داده‌ساختار احتمالاتی– توابع درهم‌سازی

محسن هوشمند،
دانشکده علم و فن رایانه
دانشگاه تحصیلات تکمیلی علوم پایه زنجان

فهرست

درهم‌ساز

مقدمه

درهم‌ساز برای تصادفی‌سازی و نمایش فشرده داده‌ها استفاده دارد و نقشی اصلی را در «داده‌ساختار احتمالاتی» ایفا می‌کند. تابع درهم‌ساز قطعات داده ورودی با هر اندازه‌ای را به شناسه‌ای عددی با اندازهٔ کوچکتر (و غالبا ثابت) به نام مقدار درهم‌ساز یا درهم‌ساز نگاشت می‌کند. توابع درهم‌ساز ورودی را فشرده می‌کنند، بنابراین، یکسانی حاصل از اعمال تابع درهم‌ساز بر دو قطعه دادهٔ متفاوت اجتناب‌ناپذیر هستند و «تصادم» معرفی می‌شود.

درهم‌سازی از حوزه‌هایی است که مستقل از میزان توجهی که در دوره‌های برنامه‌نویسی، ساختمان داده‌ها و الگوریتم‌ها به آن شده باشد، غالباً نیازمند بررسی عمیق‌تر است. داده‌ساختارهای مبتنی بر درهم‌سازی مانند جدول‌های درهم و نیز توابع درهم، در لایه‌های مختلف سامانه‌های نرم‌افزاری و شبکه‌ای حضور گسترده دارند. برای تبیین این موضوع، می‌توان فرایند به ظاهر سادهٔ نگارش رایانامه‌ای را لحاظ کرد.
در نخستین گام، هنگام احراز هویت و ورود به حساب کاربری، گذرواژهٔ کاربر با تابعی درهم‌ساز پردازش می‌شود و مقدار درهم حاصل با مقدار ذخیره‌شده در پایگاه داده مقایسه می‌گردد. این سازوکار، رکن اصلی امنیت در بسیاری از سامانه‌های احراز هویت به شمار می‌رود. در مرحلهٔ تدوین متن ایمیل، سامانهٔ بررسی املایی از درهم‌سازی برای تعیین وجود یا عدم وجود یک واژه در واژه‌نامهٔ داخلی بهره می‌گیرد. این کاربرد، نشان‌دهندهٔ نقش مهم درهم‌سازی در تسریع عملیات جست‌وجو است.
در هنگام ارسال ایمیل، معمولاً زوج آدرس‌های IP مبدأ و مقصد درهم می‌شوند تا مشخص گردد بستهٔ داده باید به کدام سرور میانی منتقل شود. این روش، بخشی از سازوکار توزیع بار در شبکه است و موجب کارایی بیشتر در مسیریابی بسته‌ها می‌شود. نهایتاً در سمت مقصد، برخی سامانه‌های پالایش هرزنامه از درهم‌سازی محتوای پیام بهره می‌برند تا نشانه‌های مرتبط با پیام‌های ناخواسته را شناسایی و پیام‌های مشکوک را فیلتر کنند.
فصل جاری هر دو موضوعِ درهم‌سازی و جدول‌های درهم را در بر می‌گیرد. هرچند این دو مفهوم دارای شباهت بسیار فراوانی هستند و گاهی با یکدیگر خلط مبحث می‌شوند ولی یکسان نیستند، در این فصل درهم‌سازی را در زمینهٔ کاربرد آن در جدول‌های درهم و مختصری در زمینهٔ رمزنگاری بررسی می‌کنیم، که به نحوی پایه‌ای برای فصل‌های آینده است که درهم‌سازی در سایر داده‌ساختارها مورد استفاده قرار می‌گیرد. جدول‌های درهم، همچون خود درهم‌سازی، ساختارهایی فراگیر هستند و برنامه‌نویسان معمولاً بدون اینکه بدانند، روزمره از نگاشت‌های کلید-مقدار از جدول درهم بهره می‌برند.
جهت درک دلیل استفادهٔ گسترده از جدول‌های درهم، باید آن‌ها را با سایر ساختارهای داده مقایسه کنیم و ببینیم هر یک تا چه اندازه برای پیاده‌سازی داده‌ساختارِ لغت‌نامه، نوع انتزاعی داده‌ای که سه عمل اصلی جست‌وجو، درج و حذف را پشتیبانی می‌کند، مناسب‌اند.
سخن کوتاه، در هر حوزه‌ای که امنیت یا سرعت بازیابی داده اهمیت داشته باشد، احتمال استفاده از درهم‌سازی بسیار بالا است. نقش بنیادی این سازوکار در سامانه‌های نو اطلاعاتی سبب شده است که فهم دقیق آن برای متخصصان علوم رایانه و امنیت ضرورت محسوب شود.

ساختارهای دادهٔ متعددی می‌توانند نقش «لغت‌نامه» را ایفا کنند، اما هر یک الگوهای کارایی متفاوتی دارند و به همین دلیل برای پیرنگ‌های کاربردی گوناگون مناسب‌اند. برای مثال، یک آرایهٔ سادهٔ نامرتب—با وجود سادگی ساختاری—در عمل درج، کارایی زمانی ثابت $O(1)$ ارائه می‌دهد زیرا عناصر جدید به انتهای آرایه افزوده می‌شوند. با این حال، عملیات جست‌وجو در بدترین حالت

نیازمند اسکن کامل آرایه و بنابراین دارای هزینهٔ زمانی خطی O(n) است. چنین ساختاری تنها زمانی گزینهٔ قابل قبول برای پیاده‌سازی لغت‌نامه محسوب می‌شود که درج بسیار پرتکرار و جست‌وجو بسیار کم‌تکرار باشد.

آرایه‌های مرتب، امکان جست‌وجوی سریع‌تر را با بهره‌گیری از جست‌وجوی دودویی فراهم می‌کنند و زمان جست‌وجو را به $O(log\ n)$ کاهش می‌دهند. مقدار مذکور برای بسیاری از اندازه‌های آرایه عملاً بسیار نزدیک به زمان ثابت است (مثلاً لگاریتم دودوییِ یک میلیارد کمتر از ۳۰ است). با این حال، حفظ ترتیب مرتب در هنگام درج یا حذف، هزینهٔ زمانی خطی تحمیل می‌کند زیرا در بدترین حالت لازم است تعداد زیادی از عناصر جابه‌جا شوند. هزینهٔ خطی معمولاً در بسیاری از برنامه‌ها نامطلوب است.

فهرست‌های پیوندی، برخلاف آرایه‌های مرتب، درج را در زمان ثابت $O(1)$، با افزودن مقدار در ابتدای فهرست، امکان‌پذیر می‌کنند. همچنین حذف عضو، در صورت داشتن اشاره‌گر مربوطه، با تغییر چند پیوند در زمان ثابت انجام می‌شود. البته در فهرست‌های پیوندی تک‌سویه، حذف نیازمند دانستن از عضوِ پیش از عضو هدف است و یافتن آن مستلزم پیمایش پیوندها است، که مجدداً زمان خطی ایجاد می‌کند. در مجموع، در ساختارهای خطی مانند آرایه و فهرست پیوندی، همواره حداقل یک عملیات وجود دارد که هزینهٔ آن O(n) است، و برای اجتناب از آن باید از ساختارهای غیرخطی استفاده کرد.

درخت‌های جست‌وجوی دودویی متوازن، عملیات لغت‌نامه را متناسب با عمق درخت اجرا می‌کنند و مکانیسم‌های متوازن‌سازی مانند AVL و قرمز–سیاه عمق را در حد $O(log\ n)$ نگه می‌دارند. بنابراین عملیات درج، جست‌وجو و حذف در بدترین حالت زمان لگاریتمی دارند. چون زمان لگاریتمی از نظر کارایی بسیار نزدیک‌تر به زمان ثابت است تا زمان خطی، این ساختارها برای بسیاری از سناریوهای عملی مناسب‌اند. افزون بر این، درخت‌های دودویی متوازن نظم مقادیر را حفظ می‌کنند و برای اجرای سریع پرس‌وجوهای بازه‌ای، همچنین یافتن مقادیر بعدی و قبلی در ترتیب ، انتخابی طبیعی محسوب می‌شوند. از منظر تئوریک، درخت‌های دودویی متوازن بهترین عملکرد ممکن را میان ساختارهایی دارند که تنها بر پایهٔ مقایسهٔ اعضاء عمل می‌کنند.

با این حال، طراحی ساختارهای دادهٔ کاراتر محدود به عملیات مقایسه نیست. رایانه‌ها عملیات متنوعی مانند شیفت‌های بیتی، محاسبات عددی و سایر تبدیل‌ها را نیز پشتیبانی می‌کنند و توابع درهم‌ساز از همین عملیات به‌صورت هوشمندانه استفاده می‌کنند تا محدودیت‌های لگاریتمی را دور بزنند.

مزیت بنیادین جدول‌های درهم‌ساز این است که هزینهٔ همهٔ عملیات لغت‌نامه‌ای را به میانگین زمان ثابت $O(1)$ کاهش می‌دهند. هرچند این زمان برای بدترین حالت تضمین‌شده نیست، زیرا در بدترین شرایط همچنان ممکن است هزینهٔ خطی O(n) رخ دهد، اما طراحی مناسب جدول درهم‌ساز تقریباً همیشه این شرایط را نادر می‌کند. این تفاوت مهمی با ساختارهای خطی دارد: در حالی که جست‌وجو در آرایهٔ نامرتب به شکلی پایدار و قابل پیش‌بینی خطی است، رخداد بدترین حالت در جدول درهم‌ساز بسیار نادر است.

علت این برتری آن است که تابع درهم‌ساز، دادهٔ ورودی را «گرد» کرده و نتیجه را برای تعیین محل ذخیره‌سازی عنصر در یکی از «سطل»های جدول درهم‌ساز به کار می‌گیرد. واژهٔ hash از واژهٔ فرانسوی hachis گرفته شده که نوعی غذای خردشده است. تشبیهی مناسب از اینکه چگونه ورودی‌ها به مقادیر خرد و پراکنده تبدیل می‌شوند. چون به‌طور میانگین عناصر مختلف به مقادیر متفاوتی درهم می‌شوند، داده‌ها در سطل‌ها نسبتاً یکنواخت پراکنده می‌گردند و این امر جست‌وجو را سریع نگه می‌دارد. با این حال، موارد استثنائی وجود دارند که در آن‌ها چند ورودی غیرمرتبط به یک مقدار یکسان درهم می‌شوند و همگی در یک سطل قرار می‌گیرند.

در این وضعیت، یافتن عضو نیازمند اسکن همهٔ عناصر آن سطل است. اگر چنین رخدادی تکرار شود، می‌توان از تابع درهم دیگری استفاده کرد.

در مقابل، جدول‌های درهم برای کاربردهایی که حفظ ترتیب عناصر اهمیت دارد مناسب نیستند. «رند» کردن داده‌ها ذاتاً ترتیب را از میان می‌برد و در نتیجه پاسخ‌دادن به پرس‌وجوهای وابسته به نظم، از جمله پرس‌وجوهای بازه‌ای در پایگاه داده، مانند یافتن تمام سنین بین ۳۵ تا ۵۶ سال یا نقاط هندسی با مختصات x بین ۳ تا ۴۵، با جدول درهم دشوار یا ناکارآمد است. جدول‌های درهم بهترین عملکرد را هنگامی ارائه می‌دهند که مسئله یافتن تطابقی دقیق باشد. البته، با استفاده از تکنیک‌هایی خاص، می‌توان از درهم‌سازی برای پرس‌وجوهای مشابهت نیز برای نمونه، در تشخیص سرقت ادبی و انتحال استفاده کرد. جدول زیر مقایسه‌ای از رایج‌ترین ساختارهای داده ارائه می‌دهد.

