی ثابت استفاده می کند.
- در اینجا در مورد دو روش استاندارد برای درهم سازی ایستا صحبت می کنیم که روش ضرب و تقسیم نام دارند.
- اگرچه درهم سازی ایستا در عمل استفاده نمی شود و از درهم سازی تصادفی استفاده می شود، اما درهم سازی ایستا پایه ای برای یادگیری مفاهیم درهم سازی است.

¹ static hashing

- روش تقسیم : در روش تقسیم¹ برای ساختن یک تابع درهم سازی کلید k را به یکی از m مکان با محاسبه باقیمانده k بر m نگاشت می کنیم. بنابراین تابع درهم سازی برابر است با $h(k) = k \bmod m$.
- برای مثال، اگر اندازه جدول درهم سازی $m = 12$ باشد و کلید مورد نظر $k = 100$ باشد، آنگاه $h(k) = 4$ است. از آنجایی که تنها عملیات مورد نیاز تقسیم است، روش تقسیم روش نسبتاً سریعی است.
- روش تقسیم وقتی m یک عدد اول باشد که به اعداد توان ۲ نزدیک نباشد خوب عمل می کند. اما هیچ تضمینی وجود ندارد که این روش در حالت میانگین کارایی خوبی داشته باشد.

¹ division method

- روش ضرب : روش ضرب برای ساخت تابع درهم سازی در دو گام عمل می کند. ابتدا، مقدار کلید k را در یک ثابت A در محدوده $0 < A < 1$ ضرب می کند و قسمت اعشاری kA را استخراج می کند. سپس مقدار به دست آمده را m ضرب می کند و کف آن را به عنوان نتیجه محاسبه می کند. بنابراین تابع درهم سازی برابر است با

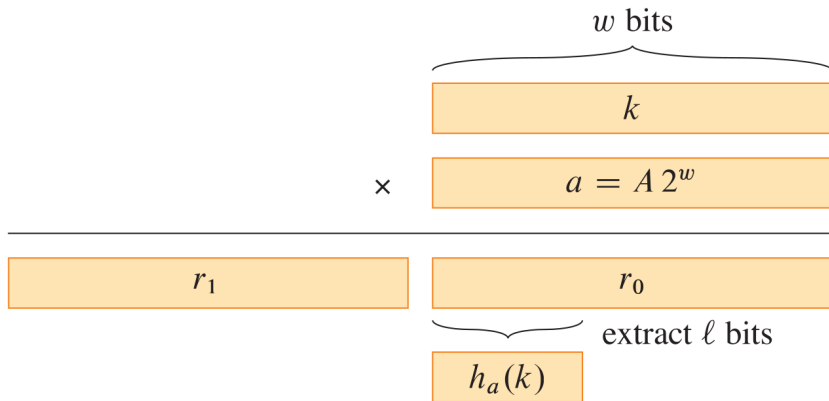
$$h(k) = \lfloor m(kA \bmod 1) \rfloor = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

- در اینجا $kA \bmod 1$ به معنای قسمت اعشاری kA است یعنی $kA - \lfloor kA \rfloor$.
- در عمل، روش ضرب وقتی m توانی از عدد ۲ است یعنی $m = 2^l$ (به ازای عدد صحیح l) به طوری که $l \leq w$ و w تعداد بیت های یک کلمه در ماشین است، خوب عمل می کند.

درهم سازی ایستا

- روش ضرب-انتقال : فرض می کنیم کلید k در یک فضای w بیتی جای می گیرد. عدد w بیتی $a = A2^w$ را انتخاب می کنیم، به طوری که $0 < A < 1$ و $0 < a < 2^w$.
- ابتدا k را در عدد صحیح w بیتی a ضرب می کنیم. نتیجه مقدار $2w$ بیتی $r_0 + r_1 2^w$ است جایی که r_1 مقدار w بیت مرتبه بالا و r_0 مقدار w بیت مرتبه پایین نتیجه ضرب است.
- مقدار l بیتی هشت تشکیل شده است از l بیت پر ارزش r_0 .
- از آنجایی که از r_1 چشم پوشی می شود، تابع درهم سازی می تواند بر روی یک کامپیوتر به گونه ای پیاده سازی شود که باقیمانده یک حاصل ضرب بر 2^w را محاسبه کند.
- تابع درهم سازی در اینجا به صورت $h_a(k) = (ka \bmod 2^w) \gg (w - l)$ محاسبه می شود.
- از آنجایی که حاصل ضرب ka از دو عدد w بیتی، میزان $2w$ فضا اشغال می کند، باقیمانده ضرب بر 2^w تعداد w بیت پر ارزش را حذف می کند (r_1) و w بیت کم ارزش را به دست می دهد (r_0).

