دانشکده‌ی علوم ریاضی

ساختمان داده — ۱۰ آذر ۹۲

جلسه‌ی ۱۸: درهم‌سازی سرتاسری - درخت جست‌وجوی دودویی

مدرّس: دکتر شهرام خزائی — *نگارنده: معین زمانی و آرمیتا اردشیری*

۱ یادآوری

همان‌طور که درجلسات پیش مطرح شد، کاربرد درهم‌سازی زمانی می‌باشد که تعداد کلیدهای ممکن بسیار زیاد است ولی ما به ذخیره کردن تعداد محدودی از آن‌ها نیاز داریم.
برای مثال ذخیره کردن IP-Address ها را برای یک رهیاب[۱] در نظر بگیرید. IP-Address ها به صورت چهارتایی مرتب (x_1, x_2, x_3, x_4) می‌باشند که x_i ها عددی در بازه ۰ تا ۲۵۵ هستند. در این مساله تعداد کلیدهای ممکن برابر است با:

$$|U| = |IP - Address| = 256^4 = 2^{32}$$

که عدد بسیار بزرگی می‌باشد، ولی یک رهیاب در واقعیت تقریبا با $n \approx 500$ رهیاب مجاور خود در ارتباط می‌باشد و نیاز دارد که ۵۰۰ کلید (IP-Address) از میان 2^{32} کلید ممکن را ذخیره کند.
در دو جلسه گذشته دو نوع درهم‌سازی با استفاده از لیست زنجیره‌ای و آدرس‌دهی باز را مطرح کردیم. حال به درهم‌سازی زنجیره‌ای باز می‌گردیم. اگر تعداد کلیدهای موجود n و حافظه مصرفی m باشد، α ضریب بار[۲] می‌باشد که به صورت $\alpha = \frac{n}{m}$ تعریف می‌شود.
نشان دادیم که در این حالت عملیات حذف در زمان $\Theta(1)$ امکان‌پذیر است و برای رسیدن به انجام عملیات جست‌وجو در زمان $\Theta(1 + \alpha)$ فرض‌های زیر را اختیار کردیم:

۱. فرض کردیم حافظه مصرفی از مرتبه n می‌باشد، یعنی $m = O(n)$.

۲. فرض کردیم هر کلید، شانس یکسانی برای ذخیره شدن در خانه‌های جدول دارد و کلیدها به صورت مستقل از هم در خانه‌های جدول ذخیره می‌شوند. این فرض را فرض درهم‌سازی یکنواخت ساده[۳] نامیدیم.

۳. فرض کردیم تابع درهم‌ساز در زمان $O(1)$ محاسبه می‌شود.

نکته حائز اهمیت این است که $\Theta(1 + \alpha)$ مسلما بدترین زمان اجرا نمی‌باشد. حالتی را در نظر بگیرید که تمامی کلیدها در یک سطر ذخیره شده باشند، در این حالت عملیات جست و جو در زمان $O(n)$ صورت می‌گیرد.
با توجه به اینکه الگوریتم مطرح شده برای درهم‌سازی زنجیره‌ای الگوریتمی تصادفی نبود، نمی‌توان زمان اجرای متوسط را برای آن محاسبه کرد. برای حل این مشکل و پیدا کردن زمان اجرای متوسط، درهم‌سازی سرتاسری را مطرح می‌کنیم.