	جستجو	درج	حذف	یافتن قبلی و بعدی
آرایهٔ نامرتب	$O(n)$	$O(1)$	$O(n)$	$O(n)$
آرایهٔ مرتب	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$
فهرست پیوندی	$O(n)$	$O(1)$	$O(1)$	$O(n)$
درخت جستجوی دودوئی متعادل	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
جدول درهم	$O(1)$	$O(1)$	$O(1)$	$O(n)$

حذف افزونگی داده‌ها در سامانه‌های پشتیبان‌گیری و ذخیره‌سازی

بسیاری از شرکت‌ها، از جمله Dropbox و سامانه‌های ذخیره‌سازی Dell EMC Data Domain، با چالش نگهداری حجم بسیار بزرگی از داده‌های کاربران مواجه هستند. مشتریان این سامانه‌ها غالباً سازمان‌های بزرگ هستند که مجموعه‌های عظیمی از داده‌ها را مدیریت می‌کنند. در چنین محیط‌هایی معمولاً از داده‌ها به‌طور منظم نسخهٔ پشتیبان یا snapshot تهیه می‌شود. اگر این snapshot-ها با فاصله‌های زمانی کوتاه، برای مثال هر ۲۴ ساعت، گرفته شوند، بخش عمده‌ای از داده‌ها میان دو snapshot متوالی بدون تغییر باقی می‌ماند. در چنین شرایطی، بسیار مهم است که بتوان بخش‌هایی از داده را که تغییر کرده‌اند به‌سرعت شناسایی کرد و تنها همان بخش‌ها را ذخیره نمود. این رویکرد باعث صرفه‌جویی قابل توجهی در زمان و فضای ذخیره‌سازی می‌شود. برای دستیابی به این هدف، لازم است بتوان محتوای تکراری را به‌صورت کارآمد تشخیص داد.

فرایند حذف داده‌های تکراری Deduplication نام دارد و در اغلب پیاده‌سازی‌های متأخر آن از توابع درهم‌ساز استفاده می‌شود. به عنوان نمونه، سامانه‌ای به نام ChunkStash که برای ارائهٔ گذردهی بالا با استفاده از حافظهٔ فلش طراحی شده است، از این روش بهره می‌گیرد. در این سامانه، فایل‌ها به قطعه‌های کوچکی با اندازهٔ ثابت مثلا هشت کیلوبایت تقسیم می‌شوند. سپس، محتوای هر قطعه با استفاده از تابع درهم SHA-۱ به اثر انگشتی (Fingerprint) ۲۰ بایتی تبدیل می‌شود. اگر این اثر انگشت پیش‌تر در سیستم ثبت شده باشد، به این معناست که همان قطعه قبلاً ذخیره شده است و بنابراین تنها یک اشاره‌گر به دادهٔ موجود نگهداری می‌شود. در مقابل، اگر اثر انگشت جدید باشد، فرض می‌شود که قطعهٔ داده نیز جدید است. در نتیجه، هم خود قطعه در مخزن داده ذخیره می‌شود و هم اثر انگشت آن همراه با اشاره‌گری به محل ذخیرهٔ قطعه در جدول درهم‌ساز ثبت می‌گردد.

تقسیم فایل‌ها به قطعه‌های کوچک مزیت مهم دیگری نیز دارد: این روش امکان شناسایی داده‌های تقریباً تکراری را فراهم می‌کند. به عبارت دیگر، اگر تنها بخش کوچکی از فایلی بزرگ تغییر کرده باشد، تنها همان قطعهٔ تغییر یافته به عنوان دادهٔ جدید ذخیره می‌شود و سایر قطعه‌ها که بدون تغییر باقی مانده‌اند مجدداً ذخیره نخواهند شد.

البته پیاده‌سازی عملی این فرایند شامل جزئیات بیشتری است. برای مثال، هنگام نوشتن قطعه‌های جدید در حافظهٔ فلش، این قطعه‌ها ابتدا در بافر نوشتن در حافظهٔ اصلی جمع‌آوری می‌شوند. هنگامی که بافر مزبور پر شد، تمامی داده‌ها به‌صورت یکجا در حافظهٔ فلش نوشته می‌شوند. این کار به این دلیل انجام می‌شود که انجام نوشتن‌های کوچک و مکرر روی صفحهٔ فلش عملیاتی پرهزینه محسوب می‌شود. با این حال، در این مرحله تمرکز بر عملیات درون حافظه است. مباحث مربوط به بافرکردن و نوشتن کارآمد روی دیسک در بخش‌های بعدی به‌طور مفصل‌تر بررسی خواهند شد.

تشخیص سرقت ادبی با MOSS و اثرانگشتی‌سازی رابین-کارپ

سامانهٔ Measure of Software Similarity (MOSS) خدمتی تخصصی برای تشخیص انتحال و سرقت ادبی است که عمدتاً در ارزیابی تکالیف برنامه‌نویسی مورد استفاده قرار می‌گیرد. یکی از ایده‌های الگوریتمی اصلی در سامانهٔ MOSS بهره‌گیری از گونه‌ای اصلاح‌شده از الگوریتم تطبیق رشتهٔ رابین-کارپ است. الگوریتم مذکور بر تکنیک اثرانگشتی‌سازی k-gram (زیررشته‌های پیوسته با طول ثابت k) تکیه دارد. پیش از پرداختن به جزئیات MOSS، مروری بر الگوریتم اصلی ضروری است.

در مسئلهٔ تطبیق رشته، رشتهٔ t نمایانگر متن بزرگ و رشتهٔ p نمایانگر الگوی کوچک‌تری است که باید وجود یا عدم وجود آن در متن بررسی شود. بخش اعظم از پژوهش‌های این حوزه بر مقایسهٔ مستقیم زیررشته‌های t و p تکیه دارد، اما الگوریتم Rabin–Karp مقایسهٔ درهمِ زیررشته‌ها را مبنا قرار می‌دهد. این الگوریتم در عمل عملکرد بسیار سریعی دارد و بخشی از این سرعت،همان‌گونه که انتظار می‌رود، ناشی از استفادهٔ هوشمندانه از درهم‌سازی است.

در این الگوریتم، مقایسهٔ نویسه‌به‌نویسه در صورتی انجام می‌شود که درهم‌های دو زیررشته برابر گزارش شود. در بدترین حالت، تصادم‌های درهم زیاد رخ می‌دهند (حالتی که دو زیررشتهٔ متفاوت درهم یکسانی ایجاد کنند) و در نتیجه، الگوریتم ناچار است همانند روش جستجوی کامل، زمان $O(|t||p|)$ را صرف کند. با این حال، در بیشتر کاربردهای عملی که تعداد انطباق‌های واقعی اندک است و از تابع درهمی مناسب استفاده می‌شود، الگوریتم با سرعتی خطی و بسیار کارآمد عمل می‌کند.
مثال- الگوریتم اثرانگشتی رابین-کارپ، هدف آن است که الگوی p = BBBBC را در رشتهٔ بزرگ‌تر t = BBBBBBBBABBBDBBBBC پیدا کنیم. مقدار درهم الگو مدنظر BBBBC، برابر با ۱۶۲ است. در ابتدای رشتهٔ t زیررشتهٔ BBBBB قرار دارد که مقدار هش آن ۱۶۱ است. بنابراین، عدم‌انطباق رخ می‌دهد. با انتقال پنجره یک نویسه به راست، بارها و بارها به مقادیر درهم متفاوتی برمی‌خوریم که همگی با درهم الگو (۱۶۲) متفاوت‌اند.
در ادامهٔ جست‌وجو، زیررشتهٔ ABBBD با مقدار درهم ۱۶۲ ظاهر می‌شود. از آن‌جا که مقدار درهم با درهم الگو یکسان است، الگوریتم رابین-کارپ در این نقطه بررسی نویسه‌به‌نویسه را آغاز می‌کند. نتیجهٔ این بررسی نشان می‌دهد که دو زیررشته واقعاً یکسان نیستند. پس، با انطباق کاذب روبروئیم.

در انتهای رشتهٔ t دوباره مقدار درهم ۱۶۲ مشاهده می‌شود، این بار بررسی نویسه‌به‌نویسهٔ زیررشتهٔ BBBBC نشان می‌دهد که این زیررشته با الگو دقیقاً برابر است. در نتیجه، الگوریتم انطباقی واقعی (true match) گزارش می‌کند.

هزینهٔ محاسبهٔ درهم معمولاً به اندازهٔ زیررشته وابسته است. بنابراین، درهم‌سازی به‌تنهایی موجب افزایش سرعت نمی‌شود. اما، رابین-کارپ از درهم غلطان (rolling hash) بهره می‌گیرد که در آن، با داشتن درهم k-gram مربوط به بازهٔ $t[j, \dots, j+k-1]$، محاسبهٔ درهم k-gram انتقال‌یافتهٔ تک نویسه به راست، یعنی $t[j+1, \dots, j+k]$، تنها زمان ثابت نیاز دارد. این امر زمانی ممکن است که بتوان اثر نویسهٔ نخست را از زیررشتهٔ قبلی «حذف» کرد و اثر نویسهٔ جدید را «افزود». روشی ساده از چنین تابع درهمی جمع مقادیر اسکی نویسه‌هاست، هرچند در عمل توابع پیچیده‌تری استفاده می‌شوند.

کاربرد بی‌واسطهٔ رابین-کارپ جهت بررسی انتحال می‌تواند شامل شکستن دو تکلیف برنامه‌نویسی به قطعه‌های کوچک و اثرانگشتی‌کردن آن‌ها باشد. اما، هدفِ MOSS مقایسهٔ تعداد زیادی از تکالیف دریافتی و یافتن همهٔ موارد بالقوهٔ مشابهت است. این کار مستلزم مقایسهٔ همهٔ فایل‌ها با یکدیگر و در نتیجه زمان اجرایی درجه‌دوم است که در عمل غیرقابل‌پذیرش است.

برای حل این مشکل، MOSS تنها تعداد محدودی اثر انگشت را به‌عنوان نمایندهٔ هر فایل انتخاب می‌کند. سپس نمایهٔ معکوس ساخته می‌شود که هر اثر انگشت را به موقعیت‌های وقوع آن در اسناد نگاشت می‌کند. با استفاده از این ساختار، مجموعهٔ فایل‌هایی که دارای انطباق هستند استخراج می‌شود. این سازوکار از انجام مقایسهٔ نابجای همه‌باهم جلوگیری می‌کند.

روش‌های متفاوتی برای انتخاب اثرانگشت‌های نماینده وجود دارد. روشی که MOSS به‌کار می‌گیرد این است که هر پنجرهٔ متوالی از نویسه‌ها در فایل، برای مثال پنجره‌ای به طول ۵۰ نویسه، حداقلِ مقدار درهم میان k-gram-های موجود در آن پنجره را انتخاب کند. انتخاب اثرانگشت در هر پنجره کمک می‌کند انطباق‌های طولانی و پیوسته میان دو سند از دست نروند و کارایی تشخیص افزایش یابد.