- این روش در شکل زیر نشان داده شده است.



- عملگر \gg یک انتقال به چپ به اندازه $w - l$ بیت انجام می دهد، که معادل است با تقسیم عدد بر 2^{w-l} و محاسبه کف نتیجه.
- تابع درهم سازی h_a می تواند با سه دستور پیاده سازی شود، ضرب، تفریق و انتقال.
- به عنوان مثال، فرض کنید $k = 123456$ و $l = 14$ و $m = 2^{14} = 16384$ و $w = 32$.
- همچنین فرض کنید $a = 2654435769$ (که توسط دونالد کنوت پیشنهاد شده است). در این صورت $ka = 327706022297664 = (76300 \times 2^{32}) + 17612864$ و $r_1 = 76300$ و $r_0 = 17612864$.
- اگر ۱۴ بیت پر ارزش r_0 را استخراج کنیم خواهیم داشت $h_a(k) = 67$.

- اگرچه روش ضرب-انتقال روش سریعی است اما تضمین نمی کند که کارایی خوبی در حالت میانگین داشته باشد.
- اگر a به صورت تصادفی از بین اعداد فرد انتخاب شود، روش ضرب-انتقال در حالت میانگین خوب عمل می کند.

درهم‌سازی تصادفی

- در درهم‌سازی ایستا می‌توانیم کلیدها را به گونه‌ای انتخاب کنیم که همه آنها به یک مکان جدول نگاشت شوند. در این حالت پیچیدگی زمانی جستجو $\Theta(n)$ خواهد بود. هرگونه درهم‌سازی ایستا دارای این نقطه ضعف است.
- یک روش برای مقابله با این مشکل استفاده از درهم‌سازی تصادفی¹ است.
- یکی از حالات خاص این روش درهم‌سازی سراسری² است که کارایی خوبی دارد وقتی تصادم‌ها با زنجیرسازی مدیریت می‌شوند.
- برای استفاده از درهم‌سازی تصادفی، در ابتدای اجرای برنامه تابع درهم‌سازی را به صورت تصادفی از میان خانواده‌ای از توابع درهم‌سازی انتخاب می‌کنیم. از آنجایی که تابع درهم‌سازی را به صورت تصادفی انتخاب می‌کنیم، الگوریتم در هر اجرا رفتار متفاوتی دارد.

¹ random hashing

² universal hashing

- فرض کنید H یک خانواده محدود از توابع درهم سازی باشد که یک مجموعه مرجع U از کلیدها را به بازه $\{0, 1, \dots, m-1\}$ نگاشت می کند. به این خانواده یک خانواده سراسری گفته می شود اگر به ازای هر جفت از کلیدهای متمایز $k_1, k_2 \in U$ ، تعداد توابع $h \in H$ که در رابطه $h(k_1) = h(k_2)$ صدق می کنند، حداکثر $|H|/m$ باشد.
- به عبارت دیگر با انتخاب یک تابع درهم سازی به صورت تصادفی که از مجموعه H انتخاب شده باشد، احتمال تصادم بین دو کلید متمایز k_1 و k_2 بیشتر از احتمال تصادم $1/m$ وقتی که $h(k_1)$ و $h(k_2)$ به صورت تصادفی و مستقل از مجموعه $\{0, 1, \dots, m-1\}$ انتخاب شده باشند نیست.

- در اینجا دو روش برای طراحی یک خانواده سراسری از توابع درهم سازی معرفی می کنیم.
- روش اول : عدد اول p که به اندازه کافی بزرگ است را انتخاب می کنیم به طوری که هر کلید k در محدوده 0 تا $p - 1$ قرار بگیرد.
- فرض کنید \mathbb{Z}_p مجموعه $\{0, 1, 2, \dots, p - 1\}$ باشد و \mathbb{Z}_p^* مجموعه $\{1, 2, \dots, p - 1\}$.
- از آنجایی که اندازه مجموعه مرجع کلیدها بزرگتر از تعداد مکان ها در جدول درهم سازی است، داریم $p > m$.