---
[۱] Router
[۲] Load Factor
[۳] Simple Uniform Hashing

۲ درهم‌سازی سرتاسری

برای درک بیشتر مشکل موجود در الگوریتم درهم‌سازی زنجیره‌ای مساله زیر را در نظر بگیرید:
می‌خواهیم مجموعه کلیدهای S را از میان مجموعه مرجع U در جدول درهم‌سازی به اندازه m ذخیره کنیم. با توجه به اصل لانه کبوتری خانه j ام جدول موجود است به طوری که تعداد عناصری که با نگاشت h به این خانه جدول نگاشته می‌شوند حداقل برابر است با $\frac{|U|}{m}$.
حال اگر اندازه مجموعه S از $\frac{|U|}{m}$ کوچک‌تر باشد، حمله کننده می‌تواند با دانستن تابع درهم‌ساز h مجموعه S را به گونه‌ای انتخاب نماید که تمامی اعضای آن به یک خانه از جدول نگاشته شوند تا بدترین عملکرد سیستم حاصل شود.
در مثال رهیاب، از این ایده در عمل برای اعمال حمله منع سرویس[۴] استفاده شده است. در واقع وقتی از تابع درهم‌ساز ثابتی استفاده شود همواره احتمال وقوع این نوع حمله و رسیدن به بدترین زمان اجرا $\Theta(n)$ وجود دارد. تنها روش حل این مشکل این است که تابع درهم‌ساز را به صورت تصادفی و مستقل از کلیدها انتخاب کنیم تا به عملکرد مناسبی در حالت متوسط برسیم. به این نوع درهم‌سازی، درهم‌سازی سرتاسری می‌گوییم.
مساله مرتب‌سازی سریع را به یاد بیاورید. در الگوریتم QuickSort، بدترین زمان اجرا $O(n^۲)$ بود ولی با انتخاب محورها به صورت تصادفی و ارائه الگوریتم Randomized-QuickSort توانستیم به زمان اجرای متوسط $O(n \log n)$ برسیم. در اینجا هم با روندی مشابه می‌خواهیم الگوریتم درهم‌سازی را تصادفی کنیم و الگوریتمی تصادفی ارائه دهیم که متوسط زمان اجرای آن $\Theta(۱+\alpha)$ باشد.
برای اینکار مجموعه‌ای از توابع درهم‌ساز را در نظر می‌گیریم و در ابتدای هر اجرا، یک تابع درهم‌ساز را به صورت تصادفی از این مجموعه انتخاب می‌کنیم. در این جا هم مانند حالت مرتب‌سازی سریع، تصادفی‌سازی ضمانت می‌کند که هیچ ورودی نمی‌تواند همواره بدترین حالت اجرا را حادث شود زیرا تابع درهم‌ساز را به صورت تصادفی انتخاب کرده‌ایم، و می‌توانیم با اطمینان برای هر ورودی انتظار زمان اجرای متوسط را داشته باشیم.

تعریف ۱ *خانواده‌ای از توابع درهم‌ساز را برای مجموعه مرجع U به صورت زیر در نظر بگیرید:*

$$\mathcal{H} = \{h : U \rightarrow \{۰, ۱, ۲, ..., m - ۱\}\}$$

می‌گوییم $\mathcal{H}$ یک خانواده از توابع درهم‌ساز سرتاسری است اگر به ازای هر جفت کلید متمایز $k_۱$ و $k_۲$ از مجموعه مرجع U، تعداد توابع درهم‌ساز h موجود در مجموعه $\mathcal{H}$ که به ازای آنها $h(k_۱) = h(k_۲)$ است، حداکثر برابر با $\frac{|\mathcal{H}|}{m}$ باشد.

به عبارت دیگر، هنگامی که یک تابع درهم‌ساز به صورت کاملاً تصادفی از مجموعه $\mathcal{H}$ انتخاب شود، احتمال وقوع برخورد بین کلیدهای دلخواه و متمایز $k_۱$ و $k_۲$ بیشتر از $\frac{۱}{m}$ نخواهد بود. دقت کنید اگر تابع درهم‌ساز به صورت کاملاً تصادفی مقادیر درهم‌سازی شده را به کلیدها اختصاص دهد، احتمال برخورد کلیدهای دلخواه و متمایز دقیقاً $\frac{۱}{m}$ است.