توابع درهم‌ساز جامع

جی. لارنس کارتر و مارک وگمن در سال ۱۳۵۸ شمسی توابع درهم‌ساز جامعی را پیشنهاد کردند که ویژگی‌های ریاضی آنها می‌تواند میانگین کمتری از تصادم‌ها را علی‌رغم انتخاب تصادفی از بین کل داده‌های موجود در هستی تضمین کند. خانواده توابع درهم‌ساز جامع H هر عضوی از دامنه ورودی را به مجموعه گسستهٔ $\{0, 1, \dots, m-1\}$ نگاشت و احتمال تصادمی پایین را با انتخاب تصادفی تابعی درهم‌ساز از خانواده زیر تضمین می‌کند:

$$pr[h(x) = h(y)] \leq \frac{1}{m} \forall x, y: x \neq y$$

بنابراین، انتخاب تصادفی تابعی درهم‌ساز از خانواده با ویژگی بالا دقیقاً همان انتخاب مقداری با احتمال یکنواخت تصادفی است. بسیاری از برنامه‌ها از خانواده درهم‌ساز زیر استفاده می‌کنند:

$$h_k(x) = [(k \cdot x)\%p]\%m$$

که علامت % تابع محاسبهٔ باقیماندهٔ تقسیم، در آن k عدد صحیحی تصادفی به پیمانهٔ p و $k \neq 0$ هستند. مقدار p عدد اول با شرط $p \geq m$ است. البته خانواده مذکور صرفا تقریبا جامع است، اما همچنان امید ریاضی احتمال تصادم کمتر از $\frac{2}{m}$ را تضمین می‌کند. صورت زیر خانواده‌ای پیچیده‌تر از توابع درهم‌ساز جامع برای درهم‌ساز اعداد صحیح است:

$$h_{k,q}(x) = \big((k \cdot x + q)\%\, p\big)\%\, m$$

که در آن k و q اعداد صحیح تصادفی به پیمانهٔ p و $k \neq 0$ هستند. مقدار p عدد اول با شرط $p \geq m$ است و معمولا برابر با یکی از اعداد اول مرسن شناخته شده خواهد شد. مثلا، برای $m = 10^9$ می‌توان از $p = M_{31} = 2^{31} - 1 \approx 2\,010^9$ استفاده کرد. لازم به ذکر است که خانواده‌های توابع درهم‌ساز فوق محدود به اعداد صحیح هستند، که برای اکثر کاربردهای عملی که نیاز به درهم‌ساز بردارهای با اندازه متغیر و همچنین نیاز به توابع درهم‌ساز سریع و مطمئن و دارای ویژگی‌های خاص هستند کارائی ندارند. به دیگر سخن، در عمل رده‌های فراوانی از توابع درهم‌ساز وجود دارد که انتخاب هر رده به طراحی آنها و نیاز مدنظر بستگی دارد. در ادامه چند مورد مهم از توابع درهم‌ساز مورد استفاده در داده‌ساختارهای احتمالی معرفی می‌شود.

توابع درهم‌ساز رمزنگاری

«توابع درهم‌ساز رمزنگاری» نگاشت‌های ثابت از رشته‌های بیتی ورودی با طول متغیر به رشته‌های بیتی خروجی با طول ثابت تعریف می‌شوند. همانطور که قبلاً بیان شد، تصادم درهم‌ساز اجتناب‌ناپذیر است، اما تابع درهم‌ساز امن باید «مقاوم» به تصادم باشد، به این معنی که دستیابی به تصادم‌ها دشوار باشد. البته، ممکن است تصادفا تصادمی یافت یا از قبل محاسبه شود. به همین دلیل چنین رده‌ای از توابع همیشه به اثبات‌های ریاضی نیاز دارد. توابع درهم‌ساز رمزنگاری در رمزنگاری بسیار مهم هستند و در بسیاری از کاربردها مانند امضاهای دیجیتال، طرح‌های احراز هویت و یکپارچگی پیام استفاده می‌شوند. از درهم‌سازهای رمزنگاری انتظار برآورده‌سازی سه شرط زیر می‌رود:

الف- ضریب تلاش، تابع درهم‌ساز رمزنگاری برای هر چه سخت‌تر کردن وارونگی جستجوی کامل رایانشا پرهزینه است.
ب- حالت گیرافتادگی، تابع درهم‌ساز رمزنگاری نباید برای یک الگوی ورودی محتمل دارای حالتی مخصوص بدان را دارا باشد.
ج- پراکنش یا انتشار، هر بیت خروجی تابع درهم‌ساز رمزنگاری باید تابعی از تمامی بیت‌های ورودی و با نسبت پیچیدگی یکسان از هر بیت ورودی باشد.

از نظر تئوری، توابع رمزنگاری را بر اساس استفاده از کلید مخفی می‌توان به توابع درهم‌ساز کلیددار و توابع درهم‌ساز بدون کلید تقسیم کرد. داده‌ساختارهای احتمالاتی فقط از توابع درهم‌ساز بدون کلید استفاده می‌کنند، که شامل توابع درهم‌ساز یک‌طرفه، توابع درهم‌ساز مقاوم در برابر تصادم، و توابع درهم‌ساز جامع یک‌طرفه هستند. این توابع صرفا در برخی ویژگی‌ها با یکدیگر تفاوت دارند. توابع درهم‌ساز یک‌طرفه شرایطی را برآورده می‌کنند. الف- آنها می‌توانند برای بلوک‌های داده با هر طولی اعمال شوند (البته، در عمل، محدود به مقدار ثابتی بسیار بزرگ). ب- همچنین، خروجی با طول ثابت تولید می‌کنند. ج- چنین توابعی باید «مقاومت پیش‌تصویر» (خاصیت یک‌طرفگی) یا یافتن ورودی که به خروجی مشخص شده درهم‌ساز شود باید از نظر محاسباتی غیرقابل اجرا باشد را رعایت کنند.

علاوه بر این شرایط، تولید مقدار درهم‌ساز برابر برای دو ورودی متفاوت در توابع درهم‌ساز مقاوم به تصادم باید بسیار بعید باشد، اگر توابع مذکور مقاوم به تصادم نباشند. در عوض باید توابع درهم‌ساز جهانی یک‌طرفه باید «مقاوم به تصادمِ هدف» یا «مقاوم به تصادمِ پیش‌تصویر دوم» باشند. موارد مذکور بدین معناست که یافتن ورودی دوم متمایز که دارای مقدار درهمی برابر با ورودی مشخص باشد رایانشا نشدنی باشد. باید توجه داشت که مقاوم بودن به تصادم برخورد به معنای مقاوم بودن به پیش‌تصویر دوم است، اما پیچیدگی عمومی یافتن یک تابع مقاوم به پیش‌تصویر دوم بسیار بیشتر از یافتن یک جفت دارای تصادم است.

به دلیل طراحی توابع درهم‌ساز رمزنگاری (به ویژه، شرط ضریب تلاش)، بسیار کندتر از توابع غیر رمزنگاری هستند. مثلا، تابع SHA-۱، که در ادامه بحث قرار می‌گیرد، در حدود ۵۴۰ مگابایت بر ثانیه است، اما توابع غیررمزنگاری در حدود ۲۵۰۰ مگابایت بر ثانیه و بیشتر هستند.

الگوریتم‌های چکیده پیام

رونالد ریویست در سال ۱۳۷۰ الگوریتم چکیده پیام $MD5$ را معرفی کرد که به عنوان جانشین الگوریتم قدیمی‌تر $MD4$ طراحی شده بود. این الگوریتم تابع درهم‌ساز رمزنگاری است که در سند ۱۳۲۱ RFC از سوی IETF تعریف شده است. الگوریتم $MD5$ پیامی با طول دلخواه را دریافت می‌کند و مقدار درهم ۱۲۸ بیتی تولید می‌کند که نماینده فشرده‌ای از داده ورودی است. هدف از این تابع ایجاد مقداری است که تغییرات کوچک در پیام باعث تغییرات بزرگ و غیرقابل پیش‌بینی در خروجی شود.

الگوریتم $MD5$ بر پایه ساختار مرکل–دامگور طراحی شده است. در این ساختار، پیام ابتدا پدگذاری می‌شود. در پدگذاری، بیت خاتمه افزوده می‌شود، سپس بیت‌های صفر به پیام اضافه می‌شود تا طول آن در پیمانه ۵۱۲ مساوی ۴۴۸ شود و در پایان، طول ۶۴ بیتی پیام اصلی اضافه می‌شود. پیام پدگذاری شده در قالب بلوک‌های ۵۱۲ بیتی تقسیم می‌شود که هر بلوک شامل شانزده کلمه ۳۲ بیتی است.

در مرحلهٔ بعد، مقداردهی اولیه انجام می‌شود. در این مرحله چهار ثبات ۳۲ بیتی با مقادیر ثابت اولیه مقداردهی می‌شوند که حالت داخلی الگوریتم را تشکیل می‌دهند. سپس، هر بلوک پیام به ترتیب وارد تابع فشرده‌سازی می‌شود که دارای چهار دور پردازش است و مجموعاً ۶۴ گام را شامل می‌شود. هر دور، الگوریتم از توابع غیرخطی، عملگرهای منطقی، جمع پیمانه‌ای و چرخش‌های بیتی استفاده می‌کند که نقش آن افزایش آشفتگی و انتشار در حالت داخلی است. خروجی هر مرحله به عنوان ورودی مرحله بعد مورد استفاده قرار می‌گیرد.

در دور اول تابع F استفاده می‌شود و و که به صورت بیت به بیت خروجی تولید می‌کند که هر بیت بر اساس انتخاب شرطی عمل می‌کند

$$F(B,C,D) \;=\; (B \text{ و } C) \text{ یا } (B \text{ نقیض و } D)$$

این تابع به رفتار «انتخابی» شبیه است و وابستگی شدیدی به بیت‌های ورودی ایجاد می‌کند . رفتاری غیرخطی را موجب می‌شود.

در دور دوم تابع G استفاده می‌شود و ساختاری مشابه F دارد اما چیدمان متفاوتی از بیت‌ها را به کار می‌گیرد

$$G(B,C,D) \;=\; (B \text{ و } D) \text{ یا } (C \text{ نقیض و } D)$$

هدف آن تقویت پخش‌شدگی در مرحله میانی است. در دور سوم تابع H استفاده می‌شود

$$H(B,C,D) \;=\; B\ XOR\ C\ XOR\ D$$

این تابع تابعی خطی‌تر است (چرا؟) اما تغییرپذیری و ترکیب‌پذیری بیشتری در بیت‌ها ایجاد می‌کند. در دور چهارم تابع I استفاده می‌شود

$$I(B,C,D) \;=\; C\ XOR\ (B \text{ یا نقیض } D)$$

این تابع رفتار غیرخطی پیچیده‌تری نسبت به H دارد و تغییرات بزرگ در حالت داخلی ایجاد می‌کند

در هر یک از ۶۴ گام، خروجی این توابع به همراه یکی از کلمات بلوک پیام، یک ثابت از پیش تعریف‌شده و چرخش چپ ترکیب می‌شود تا حالت داخلی به‌روزرسانی شود. در پایان پردازش آخرین بلوک، چهار ثبات نهایی با یکدیگر ترکیب می‌شوند و مقدار درهم ۱۲۸ بیتی تولید می‌شود. این مقدار به عنوان چکیده پیام شناخته می‌شود. الگوریتم $MD5$ به طور گسترده برای بررسی یکپارچگی داده‌ها استفاده شده است. در این کاربرد، به جای مقایسه مستقیم داده‌های بزرگ، مقادیر درهم محاسبه شده از فایل‌ها با یکدیگر مقایسه می‌شوند تا مشخص شود آیا داده‌ها تغییر کرده‌اند یا خیر.

با وجود کاربرد گسترده $MD5$ در گذشته، پژوهش‌ها نشان داده‌اند که این الگوریتم در برابر حملات تصادم مقاوم نیست. گزارش امنیتی $VU\#836068$۲ تأیید می‌کند که این الگوریتم در برابر حملات تصادم آسیب‌پذیر است. در چنین حملاتی می‌توان دو پیام متفاوت تولید کرد که مقدار درهم یکسانی داشته باشند. این ضعف می‌تواند به مهاجمان اجازه دهد اسناد، گواهی‌ها یا توکن‌های رمزنگاری جعلی تولید کنند که از نظر سامانه‌های مبتنی بر $MD5$ معتبر به نظر می‌رسند. به همین دلیل، استفاده از MD۵ در کاربردهای امنیتی و رمزنگاری دیگر توصیه نمی‌شود. با این حال، در برخی کاربردهای غیرامنیتی مانند ساختارهای داده احتمالاتی یا بررسی ساده یکپارچگی، هنوز ممکن است مورد استفاده قرار گیرد.