- به ازای هر $a \in \mathbb{Z}_p^*$ و $b \in \mathbb{Z}_p$ ، تابع h_{ab} را به صورت یک تبدیل خطی و یک کاهش با استفاده عملگر باقیمانده بر p و سپس باقیمانده بر m تعریف می کنیم.

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m$$

- برای مثال اگر $p = 17$ و $m = 6$ باشد، داریم

$$\begin{aligned} h_{3,4}(8) &= ((3 \times 8 + 4) \bmod 17) \bmod 6 \\ &= (28 \bmod 17) \bmod 6 \\ &= 11 \bmod 6 \\ &= 5 \end{aligned}$$

- به ازای p و m ، خانواده همه توابع درهم سازی به صورت زیر است.

$$H_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

- یک ویژگی این خانواده از توابع درهم سازی این است که اندازه m می تواند هر مقدار دلخواهی باشد و نیازی نیست اول باشد.

- از آنجایی که می توانیم تعداد $p - 1$ مقدار برای a و p مقدار برای b انتخاب کنیم، خانواده H_{pm} شامل $p(p - 1)$ تابع درهم سازی است.

- می توان اثبات کرد H_{pm} یک خانواده سراسری است.

- روش دوم : از این روش در عمل بیشتر استفاده می شود زیرا کارایی بالایی دارد.
- فرض کنید H یک خانواده از توابع درهم سازی ضرب-انتقال صورت زیر باشد.
 $H = \{h_a : h_a \text{ یک تابع درهم سازی ضرب انتقال است و } 1 \leq a < m \text{ و } a \text{ فرد است} : \}$
- تابع درهم سازی ضرب-انتقال را قبلا به صورت زیر تعریف کردیم.
$$h_a(k) = (ka \bmod 2^w) \gg (w - l)$$
- می توان اثبات کرد توابع خانواده H یک خانواده سراسری هستند.

- درهم سازی ورودی های بزرگ مانند وکتورها یا رشته ها:
- برخی مواقع ورودی تابع درهم سازی آنقدر بزرگ است که به سادگی نمی توان یک عدد اول p پیدا کرد که در ۶۴ بیت جای بگیرد. برای مثال ممکن است بخواهیم یک رشته را به عنوان کلید در نظر بگیریم.
- یک روش درهم سازی برای چنین ورودی هایی تعمیم روش های شرح داده شده است. برای این کار می توانیم از توابع درهم سازی رمزنگاری استفاده کنیم.

درهم سازی تصادفی

- یک تابع رمزنگاری یک ورودی با اندازه دلخواه را دریافت می کند و یک خروجی با طول ثابت تولید می کند.
- برای مثال تابع استاندارد SHA-256 یک خروجی ۲۵۶ بیتی (۳۲ بایتی) تولید می کند.
- توابع رمزنگاری به قدری مورد استفاده اند که برخی از پردازنده ها توابع رمزنگاری را به صورت سخت افزاری پیاده سازی می کنند و بنابراین سرعت اجرای بالایی دارند.
- می توانیم تابع درهم سازی برای یک کلید با طول دلخواه را به صورت زیر تعریف کنیم.

$$h(k) = \text{SHA-256}(k) \bmod m$$

- برای تعریف یک خانواده از توابع درهم سازی می توانیم یک رشته a به ابتدای ورودی الحاق کنیم.

$$h_a(k) = \text{SHA-256}(a\|k) \bmod m$$

- آدرس دهی باز¹ روشی برای رفع تصادم است که برخلاف زنجیرسازی از فضایی خارج از جدول درهم سازی استفاده نمی کند.
- در آدرس دهی باز همه عناصر در خود جدول درهم سازی ذخیره می شوند و از فضای اضافی استفاده نمی شود. بنابراین جدول درهم سازی ممکن است پر شود. در نتیجه ضریب بار α هیچ گاه نمی تواند بیشتر از ۱ شود.