مثال ۱ *می‌خواهیم برای مجموعه IP-Address ها که به صورت $U = \{(x_۱, x_۲, x_۳, x_۴) : x_i \in \{۰, ۱, ۲, ..., ۲۵۵\}\}$ می‌باشد، یک خانواده از توابع درهم‌ساز سرتاسری مانند $\mathcal{H}$ بسازیم. خانواده $\mathcal{H}$ را به صورت زیرتعریف می‌کنیم:*

$$\mathcal{H} = \{h : (x_۱, x_۲, x_۳, x_۴) \rightarrow a_۱x_۱ + a_۲x_۲ + a_۳x_۳ + a_۴x_۴ \pmod m)\}$$

که m یک عدد اول و a_i ها از بازه $\{۰, ۱, ..., m - ۱\}$ انتخاب می‌شوند. ثابت می‌کنیم این انتخاب برای $\mathcal{H}$ منجر به یک خانواده از توابع درهم‌سازی سرتاسری می‌شود. برای اثبات، دو کلید (IP-Address) متفاوت را به صورت $x = (x_۱, x_۲, x_۳, x_۴)$ و $y = (y_۱, y_۲, y_۳, y_۴)$ در نظر می‌گیریم. بدون کاسته شدن از کلیت مساله می‌توان فرض کرد $x_۱ \neq y_۱$. برای وقوع برخورد باید رابطه $h(x) = h(y)$ برقرار باشد. با بسط این رابطه به عبارت زیر می‌رسیم:

$$a_۱x_۱ + a_۲x_۲ + a_۳x_۳ + a_۴x_۴ = a_۱y_۱ + a_۲y_۲ + a_۳y_۳ + a_۴y_۴ \pmod m) \Rightarrow$$

[۴] Denial-of-Service Attack (DoS)

$$a_1(x_1 - y_1) + a_2(x_2 - y_2) + a_3(x_3 - y_3) + a_4(x_4 - y_4) = 0 \pmod m \Rightarrow$$
$$a_1(x_1 - y_1) = -\big(a_2(x_2 - y_2) + a_3(x_3 - y_3) + a_4(x_4 - y_4)\big) \pmod m$$

با توجه به اینکه m را عددی اول انتخاب کرده‌ایم و $(x_1 - y_1)$ مقداری غیر صفر می‌باشد، می‌توان معادله را به صورت زیر نوشت:

$$a_1 = -(x_1 - y_1)^{-1} \times \big(a_2(x_2 - y_2) + a_3(x_3 - y_3) + a_4(x_4 - y_4)\big) \pmod m$$

این رابطه نشان می‌دهد که به ازای هر انتخاب دلخواه برای a_2, a_3, a_4، یک مقدار یکتا برای a_1 وجود دارد که منجر به برقراری $h(x) = h(y)$ می‌شود. بنابراین، برای هر جفت کلید متمایز x و y دقیقا به ازای m^3 مقدار (a_1, a_2, a_3, a_4)، برخورد رخ می‌دهد. از آنجا که کل حالت‌های ممکن ترکیب (a_1, a_2, a_3, a_4) برابر است با m^4، احتمال برخورد کلیدهای متمایز x و y برابر است با:

$$\frac{m^3}{m^4} = \frac{1}{m}$$

پس مجموعه $\mathcal{H}$ یک خانواده از توابع درهم‌ساز سرتاسری می‌باشد.

می‌توان نشان داد با استفاده از درهم‌سازی سرتاسری، عملیات جست‌وجو در زمان متوسط $O(1 + \alpha)$ قابل پیاده‌سازی است.

۳ درخت جست‌وجوی دودویی

۱.۳ درخت جست‌وجو

درخت‌های جست‌وجو، داده‌ساختارهایی مناسب برای انجام عملیات روی مجموعه‌های پویا[۵] هستند. این داده‌ساختارها را می‌توان هم به صورت لغت‌نامه[۶] و هم به صورت یک صف اولویت[۷] استفاده کرد. عملیات ابتدایی روی یک درخت جست‌وجو، زمانی متناسب با ارتفاع درخت را صرف می‌کند.