الگوریتم‌های درهم‌ساز امن

الگوریتم‌های درهم‌ساز امن در آژانس امنیت ملی ایالات متحده (NSA) تعریف، و مؤسسهٔ ملی استانداردها و فناوری (NIST) منتشر کرد. اولین الگوریتم از خانواده، به نام SHA-۰، در سال ۱۳۷۲ منتشر شد و به سرعت جای خود را به جانشین خود SHA-۱ داد. SHA-۱ مقدار درهم‌ساز بلندتر ۱۶۰ بیتی (۲۰ بایت) تولید می‌کند، و امنیت آن با رفع نقاط ضعف SHA-۰ افزایش یافته است. SHA-۱ سالیان فراوانی در برنامه‌های مختلف استفاده می‌شد و بیشتر وب‌سایت‌ها با استفاده از الگوریتم‌های مبتنی بر آن امضا می‌شدند. با این حال، در سال ۱۳۸۴ یک ضعف در SHA-۱ کشف شد، و در نتیجه در سال ۱۳۸۹ NIST استفاده از آن را در دولت متوقف کرد و از سال ۱۳۹۰ نیز در اینترنت متوقف شد. مانند MD۵، نقاط ضعف یافت شده بر استفاده از آن به عنوان تابع درهم‌ساز برای ساختارهای داده احتمالاتی تأثیری نداشت. SHA-۲ در سال ۱۳۸۰ منتشر شد و شامل شش تابع درهم‌ساز با اندازه چکیده‌های متمایز SHA-۲۲٤، SHA-۲٥٦، SHA-۳۸٤، SHA-٥۱۲، و جز اینها بود. SHA-۲ از SHA-۱ قوی‌تر است و به ثمر نشینی حملاتی که علیه SHA-۲ انجام می‌شوند با قدرت رایانشی فعلی بسیار نامحتمل است. شبه‌کد SHA-۰ به صورت زیر است.

تابع هش SHA-۰ (طرح کلی سطح‌بالا)

```
function SHA۰(message):
                                                    # پَد کردن پیام طبق قواعد خانواده SHA
    padded_message = sha_pad(message)
                                                    # مقداردهی اولیه‌ی پنج مقدار داخلی مثل SHA − ۱
    H۰= ۰x۶۷۴۵۲۳۰۱
    H۱= ۰xEFCDAB۸۹
    H۲= ۰x۹۸BADCFE
    H۳= ۰x۱۰۳۲۵۴۷۶
    H۴= ۰xC۳D۲E۱F۰


                                                    # پردازش پیام در بلوک‌های ۵۱۲ بیتی و شکستن بلوک به شانزده کلمه‌ی ۳۲ بیتی
    for each ۵۱۲-bit block M:
        for i from ۰ to ۱۵:
            W[i] = M[i]


                                                    # گسترش برنامه‌ی پیام نسخه SHA-۰
        for t from ۱٦ to ۷۹:
```

```
        W[t] = (W[t-۳] XOR W[t-۸] XOR W[t-۱۴] XOR W[t-۱۶])
# توجه: SHA-۰ در این مرحله هیچ چرخشِ بیتی انجام نمی‌دهد


# مقداردهی متغیرهای کاری
    A = H۰, B = H۱, C = H۲, D = H۳, E = H۴


# هشتاد دور محاسبه
    for t from ۰ to ۷۹:
        if ۰ ≤ t ≤ ۱۹:
            f = (B AND C) OR ((NOT B) AND D)
            K = ۰x۵A۸۲۷۹۹۹
        else if ۲۰ ≤ t ≤ ۳۹:
            f = B XOR C XOR D
            K = ۰x۶ED۹EBA۱
        else if ۴۰ ≤ t ≤ ۵۹:
            f = (B AND C) OR (B AND D) OR (C AND D)
            K = ۰x۸F۱BBCDC
        else:
            f = B XOR C XOR D
            K = ۰xCA۶۲C۱D۶

# گام اصلی
        TEMP = (leftrotate(A, ۵) + f + E + W[t] + K) mod ۲^۳۲
        E = D
        D = C
        C = leftrotate(B, ۳۰)
        B = A
        A = TEMP


#به‌روزرسانی مقادیر هش
    H۰ = (H۰ + A) mod ۲^۳۲
    H۱ = (H۱ + B) mod ۲^۳۲
    H۲ = (H۲ + C) mod ۲^۳۲
    H۳ = (H۳ + D) mod ۲^۳۲
    H۴ = (H۴ + E) mod ۲^۳۲


#خروجی نهایی ۱۶۰ بیت است
  return H۰ || H۱ || H۲ || H۳ || H۴
```

تفاوت اصلی $SHA-۰$ با $SHA-۱$ در گام زیر رخ می‌دهد:

```
W[t] = W[t-۳] XOR W[t-۸] XOR W[t-۱۴] XOR W[t-۱۶]
```

مقدار «چرخش به چپ ۱ بیتی» در SHA-۰ وجود ندارد، ولی به SHA-۱ اضافه شد و همین موجب پخش بهتر و امنیت بیشتر شد.

W[t] = rotate(W[t-۳] XOR W[t-۸] XOR W[t-۱۴] XOR W[t-۱۶])

مثال- فرض کنید که پیام جهت رمزگذاری حرف A با کد اسکی A=۰x٤١ باشد و اجرای تک بلوک ۵۱۲ بیتی در SHA-۰ را در ادامه می‌بینیم. SHA-۰ پیام را پَد می‌کند تا به یک بلوک کامل ۵۱۲ بیتی تبدیل شود. در نتیجه، کل پیام پَدشده فقط یک بلوک۵۱۲ بیتی دارد.

۱. ساخت بلوک ۵۱۲ بیتی: به طوری کلی فرایند کلی به صورت زیر است. پیام A با پدگذاری در ابتدا ۰x۴۱ قرار می‌گیرد. سپس، ۰x۸۰ قرار گرفته و پس از آن چندین صفر قرار می‌گیرند. در انتها طول پیام (۸ بیت یا ۰x۰۰۰۰۰۰۰۸) اضافه می‌شود. پس، بلوک ۵۱۲ بیتی را به شکل ۱۶ کلمه ۳۲ بیتی نشان می‌دهیم:

W[۰] = ۰x۴۱۰۰۰۰۸۰
W[۱] = ۰x۰۰۰۰۰۰۰۰
W[۲] = ۰x۰۰۰۰۰۰۰۰
W[۳] = ۰x۰۰۰۰۰۰۰۰
...
W[۱۴] = ۰x۰۰۰۰۰۰۰۰
W[۱۵] = ۰x۰۰۰۰۰۰۰۸

۲. گسترش برنامه‌ی پیام (Message Schedule): SHA-۰ از فرمول زیر استفاده می‌کند:

W[t] = W[t−۳] XOR W[t−۸] XOR W[t−۱۴] XOR W[t−۱۶]

«سه مقدار اول» را به صورت عددی حساب می‌کنیم تا روش کاملاً روشن شود.

محاسبه W[١٦]

W[۱۶] = W[۱۳] XOR W[۸] XOR W[۲] XOR W[۰]

چون همه صفر هستند جز W[۰]:

W[١٦]=۰x ۰۰۰۰۰۰۰۰XOR ۰x ۰۰۰۰۰۰۰۰XOR ۰x ۰۰۰۰۰۰۰۰XOR ۰x ۴۱۰۰۰۰۸۰= ۰x۴۱۰۰۰۰۸۰

W[۱۷] = W[۱۴] XOR W[۹] XOR W[۳] XOR W[۱]

تمام ورودی‌ها صفر هستند:

W[۱۷] = ۰x۰۰۰۰۰۰۰۰

W[۱۸] = W[۱۵] XOR W[۱۰] XOR W[۴] XOR W[۲]

W[۱۵] = ۰x ۰۰۰۰۰۰۰۸

و بقیه صفر:

W[۱۸] = ۰x۰۰۰۰۰۰۰۸

همان‌طور که از مثال برمی‌آید گسترش پیام در SHA-۰ ساختاری نسبتاً ساده دارد.

۳. مقداردهی اولیه متغیرهای کاری

A = ۰x ۶۷۴۵۲۳۰۱

B = ۰xEFCDAB ۸۹

C = ۰x۹۸BADCFE

D = ۰x۱۰۳۲۵۴۷۶

E = ۰xC۳D۲E۱F۰

که همان مقدار اولیه استاندارد SHA-۰ و SHA-۱ است.

۴. اجرای چند دور اول: برای قابل پیگیری ماندن مثال صرفا سه دور اول را از هشتاد دور را نمایش می‌دهیم

دور صفر: در بازه $t = ۰$ تا ۱۹ از تابع f۱ استفاده می‌شود:

f = (B AND C) OR ((NOT B) AND D)

اگر مقداردهی انجام شود:

f ≈ ۰x ۱۰۳۲۵۴۷۶

K = ۰x۵A۸۲۷۹۹۹

W[۰] = ۰x۴۱۰۰۰۰۸۰

محاسبه TEMP:

TEMP = leftrotate(A,۵) + f + E + W[۰] + K

leftrotate(A,۵) ≈ ۰xE۸A۴۶۰۲C

محاسبه جمع:

TEMP ≈ 0xE۸A۴۶۰۲C + ۰x۱۰۳۲۵۴۷۶+ ۰xC۳D۲E۱F۰+۰x۴۱۰۰۰۰۸۰+۰x۵A۸۲۷۹۹۹

= ۰xA۰F۸A ۷۱۱(mod ۲^۳۲)

به‌روزرسانی متغیرها:

A = TEMP = ۰xA۰F۸A۷۱۱

B = قبلی A = ۰x۶۷۴۵۲۳۰۱

C = leftrotate(B,۳۰) = ۰x۵۹D۴۸D۸C

D = قبلی C = ۰x۹۸BADCFE

E = قبلی D = ۰x۱۰۳۲۵۴۷۶

دور ۱: f همان تابع (چون $t = ۱$ هنوز در ۰–۱۹ است)

W[۱] = ۰x ۰۰۰۰۰۰۰۰

K = ۰x۵A۸۲۷۹۹۹

leftrotate(A,۵)
≈۰x۱F۱۴E۲۳۳

TEMP = ۰x۱F۱۴E ۲۳۳ + f(جدید) + E + W[۱] + K ≈۰x۹D۲۲۸۹AB

به‌روزرسانی:

A = ۰x۹D۲۲۸۹AB
B = ۰xA۰F۸A ۷۱۱
C = leftrotate(B,۳۰) = ۰x۲۸۳E۲۹C ۴
D = ۰x۵۹D۴۸D۸C
E = ۰x۹۸BADCFE

دور دوم:

W[۲] = ۰
K = همان مقدار
leftrotate(A,۵) ≈۰xA۴۵۱۳۷۵۷
TEMP ≈ ۰x۸D۳FA۷۶۰

به‌روزرسانی:

A = ۰x۸D۳FA۷۶۰
B = ۰x۹D۲۲۸۹AB
C = ۰x۲۸۳EA۹C۲
D = ۰x۲۸۳E۲۹C٤
E = ۰x۵۹D۴۸D۸C

تا اینجا سه دور اول از ۸۰ دور انجام شده است. روند مزبور تا دور ۷۹ ادامه می‌یابد. در پایان، مقادیر A,B,C,D,E به مقدارهای اولیه افزوده می‌شوند تا خروجی نهایی بلوک تولید شود.