¹ open addressing

- در این روش تصادم به صورت زیر مدیریت می شود : وقتی می خواهیم یک عنصر در جدول وارد کنیم، این عنصر جدید در صورت امکان در مکانی که به عنوان انتخاب اول آن محاسبه می شود قرار می گیرد. این روند ادامه پیدا می کند تا جایی که یک مکان خالی پیدا شود.
- برای جستجوی یک عنصر مکان های جدول به ترتیب انتخاب ها بررسی می شوند تا جایی که یا عنصر یافته شود و یا به یک خانه خالی برخورد کنیم که در این صورت پیام جستجوی ناموفق صادر می کنیم.
- آدرس دهی باز از هیچ اشاره گر و فضای اضافی استفاده نمی کند، در نتیجه می توان از جدول بزرگتری برای داده های آن استفاده کرد.

- برای درج در جدول توسط آدرس دهی باز، به ترتیب همه انتخاب ها برای مکان کلید مورد نظر را واری¹ می کنیم تا وقتی که یک فضای خالی در جدول پیدا شود.
- تابع درهم سازی به ازای یک کلید باید شماره انتخاب یا شماره واری² را به عنوان ورودی دوم دریافت کند. بنابراین دامنه و برد تابع درهم سازی به صورت زیر خواهد بود.

$$h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$$

- در آدرس دهی باز نیاز داریم به ازای هر کلید k ، یک دنباله انتخاب ها یا دنباله واری³ صورت $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ تولید کنیم تا همه مکان های جدول مورد بررسی قرار بگیرند.

¹ probe

² probe number

³ probe sequence

- تابع Hash-Insert جدول T و کلید k را دریافت می‌کند و مکان ذخیره آن را باز می‌گرداند و در صورتی که جدول پر باشد پیام خطا صادر می‌کند.

Algorithm Hash Insert

```
function HASH-INSERT(T,k)
1: i = 0
2: repeat
3:   q = h(k,i)
4:   if T[q] == NIL then
5:     T[q] = k
6:     return q
7:   else i = i + 1
8: until i == m
9: error "hash table overflow"
```

- الگوریتم جستجوی کلید k نیز دنباله ای از مکان های جدول را بررسی می کند تا وقتی که کلید k یافته شود. وقتی به یک مکان خالی می رسیم جستجو با پیامی مبنی بر عدم موفقیت و بازگرداندن NIL به پایان می رسد.

- تابع Hash-Search جدول T و کلید k را دریافت می‌کند و در صورتی که مکان q شامل کلید k پیدا شد مکان q را باز می‌گرداند و در صورتی که k در جدول وجود نداشت مقدار NIL را باز می‌گرداند.

Algorithm Hash Search

```
function HASH-SEARCH(T,k)
1: i = 0
2: repeat
3:   q = h(k,i)
4:   if T[q] == k then
5:     return q
6:   i = i + 1
7: until T[q] == NIL or i == m
8: return NIL
```

- حذف کردن از جدول درهم سازی با آدرس دهی باز کمی پیچیده تر است. وقتی یک کلید از مکان q را حذف می کنیم ممکن است به اشتباه آن مکان به عنوان مکان خالی در نظر گرفته شود. اگر مکان حذف شده را با NIL علامتگذاری کنیم، ممکن است نتوانیم کلید k را که مکان q برای آن واریسی یافته شده بوده در حالی که مکان اصلی آن اشغال بوده است را پیدا کنیم.
- برای حل این مشکل، وقتی یک واریسی کلید را در جدول حذف می کنیم به جای علامتگذاری آن با مقدار NIL ، مقدار Deleted را در مکان حذف شده قرار می دهیم.

- در هنگام درج در جدول، تابع Hash-Insert مکانی که با Deleted علامتگذاری شده است را به عنوان مکان خالی در نظر می گیرد. در هنگام جستجو اما وقتی تابع Hash-Search به مکان Deleted برخورد می کند جستجو را ادامه می دهد، زیرا مکانی که با Deleted علامتگذاری شده در هنگام درج کلید مورد جستجو اشغال بوده است.
- زمان جستجو در روش آدرس دهی باز به علت استفاده از این علامتگذاری به ضریب α بستگی ندارد و به همین دلیل وقتی می خواهیم کلیدها را حذف کنیم از روش زنجیرسازی استفاده می کنیم که زمان جستجوی بهتری دارد.