۲.۳ درخت جست‌وجوی دودویی

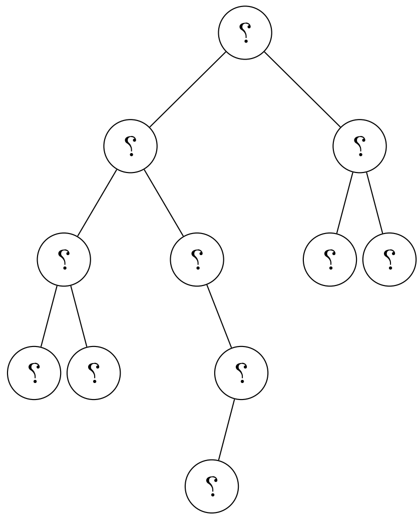
درخت‌های جست‌وجوی دودویی، یکی از مهم‌ترین داده‌ساختارها برای مجموعه‌های پویا هستند. این داده‌ساختار بسیاری از عملیات روی مجموعه‌های پویا را در زمان $O(h)$ انجام می‌دهد، که h ارتفاع درخت است. در این داده‌ساختار هر گره[۸] دارای حداکثر دو فرزند است و دارای مؤلفه‌های زیر است:

- key: که داده موجود در گره است.
- $left$: که به گره سمت چپ گره کنونی اشاره می‌کند.
- $right$: که به گره سمت راست گره کنونی اشاره می‌کند.
- p: که به والد گره کنونی اشاره می‌کند.
- کلید ذخیره شده باید در شرط درخت جست‌وجوی دودویی بودن صدق کند:

[۵] Dynamic Sets
[۶] Dictionary
[۷] Priority queue
[۸] Node

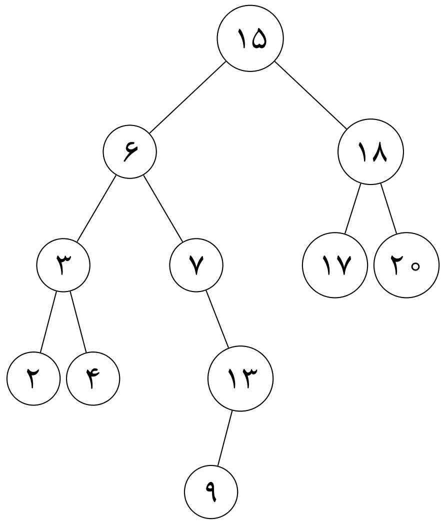
- اگر گره y در زیر درخت سمت چپ گره x قرار دارد، آن گاه $y.key \leq x.key$
- اگر گره y در زیر درخت سمت راست گره x قرار دارد، آن گاه $y.key > x.key$

مثال ۲ *آرایه* $\langle ۱۵, ۱۸, ۲۰, ۱۷, ۱۳, ۷, ۳, ۶, ۹, ۴, ۲ \rangle$ *را در درخت دودویی زیر طوری وارد کنید که حاصل ویژگی درخت دودویی جست‌وجو را داشته باشد:*



شکل ۱: یک درخت دودویی

جواب. آرایه مرتب‌شده را در نظر بگیرید و خود را قانع کنید که پاسخ به صورت زیر است!



شکل ۲: درخت جست‌وجوی دودویی پر شده

۱.۲.۳ پیمایش درخت جست‌وجوی دودویی

درخت جست‌وجوی دودویی به ما این امکان را می‌دهد که کلیدهای موجود در درخت را به صورت مرتب نمایش دهیم. برای این منظور لازم است که درخت به نحو مناسبی اصطلاحاً پیمایش[۹] شود. نحوه پیمایش مناسب برای نمایش آرایه مرتب متناظر با یک درخت جست‌وجوی دودویی، پیمایش میان‌ترتیب[۱۰] نامیده می‌شود که با فراخوانی الگوریتم Inorder-BST-Traverse روی ریشه درخت امکان‌پذیر است. این الگوریتم روی هر گرهی مانند x از درخت قابل فراخوانی است؛ ابتدا بررسی می‌کند که گره ابتدایی برابر NIL نباشد. اگر این گونه نبود به صورت بازگشتی کلید گره‌های زیر درخت سمت چپ x، سپس کلید خود x را و در آخر به صورت بازگشتی کلید گره‌های زیر درخت سمت راست x را نمایش می‌دهد.