شبه‌کد آن به صورت زیر است

الگوریتم SHA-۲
ورودی: پیام M
خروجی: هش(M)SHA-۲

۱. پدگذاری پیام با یک بیت ۱ و چند بیت ۰ به اندازه مناسب تا طول آن برابر با k
۲. تقسیم پیام پد شده به بلاک‌هایی با طول ثابت
۳. تنظیم مقدار اولیه درهم‌ها
۴ .برای هر بلاک:
الف. تعیین مقادیر اولیه word
ب. ۶۴ دور (round) اجرا می‌شود:
- محاسبه تابع چرخش و انتخاب(choose, majority)

- به‌روزرسانی مقادیر word

ج. به‌روزرسانی مقدار درهم‌ها با نتایج دورها

۵ .خروجی، ترکیب نهایی مقدار درهم

رادیو قفل گاتن RadioGatún

خانواده تابع درهم‌ساز رمزنگاری به نام RadioGatún را گویدو برتونی، خوان دائمون، میکائل پیترز، و ژیل ون آسخ در سال ۱۳۸۵ معرفی کردند که بر اساس بهبود تابع درهم‌ساز پاناما بود. رادیو قفل گاتن خانواده‌ای از ۶۴ تابع درهم متفاوت است که تمایز آنها در طول کلمه به بیت از ۱ تا ۶۴ است. اما صرفا دو بردار ۳۲ و ۶۴ بیتی استفاده شدند. همانند سایر توابع درهم‌ساز، ورودی به دنباله‌ای از بلوک‌ها تقسیم می‌شود که با استفاده از تابعی خاص به حالت داخلی الگوریتم تزریق می‌شوند و سپس با اعمال تکراری یک تابع دور غیر رمزنگاری (به نام تابع دور تسمه و آسیاب) دنبال می‌شود. در هر دور، حالت به عنوان دو بخش، تسمه و آسیاب، نمایش داده می‌شود که توسط تابع دور به طور متفاوتی پردازش می‌شوند. اعمال تابع دور شامل چهار عملیات به صورت موازی است: ۱) تابع غیرخطی اعمال شده بر آسیاب، ۲) تابع خطی ساده اعمال شده بر تسمه، ۳) بازخورد برخی از بیت‌های آسیاب به تسمه به صورت خطی، ۴) بازخورد برخی از بیت‌های تسمه به آسیاب به صورت خطی. پس از تزریق تمام بلوک‌های ورودی، الگوریتم تعدادی دور بدون ورودی یا خروجی (دورهای خالی) انجام می‌دهد. پس از آن بخشی از حالت به عنوان مقدار درهم‌ساز نهایی برگردانده می‌شود. در میان خانواده، RadioGatún٦٤ با کلمات ۶۴ بیتی، انتخاب پیش‌فرض است و برای بسترهای۴ بیتی بهینه است. برای بهترین عملکرد در پلتفرم‌های ۳۲ بیتی، می‌توان از RadioGatún٣٢ با کلمات ۳۲ بیتی نیز استفاده کرد. برای همان فرکانس ساعت، ادعا می‌شود که RadioGatún٣٢ برای ورودی‌های طولانی ۱۲ برابر سریع‌تر از SHA-٢٥٦ و برای ورودی‌های کوتاه ۳.۲ برابر سریع‌تر است، در حالی که گیت‌های کمتری دارد. RadioGatún٦٤ حتی برای ورودی‌های طولانی ۲۴ برابر سریع‌تر از SHA-٢٥٦ است اما حدود پنجاه درصد گیت منطقی بیشتری دارد.

توابع درهم‌ساز غیررمزنگاری

توابع غیررمزنگاری، برعکس توابع درهم‌ساز رمزنگاری، برای دفع حملات با هدف یافتن تصادم طراحی نشده‌اند، بنابراین به امنیت و مقاومت بالا در برابر تصادم نیاز ندارند. چنین توابعی صرفاً باید سریع باشند و احتمال پایین تصادم را تضمین کنند و اجازه دهند مقدار زیادی داده به سرعت با احتمال خطای معقول درهم‌ساز شود.

فاولر-نول-وو Fowler/Noll/Vo

الگوریتم درهم‌ساز غیررمزنگاریFowler/Noll/Vo (مشهور به FNV یا FNV١) نخستین بار در سال ۱۳۷۰ خورشیدی توسط گلن فاولر و فونگ وو به کمیته IEEE POSIX P١٠٠٣,٢ ارائه شد. پس از آن، لندون کرت نول با اعمال بهبودهایی در ساختار آن، نسخه نهایی این الگوریتم را معرفی کرد [Fo١٨]. الگوریتم FNV در جدول‌های درهم، اثرانگشتی‌سازی داده‌ها، نمایه‌گذاری پرونده‌ها و کاربردهای درهم‌سازی سبک (غیررمزنگاری) به کار می‌رود. هدف اصلی طراحی این الگوریتم، دستیابی به عملیاتی بسیار کوچک و سریع، تولید توزیعی مناسب، سادگی پیاده‌سازی در هر زبان برنامه‌نویسی، و قابلیت پردازش جریانی ورودی به صورت بایت‌به‌بایت است.

الگوریتم FNV حالتی داخلی را نگهداری می‌کند که در ابتدا با مقدار ثابتی به نام پایهٔ افست (offset basis) مقداردهی اولیه می‌شود. سپس، الگوریتم به ازای هر بایت از ورودی، عملیات زیر را به ترتیب انجام می‌دهد: ضرب کردن حالت جاری در ثابت عددی بزرگ

به نام عدد اول FNV (FNV Prime)، و سپس اعمال عملگر XOR منطقی میان حاصل و بایت ورودی. پس از پردازش آخرین بایت، مقدار نهایی حالت به عنوان مقدار درهم (درهم‌ساز) گزارش می‌شود. عدد اول FNV و پایهٔ افست، پارامترهای طراحی هستند که به طول بیتی مقدار درهم تولیدی (مانند ۳۲، ۶۴، ۱۲۸ بیت و غیره) وابستگی دارند. به گفتهٔ لندون کرت نول، انتخاب عدد اول مناسب نقشی تعیین‌کننده در کارایی درهم‌سازی دارد. به گونه‌ای که حتی برای اندازه درهم یکسان، برخی اعداد اول نتایج بهتری نسبت به دیگران ارائه می‌دهند.

نسخه بهبودیافته FNV۱a تغییری جزئی در ترتیب عملیات دارد. در این نسخه، ابتدا عملگر XOR روی حالت و بایت ورودی اعمال می‌شود و سپس حاصل در عدد اول FNV ضرب می‌گردد. با وجود استفادهٔ FNV۱a از همان پارامترهای اولیه و عدد اول FNV مانند FNV۱ استفاده می‌کند، این جابه‌جایی ساده بدون آنکه تأثیر منفی بر عملکرد پردازنده داشته باشد موجب پراکندگی (توزیع) اندکی بهتر بیت‌های خروجی می‌شود. در حال حاضر، خانواده FNV شامل الگوریتم‌هایی برای مقادیر درهم ۳۲، ۶۴، ۱۲۸، ۲۵۶، ۵۱۲ و ۱۰۲۴ بیتی است.

الگوریتم FNV با وجود سادگی بسیار زیاد در پیاده‌سازی، برای رشته‌های تقریباً یکسان عملکرد مناسبی دارد، اما خروجی آن نسبتاً «تنک» (sparse) است (بسیاری از بیت‌ها صفر هستند) و این ضعف، آن را برای کاربردهای رمزنگاری نامناسب می‌سازد. این الگوریتم به طور گسترده در سرورهای DNS، توییتر، درهم‌سازهای نمایه پایگاه داده، موتورهای جستجوی وب و بسیاری سامانه‌های دیگر استفاده می‌شود. همچنین چند سال پیش نسخه ۳۲ بیتی FNV۱a به عنوان الگوریتم درهم‌ساز پیشنهادی برای تولید برچسب جریان IPv۶ توصیه شد[An۱۲].

همان‌طور که اشاره شد، فـن‌و بر سه مؤلفهٔ اصلی استوار است:

۱. پایهٔ افست Offset basis: عددی ثابت و بزرگ که حالت اولیهٔ درهم را تعیین می‌کند. به عنوان مثال (برای نسخه‌های استاندارد): نسخهٔ ۳۲ بیتی ۰x۸۱۱C۹DC۵ و نسخه ۶۴ بیتی ۰xCBF۲۹CE۴۸۴۲۲۲۳۲۵ است.
۲. عدد اول FNV prime: عدد اول بزرگ که برای پخش کردن (mixing) بیت‌ها استفاده می‌شود. در نسخهٔ ۳۲ بیتی ۰□۰۱۰۰۰۱۹۳ و در نسخهٔ ۶۴ بیتی ۰□۱۰۰۰۰۰۰۱□۳ است.
۳. دو عمل ساده برای هر بایت ورودی اعم از یای انحصاری و ضرب در عدد اول است و در پایان همیشه عملیات به پیمانهٔ ۲ به توان n (مثلاً $۲^{۳۲}$ یا $۲^{۶۴}$) نگه داشته می‌شود. ضرب در عدد اول بزرگ موجب پخش مؤثر بیت‌ها و جلوگیری از ایجاد الگوهای تکراری می‌شود. عملگر XOR انحصاری نیز بایت ورودی را مستقیماً با حالت فعلی ترکیب می‌کند. ترکیب این دو عملیات، درهمی بسیار سریع و سبک با توزیعی مناسب (و نه رمزنگاری ایمن) تولید می‌کند.

همین سادگی دلیل سرعت بالای FNV است. در ادامه الگوریتم اصلی و بهبودها را معرفی می‌کنیم.

الگوریتم فـن‌و-۱

برای هر بایت ورودی b:

```
hash = offset_basis

for each byte b:
    hash = hash * FNV_prime
    hash = hash XOR b
```

در پایان، مقدار درهم hash نتیجهٔ نهایی است.

الگوریتم فـن‌و-۱الف

نسخه مزبور رایج‌تر و مناسب‌تر است زیرا این نسخه ترتیب عملیات را تغییر می‌دهد و کیفیت پراکندگی را بیشتر می‌کند:

```
hash = offset_basis

for each byte b:
    hash = hash XOR b
    hash = hash * FNV_prime
```

برتری فـن‌و-۱الف در «رفتار بهمنی» (Avalanche effect) بهتر و تصادم‌های کمتر نسبت به نسخهٔ اولیه است. دلیل رفتار بهمنی بهتر در فـن‌و-۱الف نسبت به فـن‌و-۱ این است که در فـن‌و-۱الف عمل XOR با بایت ورودی پیش از ضرب انجام می‌شود. بنابراین، تأثیر بایت جدید پیش از پخش شدن در ضرب، کاملاً در حالت داخلی ترکیب می‌گردد، در حالی که در فـن‌و-۱ ضرب پیش از XOR رخ می‌دهد و بایت ورودی بدون پخش کافی به حالت اضافه می‌شود.

ضرب در عدد اول بزرگ باعث می‌شود بیت‌ها به شکل مؤثری پخش شوند و از ایجاد الگوهای تکراری جلوگیری کند. پای انحصاری نیز بایت ورودی را به طور مستقیم با حالت فعلی ترکیب می‌کند. ترکیب این دو منجر به تولید درهمی سبک و بسیار سریع ولی با توزیع مناسب می‌شود.

مثال، درهم رشتهٔ A با استفاده از فـن‌و-۱الف نسخهٔ ۳۲ بیتی را حساب می‌کنیم. پایه آفست برابر مقدار زیر است.