- دو روش برای طراحی توابع درهم سازی توسط آدرس دهی باز وجود دارد که در اینجا در مورد آنها صحبت خواهیم کرد : درهم سازی دوگانه و واریسی خطی.

- درهم سازی دوگانه ¹ : درهم سازی دوگانه یکی از بهترین روش های موجود برای آدرس دهی باز است.
- درهم سازی دوگانه از تابع درهم سازی به صورت زیر استفاده می کند.

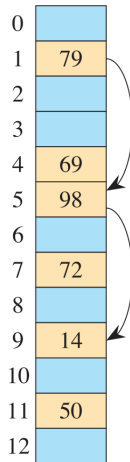
$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

- به طوری که h_1 و h_2 دو تابع درهم سازی هستند.
- اولین مکان انتخابی $T[h_1(k)]$ است و مکان های انتخابی بعدی گام هایی به اندازه ضربی از $h_2(k)$ بعد از مکان انتخابی اول هستند.

¹ double hashing

آدرس دهی باز

- در شکل زیر یک مثال از درهم سازی دوگانه نشان داده شده است.



- اندازه جدول ۱۳ است و $h_1(k) = k \bmod 13$ و $h_2(k) = 1 + (k \bmod 11)$.
- کلید 14 بعد از واریسی مکان‌های 1 و 5 که اشغال شده‌اند وارد مکان 9 می‌شود.

- برای این که همه جدول مورد جستجو قرار بگیرد مقدار $h_2(k)$ باید نسبت به اندازه جدول یعنی m اول باشد. یک روش مناسب برای اطمینان حاصل کردن از برقراری این شرط است که m را توانی از ۲ انتخاب کنیم و h_2 را به گونه ای طراحی کنیم که همیشه عدد فرد تولید کند. یک روش دیگر این است که m را اول انتخاب کنیم و h_2 را به گونه ای طراحی کنیم که عددی مثبت کوچکتر از m تولید کند.
- برای مثال می توانیم m را یک عدد اول انتخاب کنیم و توابع را به صورت زیر طراحی کنیم.

$$h_1(k) = k \bmod m$$

$$h_2(k) = 1 + (k \bmod m')$$

- به طوری که m' عددی است که کمی کوچکتر از m است (مثلاً $m - 1$).

- به عنوان مثال، اگر $k = 123456$ و $m = 701$ باشد و $m' = 700$ باشد آنگاه $h_1(k) = 80$ و $h_2(k) = 257$ خواهد بود، بنابراین اولین واریسی مکان 80 را انتخاب می کند و در واریسی های بعدی ضرایبی از 257 مکان بعدی (با محاسبه باقیمانده بر m) انتخاب می شوند تا وقتی که یک مکان خالی پیدا شود.
- اگرچه مقادیر m که اول یا توانی از ۲ نیستند می توانند در درهم سازی دوگانه استفاده شوند، اما در عمل انتخاب $h_2(k)$ سخت تر می شود، زیرا باید اطمینان حاصل شود که نسبت به m اول است.
- وقتی m عدد اول یا توانی از ۲ باشد، درهم سازی دوگانه تعداد $\Theta(m^2)$ دنباله واریسی تولید می کند زیرا هر جفت $(h_1(k), h_2(k))$ یک دنباله واریسی متفاوت است.

- واریسی خطی¹ : واریسی خطی یک حالت خاص از درهم سازی دوگانه است و یکی از روش های بسیار ساده در آدرس دهی باز برای رفع تصادم است.
- همانند درهم سازی چندگانه ابتدا مکان $T[h_1(k)]$ بررسی می شود. اگر این مکان اشغال بود، مکان بعدی یعنی $T[h_1(k) + 1]$ بررسی می شود. بررسی تا $T[m - 1]$ ادامه پیدا می کند و مجدداً به $T[0]$ باز می گردد تا به $T[h_1(k) - 1]$ برسد.
- بنابراین گام واریسی که در درهم سازی دوگانه برابر با $h_2(k)$ بود در واریسی خطی برابر با ۱ است پس $h_2(k) = 1$ به ازای همه مقادیر k .

¹ linear probing

- پس تابع درهم سازی در واریسی خطی به ازای $i = 0, 1, \dots, m - 1$ برابر است با :

$$h(k, i) = (h_1(k) + i) \bmod m$$

- مقدار $h_1(k)$ می تواند هریک از مقادیر $\{0, 1, \dots, m - 1\}$ باشد.