Algorithm 1 Inorder-BST-Traverse

```
function Inorder-BST-Traverse(node x)
    if x ≠ NIL then
        Inorder-BST-Traverse(x.left)
        Print x.key
        Inorder-BST-Traverse(x.right)
```

زمان اجرا

- برای یک درخت با n گره، هر گره را یک بار بررسی می‌کنیم در نتیجه زمان اجرا از مرتبه $\Theta(n)$ خواهد بود.

مثال ۳ *به عنوان مثال، الگوریتم پیمایش میان ترتیب درخت را بر روی درخت مثال ۲ اجرا می‌کنیم.*
برای این‌کار از ریشه درخت شروع می‌کنیم و به سراغ زیردرخت سمت چپ آن می‌رویم که مقدار کلید ریشه آن برابر است. با ادامه الگوریتم به چپ‌ترین گره می‌رسیم، که مقدار کلیدش برابر ۲ است. با توجه به اینکه زیردرخت سمت چپ این گره تهی می باشد، این گره خوانده می‌شود و بعد از آن گره با کلید ۳ و سپس گره با کلید ۴ خوانده می‌شوند.
با ادامه الگوریتم آرایه خوانده شده به صورت زیر بدست می‌آید:

$$\langle ۲, ۳, ۴, ۶, ۷, ۹, ۱۳, ۱۵, ۱۷, ۱۸, ۲۰ \rangle$$

مشاهده می‌شود که با پیمایش میان ترتیب درخت جست‌وجوی دودویی، آرایه خروجی حاصل، به صورت صعودی می‌باشد.

۲.۲.۳ جست‌وجو در درخت جست‌وجوی دودویی

برای یافتن کلیدی در درخت جست‌وجوی دودویی از الگوریتم BST-Search استفاده می‌کنیم. برای اینکار از ریشه درخت شروع می‌کنیم. اگر ریشه برابر با NIL باشد، درخت تهی می‌باشد و جست‌وجو ناموفق خواهد بود. در غیر این صورت، کلید مورد نظر را با مقدار کلید گره ریشه مقایسه می‌کنیم، اگر برابر بودند، گره ریشه همان گره مورد نظر است. در غیر این صورت، دو حالت پیش خواهد آمد:

- مقدار مورد جست‌وجو از گره ریشه کوچکتر است. در این حالت، هیچ عنصری در زیردرخت سمت راست وجود ندارد که کلید آن برابر با مقدار مورد جست‌وجو باشد. بنابراین جست‌وجو را در زیردرخت سمت چپ ادامه می‌دهیم.

[۹] traverse
[۱۰] inorder traversal

• مقدار مورد جست‌وجو از گره ریشه بزرگتر است. در این حالت، هیچ عنصری در زیردرخت سمت چپ وجود ندارد که کلید آن برابر با مقدار مورد جست‌وجو باشد. بنابراین جست‌وجو را در زیردرخت سمت راست ادامه می‌دهیم.

سپس باتوجه به یکی از دو حالت فوق، زیردرخت سمت چپ یا زیردرخت سمت راست را به روش بازگشتی و با استفاده از این الگوریتم جست‌وجو می‌کنیم.

Algorithm 2 BST-Search

```
function BST-Search(node x, key k)
    if x = NIL or k = x.key then
        return x
    if k < x.key then
        return BST-Search(x.left, k)
    else
        return BST-Search(x.right, k)
```

زمان اجرا

• این الگوریتم به صورت بازگشتی گره‌ها را از ریشه به پایین بررسی می‌کند، در نتیجه زمان اجرا از مرتبه $O(h)$ خواهد بود، که در آن h ارتفاع درخت است.