Hash= ۰x۸۱۱C۹DC۵

گام ۱: عمل XOR با بایت ۰x۴۱

hash = ۰x۸۱۱C۹DC۵XOR ۰x۴۱= ۰x۸۱۱C۹D۸۴

گام ۲: ضرب در FNV prime

hash = ۰x۸۱۱C۹D۸۴ * ۱۶۷۷۷۶۱۹(% $۲^{۳۲}$) = ۰xE۴۰C۲۹۲C

درهم نهایی A:

۰xE۴۰C۲۹۲C

هر چند فـن‌و برای درهم رمزنگاری و گذرواژه یا امنیت مناسب نیست، اما برای کاربردهائی از قبیل جدول‌های درهم، مقایسهٔ سریع فایل‌ها (checksums)، کلیدهای درهم در پروتکل‌های شبکه، حذف افزونگی، اثرانگشت‌های ۶۴ بیتی، و تولید بذر اولیهٔ درهم‌های غلطان یا سریع مناسب است.

تمرین-

فـن‌و-۱ را فـن‌و-۱الف مقایسه کنید.

نویسندگان اشاره به اهمیت انتخاب عدد اول داشتند. چرا انتخاب عدد اول در درهم نهایی اهمیت دارد؟ نحوهٔ انتخاب عدد اول را روشن کنید.

وقتی فایل بیش از یک بایت است چه اتفاقی می‌افتد و چگونه کار می‌کند. آیا درهم حاصل موقعیت وابسته است؟ مثلا ABC یا CBA یک پاسخ دارد؟ چرا بایت‌های اولیه دارای وزن بیشتری در درهم نهایی هستند؟

الگوریتم درهم مورمور

روش‌های درهم مورمور MurmurHash خانواده‌ای از توابع درهم‌ساز غیررمزنگاری است که آستین اپلبی (Austin Appleby) در سال ۱۳۸۶ خورشیدی منتشر کرد. نسخه نهایی، $MurmurHash۳$، در سال ۱۳۹۰ معرفی گردید [۱۱Ap].
هدف اصلی درهم مورمور سرعت بسیار بالا همراه با توزیع عالی (اثر بهمنی خوب) است. این الگوریتم برای کاربردهایی مانند جدول‌های درهم، فیلترهای بلوم، و کش‌های کلید-مقدار طراحی شده است، و برای اهداف رمزنگاری مناسب نیست.
از ویژگی‌های کلیدی آن سرعت است. ادعا می‌شود بیش از دو برابر سریع‌تر از تابع $lookup۳$ (که بهینه‌سازی شده برای سرعت) است. دارای نسخه‌هائی شامل $MurmurHash۳$ برای ۳۲ بیت $(x۸۶)$، ۶۴ بیت $(x۶۴)$ و ۱۲۸ بیت است. همچنین، در بررسی کیفیت توزیع، آزمون‌های آماری را به خوبی پشت سر می‌گذارد.

در نسخهٔ $MurmurHash۳$ برای ۳۲ بیت $(x۸۶)$، ورودی به صورت بلوک‌های ۴ بایتی (۳۲ بیتی) پردازش می‌شود. اگر طول ورودی مضرب ۴ نباشد، بایت‌های باقی‌مانده جداگانه پردازش می‌شوند.
دو ثابت طراحی که به صورت تجربی برای آمیختگی (mixing) بیت‌ها انتخاب شده‌اند عبارتند از مقادیر زیر هستند.

c۱= ۰xcc۹e۲d۵۱

c۲= ۰x۱b۸۷۳۵۹۳

این اعداد به گونه‌ای انتخاب شده‌اند که پخش‌شدگی (اثر بهمن) بهینه داشته باشند.

تابع کمکی یا چرخش به چپ به اندازه r بیت به این معناست که بیت‌ها به چپ حرکت می‌کنند و بیت‌های بیرون زده از سمت چپ، به سمت راست وارد می‌شوند.

مثال- در ۳۲ بیت زیر داریم:

rotl۳۲(۰x۱۲۳۴۵۶۷۸, ۴) = ۰x۲۳۴۵۶۷۸۱

در عمل، این کار با سه شیفت و یک OR پیاده‌سازی می‌شود.

مراحل مفهومی الگوریتم به شرح زیر است. در مرحلهٔ نخست مقداردهی اولیه صورت می‌پذیرد و مقدار h (حالت داخلی) برابر با دانه اولیه که کاربر مشخص می‌کند قرار می‌گیرد. در مرحلهٔ دوم بلوک‌های ۴ بایتی پردازش می‌شود. برای هر بلوک ۳۲ بیتی از ورودی داریم:

۱. k برابر مقدار بلوک جاری است که به صورت little-endian از بایت‌ها خوانده می‌شود.
۲. مقدار k در ثابت نخست ضرب می‌شود: k = k * c۱.
۳. مقدار حاصل ۱۵ بیت به چپ چرخش داده می‌شود: k = rotl۳۲(k, ۱۵).
٤. حاصل در ثابت دوم ضرب می‌شود: k = k * c۲.
۵. حاصل با حالت ترکیب می‌شود: h = h xor k.
٦. حالت چرخش می‌یابد: h = rot۳۲(h, ۱۳)
۷. اختلاط دیگر اجرا می‌شود: h = h * ۵ + ۰xe٦۵٤٦b٦٤.

در مرحلهٔ سوم بایت‌های باقی‌مانده (tail) پردازش می‌شوند. اگر طول ورودی مضرب ۴ نباشد، بین ۱ تا ۳ بایت باقی می‌ماند. این بایت‌ها به صورت زیر در متغیر ۳۲ بیتی k قرار می‌گیرند (بایت اول در کم‌ارزش‌ترین بیت). در ابتدا k با صفر به صورت زیر مقداردهی می‌شود و سپس بایت‌های باقیمانده به صورت زیر با آن ترکیب می‌شوند:

k=۰

اگر بایت سوم وجود دارد: k = k XOR(B[۲]<<۱٦). به دیگر سخن، ابتدا مقدار بایت سوم را شانزده بیت به سمت چپ جابجا کن، یا شانزده صفر در مقابل آن قرار بده و یا معادل در ۲^۱۶ ضرب کن. سپس، عملیات یای انحصاری را روی حاصل و k اعمال و در k قرار بده.

اگر بایت دوم وجود دارد: k = k XOR(B[۱]<<۸).

اگر بایت اول وجود دارد: k = k XOR(B[۰]).

سپس، k حاصل مانند بلوک کامل (مراحل دو-۱ تا دو-۴) پردازش شده و با h ترکیب می‌شود.

در مرحلهٔ چهارم برای جلوگیری از تصادم بین پیام‌های با طول متفاوت که محتوای یکسانی دارند طول پیام افزوده می‌شود:

h = h XOR length

مرحلهٔ پنجم نهایی‌سازی و استفاده از تابعی خاص مشهور به تابع $fmix۳۲$ است. هدف این مرحله از بین بردن هرگونه الگوی خطی باقی‌مانده و اطمینان از این است که هر بیت از ورودی روی همه بیت‌های خروجی تأثیر بگذارد.

h = h XOR (h >> ۱۶)

h = h * ۰x۸۵ebca۶b

h = h XOR (h >> ۱۳)

h = h * ۰xc۲b۲ae۳۵

h = h XOR (h >> ۱۶)

در مرحلهٔ ششم مقدار h به عنوان درهم‌ساز نهایی برگردانده می‌شود.

شبه‌کد الگوریتم به صورت زیر است.

```
def rotl۳۲(x, r):
    return ((x << r) & ۰xFFFFFFFF) | (x >> ( ۳۲- r))

def MurmurHash۳_x۸۶_۳۲(key, length, seed):
    c ۱= ۰xcc۹e۲d۵۱
    c ۲= ۰x۱b۸۷۳۵۹۳
    h = seed

                                    # پردازش بلوک‌های ۴ بایتی
    nblocks = length // ۴
    for i in range(nblocks):
                                    # خواندن ۴ بایت به صورت little-endian (فرض می‌شود key یک آرایه بایت است)
        k = (key[۴*i] & ۰xFF) | ((key[۴*i+۱] & ۰xFF) << ۸)\ |
            ((key[۴*i+۲] & ۰xFF) << ۱۶) | ((key[۴*i+۳] & ۰xFF) << ۲۴)

        k = (k * c۱) & ۰xFFFFFFFF
        k = rotl۳۲(k, ۱۵)
        k = (k * c۲) & ۰xFFFFFFFF

        h = h ^ k
        h = rotl۳۲(h, ۱۳)
        h = (h *  ۵+ ۰xe۶۵۴۶b۶۴) & ۰xFFFFFFFF
```

```
    # پردازش بایت‌های باقی‌مانده tail
    tail = key[nblocks* ۴: length]
    k = ۰
    if length %  ۴>= :۳
        k = k ^ (tail[۲] << ۱۶)
    if length %  ۴>= :۲
        k = k ^ (tail[۱] << ۸)
    if length %  ۴>= :۱
        k = k ^ (tail[۰])

    if k != :۰
        k = (k * c۱) & ۰xFFFFFFFF
        k = rotl۳۲(k, ۱۵)
        k = (k * c۲) & ۰xFFFFFFFF
        h = h ^ k

    # افزودن طول
    h = h ^ length

    # نهایی‌سازی (fmix۳۲)
    h = h ^ (h >> ۱۶)
    h = (h * ۰x۸۵ebca۶b) & ۰xFFFFFFFF
    h = h ^ (h >> ۱۳)
    h = (h * ۰xc۲b۲ae۳۵) & ۰xFFFFFFFF
    h = h ^ (h >> ۱۶)

    return h & ۰xFFFFFFFF
```

ترتیب خواندن بایت‌ها (Endianness) اهمیت فراوان دارد. در پیاده‌سازی واقعی، بایت‌های هر بلوک به صورت little-endian (کم‌ارزش‌ترین بایت در آدرس پایین‌تر) خوانده می‌شوند. این فرض در معماری‌های $x۸۶/x۶۴$ برقرار است. اگر الگوریتم روی معماری big-endian اجرا شود، باید بایت‌ها را جابجا کرد.

تمرین

- نسخه ۱۲۸ بیتی $MurmurHash۳\_x۶۴\_۱۲۸$ را بیابید، بنویسید و تحلیل کنید.
- تفاوت $MurmurHash۲$ و $MurmurHash۳$ را توضیح دهید.

مثال- با توجه به الگوریتم تعریف شده مثالی عددی و مرحله‌به‌مرحله از اجرای $MurmurHash۳$ (نسخه ۳۲ بیتی) را بررسی می‌کنیم تا درک بهتری از نحوه اجرای الگوریتم ایجاد شود. جهت سادگی از ورودی کوچکی abc استفاده می‌کنیم که در قالب بایت‌ها و بر اساس کد اسکی‌در قالب شانزده شانزدهی ۶۱ ۶۲ ۶۳ است. دانه یا مقدار اولیه را برابر صفر می‌گیریم در مرحلهٔ ۱ ورودی را بلوک‌های۴ بایتی تبدیل می‌کنیم. چون طول پیام ۳ بایت است، پس بلوک کاملی وجود ندارد و مستقیما وارد مرحلهٔ ذیل می‌شویم.
پس مرحلهٔ دوم مرحلهٔ پردازش ذیل است که

tail =[۶۱,۶۲,۶۳]

به ترتیب:

k = ۰

k ^= ۶۳<< ۱۶

k ^= ۶۲<< ۸

k ^= ۶۱

محاسبه

k =

۶۵۵۳۶ * ۶۳= ۴۱۲۸۷۶۸

۲۵۶ * ۶۲= ۱۵۸۷۲

۶۱

پس

k = ۴۱۲۸۷۶۸+ ۱۵۸۷۲+ ۶۱

عدد دقیق به هگز

k = ۰x۰۰۳e۳d۳d

حال وارد مراحل الگوریتم می‌شویم

k = k * c۱

c۱= ۰xcc۹e۲d۵۱

محاسبه

k = ۰x۰۰۳e۳d۳d × ۰xcc۹e۲d ۵۱

نتیجه مقدار بزرگ ۶۴ بیتی می‌شود ولی در نهایت فقط ۳۲ بیت (ضرب پیمانه ۳۲ بیت) نگه داشته می‌شود
سپس، بر k چرخش چپ ۱۵ بیتی اعمال می‌شود:

k = rotl۳۲(k, ۱۵)

در ادامه،

k = k * c ۲

c۲= ۰x۱b۸۷۳۵۹۳

در نهایت،

h = seed XOR k

چون $seed = ۰$، پس

h = k

مرحله ۳، افزودن طول پیام و چون طول = ۳، پس،

h = h XOR ۳

مرحله ۴ مرحله نهایی‌سازی است و مراحل $fmix۳۲$ روی h اعمال می‌شود که شامل

h ^= h >> ۱۶

h *= ۰x۸۵ebca۶b

h ^= h >> ۱۳

h *= ۰xc۲b۲ae ۳۵

h ^= h >> ۱۶

پس از این محاسبات، خروجی نهایی به دست می‌آید و خروجی نهایی $MurmurHash۳("abc")$ مقدار $۰xb۳dd۹۳fa$ و به صورت دهدهی ۳۰۱۷۶۴۵۰۵۰

در حال حاضر، MurmurHash۳ از الگوریتم‌های پر استفاده است و در Apache Hadoop، Apache Cassandra، Elasticsearch، libstdc++، nginx و جز اینها استفاده می‌شود.