- در اینجا آدرس دهی باز را بر اساس ضریب بار جدول درهم سازی یعنی $\alpha = n/m$ تحلیل می کنیم.
- در آدرس دهی باز، حداکثر یک عنصر در یک مکان ذخیره می شود و بنابراین $n \leq m$ است و نتیجه $\alpha \leq 1$ است.
- از آنجایی که حذف از جدول درهم سازی تأثیری در فرایند جستجو ندارد فرض می کنیم حذف وجود ندارد.

- همانند قبل فرض می‌کنیم درهم‌سازی جایگشت‌ها را یکنواخت و مستقل تولید می‌کند. بنابراین در این شرایط دنباله واریسی $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ که برای درج یا جستجو برای کلید k استفاده می‌شود با احتمال برابر می‌تواند هریک از جایگشت‌های $\langle 0, 1, \dots, m-1 \rangle$ باشد. البته هر کلید یک دنباله واریسی یکتا خواهد داشت.
- حال تعداد مورد انتظار واریسی‌ها برای درهم‌سازی با آدرس‌دهی باز را با شروع از تعداد مورد انتظار واریسی‌ها در یک جستجوی ناموفق تحلیل می‌کنیم.
- کران به دست آمده برابر است با $1 + \alpha + \alpha^2 + \alpha^3 + \dots$
- اولین واریسی همیشه رخ خواهد داد، اگر مکان اول انتخاب شده با احتمال α اشغال باشد، انتخاب دوم بررسی می‌شود. احتمال این‌که هر دو انتخاب اول و دوم اشغال باشند برابر است با $(\alpha)^2$ و به همین ترتیب الی آخر.

- قضیه : به ازای یک جدول درهم سازی با آدرس دهی باز و ضریب بار $\alpha = n/m < 1$ ، تعداد واریسی ها در یک جستجوی ناموفق حداکثر $1/(1 - \alpha)$ است.

- اگر α ثابت باشد، این قضیه پیش بینی می کند که جستجوی ناموفق در زمان $O(1)$ اجرا می شود.
- برای مثال، اگر جدول درهم سازی نیمه پر لیوان باشد، تعداد متوسط واریسی ها در یک جستجوی ناموفق حداکثر $2 = 1/(1 - 0.5)$ است. اگر 90% درصد اشغال باشد، تعداد متوسط واریسی ها حداکثر $10 = 1/(1 - 0.9)$ است.

- بنابراین درج یک عنصر در جدول درهم سازی با آدرس دهی باز در جدولی با ضریب بار α حداکثر $1(1 - \alpha)$ واریسی به طور میانگین نیاز دارد.

- قضیه : به ازای یک جدول درهم سازی با آدرس دهی باز و ضریب بار $\alpha > 1$ ، تعداد مورد انتظار واریسی ها در یک جستجوی موفق برابر است با حداکثر

$$\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$$

- اثبات : در جستجوی کلید k همان دنباله واریسی تولید می شود که در هنگام درج کلید k تولید شده است. اگر $(i + 1)$ امین کلید درج شده در جدول درهم سازی کلید k باشد، ضریب بار در هنگام درج i/m بوده است و بنابراین طبق قضیه قبل تعداد مورد انتظار واریسی ها در جستجوی کلید k برابر با $1(1 - i/m) = m/(m - i)$ خواهد بود.
- اگر بر روی همه n کلید در جدول میانگین بگیریم مقدار زیر را برای مقدار مورد انتظار واریسی ها در یک جستجوی ناموفق به دست می آوریم :

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} \\ &= \frac{m}{n} \ln \frac{m}{m-n} \\ &= \frac{1}{\alpha} \ln \frac{1}{1-\alpha} \end{aligned}$$

- در یک جدول درهم سازی نیمه پر ، تعداد مورد انتظار واری‌ها در یک جستجوی موفق برابر با 1.387 است. اگر ۹۰ درصد جدول اشغال باشد، تعداد مورد انتظار واری‌ها کمتر از 2.559 است. اگر $\alpha = 1$ باشد، در یک جستجوی ناموفق، همه m مکان جدول باید واری شوند.