۳.۲.۳ پیدا کردن بیشینه و کمینه در درخت جست‌وجوی دودویی

شرط درخت جست‌وجوی دودویی بودن ضمانت می‌کند که:

• کلید کمینه در چپ‌ترین گره، موجود است.

• کلید بیشینه در راست‌ترین گره، موجود است.

با استفاده از این شرط می‌توانیم درخت را تا جایی که به NIL برسیم از چپ (یا راست) برای رسیدن به کلید کمینه (یا بیشینه) پیمایش کنیم. این کار را می‌توانیم با دو الگوریتم BST-Minimum و BST-Maximum انجام دهیم.

Algorithm 3 BST-Minimum

```
function BST-Minimum(node x)
    while x.left ≠ NIL do
        x ← x.left
    return x
```

Algorithm 4 BST-Maximum

```
function BST-Maximum(node x)
    while x.right ≠ NIL do
        x ← x.right
    return x
```

زمان اجرا

• هر دو الگوریتم مانند الگوریتم پیشین (جست‌وجو در درخت جست‌وجوی دودویی) عمل می کنند، در نتیجه زمان اجرا از مرتبه $O(h)$ خواهد بود.

۴.۲.۳ گره پیشین و پسین در درخت جست‌وجوی دودویی

اگر فرض کنیم هیچ دو کلیدی در درخت جست‌وجوی دودویی برابر نباشند، گره پسین[۱۱] x ، گره y است که در آن $y.key$ کوچک‌ترین کلید بزرگ‌تر از $x.key$ است. اگر x بزرگ‌ترین کلید در درخت را داشته باشد آن‌گاه گره پسین آن را NIL اعلام می‌کنیم. پس دو حالت وجود دارد:

۱. اگر گره x زیر درختی ناتهی در سمت راست داشته باشد، آن‌گاه گره پسین x کلید کمینه در زیر درخت سمت راست x است.

۲. اگر گره x زیر درختی تهی در سمت راست داشته باشد، آن‌گاه برای پیدا کردن گره پسین x از این گره به سمت ریشه حرکت می‌کنیم تا به گره‌ای برسیم که فرزند سمت چپ پدرش باشد. پدر این گره، گره مورد نظر می‌باشد.

با توضیحات بالا الگوریتم BST-SUCCESSOR را ارائه می دهیم.

Algorithm 5 BST-SUCCESSOR

```
function BST-SUCCESSOR(node x)
    if x.right ≠ NIL then
        return BST-MINIMUM(x.right)
    y ← x.p
    while y ≠ NIL and x = y.right do
        x ← y
        y ← y.p
    return y
```

مثال ۴ *به عنوان نمونه درخت دودویی ارائه شده در مثال ۲ (شکل ۱) را در نظر بگیرید:*

- *گره پسین گره با کلید ۱۵ ، گره با کلید ۱۷ است، یعنی گره کمینه زیردرخت سمت راست آن.*
- *گره پسین گره با کلید ۴، گره با کلید ۶ است.*

زمان اجرا

- این الگوریتم نیز مانند ۳ الگوریتم پیشین به دلیل پیمایش درخت به پایین (یا بالا)، دارای زمان اجرا از مرتبه $O(h)$ خواهد بود.

گره پیشین[۱۲] را می‌توان با الگوریتم مشابه BST-PREDECESSOR پیدا کرد.

Algorithm 6 BST-PREDECESSOR

```
function BST-PREDECESSOR(node x)
    if x.left ≠ NIL then
        return BST-MAXIMUM(x.left)
    y ← x.p
    while y ≠ NIL and  x = y.left do
        x ← y
        y ← y.p
    return y
```

[۱۱] Successor
[۱۲] Predecessor