CityHash و FarmHash

در سال ۱۳۹۰، گوگل خانواده‌ای جدید از توابع درهم‌ساز برای رشته‌ها به نام CityHash را منتشر کرد که جف پایک و جیرکی آلکویالا تعریف کرده بودند [Pi۱۱]. توابع CityHash توابع درهم‌ساز غیر رمزنگاری ساده‌ای بر اساس الگوریتم MurmurHash۲ هستند.

خانواده CityHash با تمرکز بر رشته‌های کوتاه (مثلاً تا ۶۴ بایت) که بیشترین میزان توجه را در داده‌ساختارهای احتمالاتی و جدول‌های درهم‌ساز دارند، توسعه یافتند و شامل نسخه‌های ۳۲، ۶۴، ۱۲۸ و ۲۵۶ بیتی است. برای چنین رشته‌های کوتاهی، نسخه ۶۴ بیتی CityHash٦٤ سریع‌تر از MurmurHash است و از CityHash۱۲۸ با اندازهٔ ۱۲۸ بیتی بهتر عمل می‌کند. با این حال، در حالی که برای رشته‌های طولانی با حداقل چند صد بایت، CityHash۱۲۸ نسبت به سایر توابع درهم‌ساز خانواده CityHash ترجیح داده می‌شود، در عمل توصیه بر استفاده از MurmurHash۳ است. از معایب CityHash پیچیدگی نسبی آن است که منجر به رفتار غیربهینه در کامپایلرهای مختلف می‌شود و می‌تواند سرعت آن را تا حد قابل توجهی کاهش دهد.

در سال ۱۳۹۳ گوگل جانشین CityHash به نام FarmHash را منتشر کرد که جف پایک تعریف کرده بود [Pi۱٤]. الگوریتم جدید شامل بیشتر تکنیک‌های استفاده شده در CityHash (اعم از پیچیدگی آن) و نسل جدید MurmurHash بود. توابع FarmHash بیت‌های ورودی را به طور کامل مخلوط می‌کنند، اما برای استفاده در رمزنگاری کافی نیست. FarmHash از بهینه‌سازی‌های خاص CPU استفاده می‌کند و هنوز برای دستیابی به بهترین عملکرد نیاز به تنظیم کامپایلر دارد و به بستر وابسته است. لازم به ذکر است، مقادیر درهم‌ساز محاسبه شده در بسترهای مختلف متفاوت هستند. توابع FarmHash در نسخه‌های متعددی عرضه می‌شوند و نسخه ۶۴ بیتی Farm٦٤ در آزمون‌های روی بسیاری از بسترها، از جمله تلفن‌های همراه، از الگوریتم‌هایی مانند CityHash، MurmurHash۳، و FNV بهتر عمل می‌کند.

جدول‌های درهم

جدول درهم داده‌ساختاری لغت‌نامه‌ای است که از «آرایه‌ای انجمنی» نامرتب به طول m تشکیل شده است که ورودی‌های آن سطل نامیده می‌شوند و با کلیدی در محدوده $\{0,1,\dots,m-1\}$ اندیس می‌شوند. برای درج مقداری در جدول درهم، از تابع درهم‌ساز برای محاسبه کلیدی استفاده می‌شود که برای انتخاب سطل مناسب برای ذخیره مقدار به کار می‌رود. معمولا دامنه‌ای که عناصر ورودی از آن گرفته می‌شوند بسیار بزرگتر از ظرفیت m جدول درهم است، بنابراین تصادم در کلیدها اجتناب‌ناپذیر است. علاوه بر این، وقتی تعداد عناصر در جدول درهم‌ساز افزایش می‌یابد، تعداد تصادم‌ها نیز افزایش می‌یابد. ضریب بار α مفهوم بنیادی جدول‌های درهم است که از نسبت تعداد کلیدهای استفاده شده n به طول کل جدول m به دست می‌آید:

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

ضریب بار معیاری از میزان پر بودن جدول درهم است و چون $n \leq m$، کران بالای آن یک است. وقتی α به مقدار حداکثر خود میل می‌کند، احتمال تصادم افزایش قابل توجهی می‌کند که می‌تواند نیاز به افزایش ظرفیت را ایجاد کند. همه پیاده‌سازی‌های جدول درهم‌ساز باید مشکل تصادم‌ها را تدبیر کنند و راهکاری برای مدیریت آنها پیشنهاد دهند. دو تکنیک اصلی در این زمینه شامل آدرس‌دهی بسته و آدرس‌دهی باز است. در آدرس‌دهی بسته عناصر تصادمی در ذیل همان کلید در داده‌ساختاری ثانویه ذخیره می‌شوند. در آدرس‌دهی باز عناصر تصادمی در موقعیت‌هایی غیر از موقعیت‌های ترجیحی خود با ارائه راهی برای آدرس‌دهی آنها ذخیره می‌شوند.

تکنیک آدرس‌دهی بسته دم دست‌ترین راه حل تصادم‌ها است و دارای پیاده‌سازی‌های مختلفی است. مثلا، زنجیره‌سازی جداگانه که عناصر تصادمی کرده را در فهرستی پیوندی ذخیره می‌کند، درهم‌ساز کامل که از توابع درهم‌ساز خاص و جدول‌های درهم‌ساز ثانویه با طول‌های مختلف استفاده می‌کند.

به جای ایجاد داده‌ساختاری ثانویه به هر شکلی، می‌توان تصادم‌ها را با ذخیره عناصر تصادمی در جای دیگر جدول اصلی و ارائه الگوریتمی برای آدرس‌دهی آنها حل کرد. از آنجایی که آدرس عنصر از ابتدا مشخص نیست، این تکنیک به عنوان آدرس‌دهی باز شناخته می‌شود. «کاوش خطی» و «فاخته» دو پیاده‌سازی آدرس‌سازی باز است که در ادامه توضیح می‌دهیم.

کاوش خطی

از پیاده‌سازی‌های ساده جدول درهم‌ساز با آدرس‌دهی باز، الگوریتم کاوش خطی است که در سال ۱۳۳۳ جین آم دال، الین ام. مک‌گرا، و آرتور ساموئل اختراع کردند و دونالد کنوث آن را در سال۱۳۴۲ تحلیل کرد.

تمرین- تحلیل کنوث را گزارش کنید.

الگوریتم مقدار تصادمی را در مدخل خالی بعدی قرار می‌دهد. نام آن از این واقعیت ناشی می‌شود که موقعیت نهایی عنصر به صورت خطی از مدخل ترجیحی جابه‌جا می‌شود زیرا مدخلی پس از دیگری را کاوش می‌کند. جدول درهم‌ساز کاوش خطی را می‌توان به صورت آرایهٔ دواری دید که مقادیر اندیس شده را در سطل‌ها ذخیره می‌کند. برای درج مقدار جدید x، کلید آن را $k = h(x)$ با استفاده از تابع درهم‌ساز واحد h محاسبه می‌شود. اگر سطلی که با آن کلید مطابقت دارد پر باشد و حاوی مقدار متفاوتی باشد، به معنای تصادم است، به جستجو در جهت عقربه‌های ساعت در سطل‌های بعدی ادامه می‌دهیم تا جایی که فضای خالی پیدا کنیم که بتوانیم عنصر x را در آن اندیس کنیم. نظارت بر ضریب بار جدول درهم‌ساز می‌تواند تضمین کند که قطعاً در نقطه‌ای فضای خالی پیدا خواهیم کرد.

به طور مشابه، وقتی می‌خواهیم عنصری x را جستجو کنیم، کلید آن k را با استفاده از همان تابع درهم‌ساز h محاسبه می‌کنیم و شروع به بررسی سطل‌ها در جهت عقربه‌های ساعت می‌کنیم، از سطل ترجیحی با کلید $k = h(x)$ شروع می‌کنیم، تا زمانی که عنصر مورد نظر x را پیدا کنیم یا اولین سطل خالی ظاهر شود، که نتیجه می‌دهد عنصر در جدول نیست.

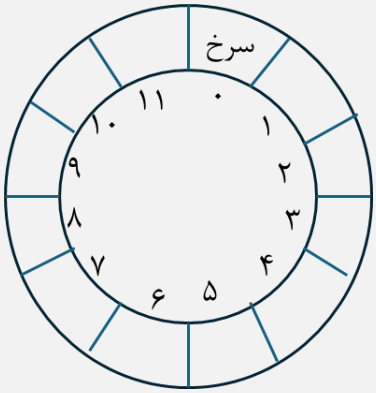
مثال- کاوش خطی، جدول درهم‌ساز کاوش خطی به طول $m = ۱۲$ و تابع درهم‌ساز مبتنی بر MurmurHash۳ ۳۲ بیتی که دامنه را به محدوده $\{۰,۱, \dots, m - ۱\}$ نگاشت می‌دهد را در نظر می‌گیریم

$$h(x) = MurmurHash3(x) \,\%\, m$$

حال، نام‌های مختلف رنگ‌ها را در جدول درهم‌ساز ذخیره می‌کنیم، از سرخ شروع می‌کنیم. مقدار تابع درهم‌ساز برای عنصر است

$$h = h(red) = ۲۳۵۲۵۸۶۵۸۴ \,\%\, ۱۲ = ۰$$

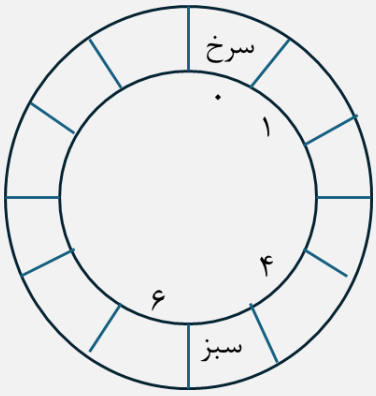
از آنجایی که جدول درهم‌ساز کاوش خطی در ابتدا خالی است، سطل با کلید k=۰ هیچ عنصری ندارد، بنابراین ما فقط عنصر را در آنجا اندیس می‌کنیم:



سپس، عنصر سبز را می‌گیریم که مقدار درهم‌ساز آن است

$$h = h(green) = ۱۵۰۸۳۱۱۲۵ \,\%\, ۱۲ = ۵$$

کلید k=۵ است، چون این سطل خالی است، می‌توان عنصر را در همان سطل ذخیره کرد.

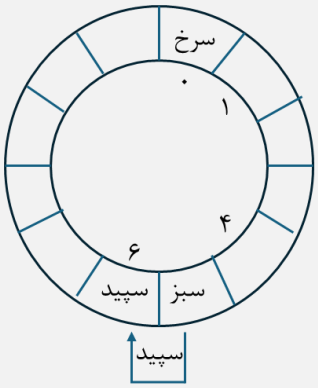


حال، عنصر سفید را در نظر می‌گیریم. مقدار درهم‌ساز آن است

$$h = h(green) = ۱۶۷۲۸۹۰۵ \,\%\, ۱۲ = ۵$$

سطل ترجیحی برای این عنصر، سطلی با کلید k=۵ است. با این حال، سطل قبلاً با عنصری متفاوت اشغال شده است، پس تصادم واقع شده است. در این حالت، الگوریتم کاوش خطی را اعمال می‌کنیم و سعی می‌کنیم سطل خالی بعدی را در جهت عقربه‌های

ساعت از موقعیت سطل ترجیحی پیدا کنیم. خوشبختانه، سطل بعدی، با کلید k=٦ آزاد است و عنصر سفید را در آنجا ذخیره می‌کنیم.



وقتی عنصر سفید را در جدول درهم‌ساز کاوش خطی جستجو می‌کنیم، ابتدا سطل ترجیحی آن را با کلید k=٥ بررسی می‌کنیم. از آنجایی که آن سطل حاوی مقداری متفاوت از عنصر است، شروع به بررسی سطل‌ها در جهت عقربه‌های ساعت می‌کنیم، از کلید $k + ۱ = ۶$ شروع می‌کنیم. خوشبختانه، سطل بعدی با کلید k = ٦ حاوی مقدار موردنظر است و می‌توانیم نتیجه بگیریم که عنصر در جدول درهم‌ساز وجود دارد.

الگوریتم برای هر عملیات به زمان O(١) نیاز دارد، تا زمانی که جدول درهم‌ساز کاوش خطی پر نباشد (ضریب بار به شدت کمتر از یک است). طولانی‌ترین دنباله کاوش در کاوش خطی به طول مورد انتظار O(log(n)) است. الگوریتم کاوش خطی به انتخاب تابع درهم‌ساز h بسیار حساس است زیرا باید توزیع یکنواخت ایده‌آل را ارائه دهد. متأسفانه، در عمل این امکان‌پذیر نیست و عملکرد الگوریتم با انحراف توزیع واقعی به سرعت کاهش می‌یابد. برای رفع این مشکل، از تکنیک‌های مختلفی برای تصادفی‌سازی اضافی به طور گسترده استفاده می‌شود.

درهم‌ساز فاخته

یکی دیگر از پیاده‌سازی‌های آدرس‌دهی باز، درهم‌ساز فاخته است که راسموس پاگ و فلمینگ فریشه رودلر در سال ۱۳۸۰ معرفی و در سال ۱۳۸۳ منتشر شد [Pa٠٤]. الگوریتم مزبور به جای بهره از یک تابع درهم‌ساز از دو تابع درهم‌ساز بهره می‌برد. جدول درهم‌ساز فاخته آرایه‌ای از سطل‌ها است، که در آن به جای یک سطل ترجیحی مانند کاوش خطی و بسیاری از الگوریتم‌های دیگر، هر مقدار دارای دو سطل نامزد است که با دو تابع درهم‌ساز متفاوت تعیین می‌شوند.

برای اندیس کردن مقدار جدید x در جدول فاخته، کلیدهای دو سطل نامزد را با توابع درهم‌ساز $h_۱$ و $h_۲$ محاسبه می‌کنیم. اگر حداقل یکی از آن سطل‌ها خالی باشد، مقدار را در آن سطل درج می‌کنیم. در غیر این صورت، به طور تصادفی یکی از آن سطل‌ها را انتخاب می‌کنیم و مقدار x را در آنجا ذخیره می‌کنیم، در حالی که مقدار را از آن سطل به سطل نامزد جایگزین آن منتقل می‌کنیم. این روش را تا زمانی که سطلی خالی پیدا شود یا تا زمانی که به حداکثر تعداد جابجایی‌ها برسیم، تکرار می‌کنیم. اگر سطل خالی وجود نداشته باشد، جدول درهم‌ساز پر در نظر گرفته می‌شود. اگرچه درهم‌ساز فاخته ممکن است دنباله‌ای از جابجایی‌ها را اجرا کند، اما زمان ثابت $O(۱)$ را برای تکمیل حفظ می‌کند.

روش جستجو ساده است و می‌تواند در زمان ثابت انجام شود. ما به سادگی باید سطل‌های نامزد را برای مقدار ورودی با محاسبه درهم‌سازهای $h_۱$ و $h_۲$ تعیین کنیم و بررسی کنیم که آیا چنین مقداری در یکی از آن سطل‌ها وجود دارد یا خیر. روش حذف نیز می‌تواند به طور مشابه انجام شود.

مثال: درهم‌ساز فاخته

جدول درهم‌ساز فاخته به طول $m = ۱۲$ با دو تابع درهم‌ساز ۳۲ بیتی MurmurHash۳ و FNV۱a که مقادیری در محدوده $\{۰,۱, \dots, m-۱\}$ تولید می‌کنند را در نظر می‌گیریم:

$$h_۱(x) = MurmurHash3(x)\%\, m$$
$$h_۲(x) = FNV1a(x)\%\, m$$

مانند مثال قبل، نام‌های رنگ را در جدول درهم‌ساز اندیس می‌کنیم و با سرخ شروع می‌کنیم. کلیدهای سطل نامزد را با اعمال آن توابع درهم‌ساز به دست می‌آوریم:

$$h_۱(red) = ۲۳۵۲۵۸۶۵۸۴ \,\%\, ۱۲ = ۰$$
$$h_۲(red) = ۱۰۸۹۷۶۵۵۹۶ \,\%\, ۱۲ = ۸$$

جدول درهم‌ساز فاخته خالی است، بنابراین از یکی از سطل‌های نامزد استفاده می‌کنیم، به عنوان مثال، سطل با کلید $k = ۰$ $h_۱(red) = ۰$ و عنصر را اندیس می‌کنیم.

0	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱
سرخ											

بعد، عنصر سیاه را اندیس می‌کنیم که سطل‌های نامزد آن $h_۱(black) = ۶$ و $h_۲(black) = ۰$ هستند. از آنجایی که سطل با کلید $k = ۰$ با عنصر دیگری اشغال شده است، فقط می‌توانیم آن را در سطل جایگزین $k = ٦$ اندیس کنیم که آزاد است.

0	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱
سرخ						سیاه					

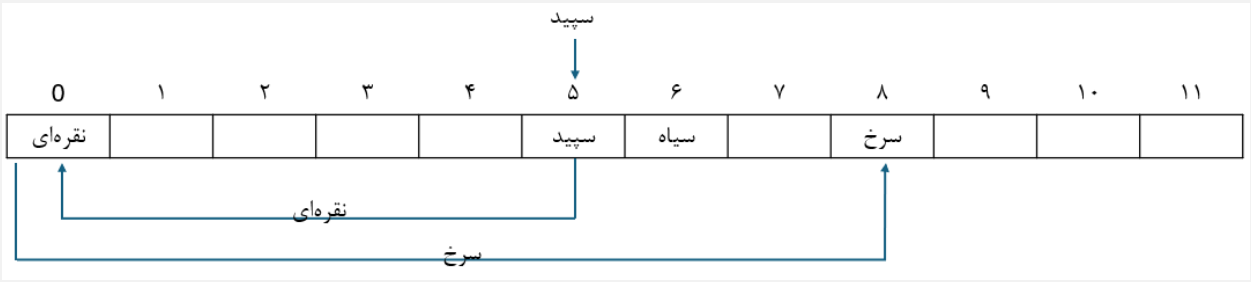
وضعیت مشابهی برای عنصر نقره‌ای با $h_۱(silver) = ۵$ و $h_۲(silver) = ۰$ وجود دارد. این عنصر را در سطل با کلید $k = ۵$ ذخیره می‌کنیم زیرا ۰ اشغال شده است.

0	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱
سرخ					نقره‌ای	سیاه					

حال عنصر سفید را در نظر می‌گیریم. مقادیر درهم‌ساز این عنصر هستند:

$$h_۱(white) = ۱۶۷۲۸۹۰۵ \,\%\, ۱۲ = ۵$$
$$h_۲(white) = ۳۷۲۴۶۷۴۹۱۸ \,\%\, ۱۲ = ۶$$

همانطور که می‌بینیم، هر دو سطل نامزد برای این عنصر اشغال شده‌اند، و باید جابجایی‌ها را طبق طرح درهم‌ساز فاخته انجام دهیم. ابتدا، به طور تصادفی یکی از سطل‌های نامزد را انتخاب کنید، فرض کنید سطل با کلید $k = ٥$ و عنصر سفید را در آن قرار دهید. عنصر نقره‌ای از سطل ۵ باید به سطل جایگزین خود، که ۰ است، منتقل شود. همانطور که می‌بینیم، سطل با کلید ۰ خالی نیست. بنابراین، عنصر نقره‌ای را ذخیره می‌کنیم و عنصر قرمز را از آن سطل به دیگر سطل نامزد خود منتقل می‌کنیم. خوشبختانه، سطل جایگزین با کلید ۸ برای عنصر قرمز آزاد است و پس از ذخیره آن در آن سطل، روش درج را به پایان می‌رسانیم.



به عنوان مثال، وقتی می‌خواهیم عنصر نقره‌ای را جستجو کنیم، فقط سطل‌های نامزد آن را که ۵ و ۰ هستند، همانطور که قبلاً محاسبه کردیم، بررسی می‌کنیم. از آنجایی که این عنصر در یکی از آنها، در این مورد سطل با کلید ۰، وجود دارد، نتیجه می‌گیریم که عنصر نقره‌ای در جدول درهم‌ساز فاخته وجود دارد.

درهم‌ساز فاخته اشغال فضای بالا را تضمین می‌کند اما نیاز دارد که طول جدول درهم‌ساز کمی بزرگتر از فضای مورد نیاز برای نگهداری همه عناصر باشد. با اصلاح طرح درهم‌ساز فاخته، داده‌ساختار احتمالاتی به نام فیلتر فاخته استفاده می‌شود که در بخش‌های آینده به تفصیل آن را توضیح خواهیم داد.

نتیجه‌گیری

در این فصل مروری بر درهم‌ساز، مشکلات آن و اهمیت آن در ساختارهای داده داشتیم. در مورد توابع درهم‌ساز رمزنگاری در مقابل غیر رمزنگاری بحث کردیم، فهرستی از توابعی که بیشتر در عمل استفاده می‌شوند را مرور کردیم، و در مورد درهم‌ساز جامع که نظرا بسیار مهم است، یاد گرفتیم. به عنوان کاربرد توابع درهم‌ساز، جدول‌های درهم‌ساز را در نظر گرفتیم که ساختارهای داده ساده‌ای هستند که کلیدها را به مقادیر نگاشت می‌دهند و به پرسش‌های عضویت پاسخ می‌دهند. نمونه‌هایی از جدول‌های درهم‌ساز آدرس‌دهی باز را مطالعه کردیم که در فصل‌های بعدی برای ساختارهای داده احتمالاتی از آنها استفاده خواهیم کرد.

در فصل بعد به بحث در مورد اولین ساختارهای داده احتمالاتی خواهیم پرداخت و افزونه‌های جدول‌های درهم‌ساز، به نام فیلترها را مطالعه خواهیم کرد که برای پاسخ به پرسش‌های عضویت تحت الزاماتی که برای کاربردهای کلان‌داده رایج هستند، مانند زمانی که فضای ذخیره‌سازی کارآمد و یا تسریع هر چه بیشتر پاسخ جستجوها لازم است استفاده می‌شوند.