جزوه جلسه بیست و هفتم داده ساختارها و الگوریتم

۵ دی ۱۴۰۰

فهرست مطالب

۱ داده ساختارهای مرتبط با رشته (String)

۱.۱ مسئله تطابق رشته ها

یک رشته بزرگ به نام t و یک رشته کوچک s داریم. میخواهیم s را به عنوان زیررشته ای از t پیدا کنیم.
مسئله فوق یک مسئله کلاسیک در الگوریتم است.
الگوریتم های معروف حل این مسئله عبارتند از:

- الگوریتم KMP در زمان $\Theta(|t| + |s|)$
- الگوریتم بویر-مور که از روی رشته s یک اتوماتا درست می کند و با پیمایش t در آن اتوماتا، وجود s بررسی می شود. زمان اجرای آن نیز $\Theta(|t| + P(|s|))$ می باشد که P یک چندجمله ای می باشد.
- الگوریتم رابین-کارپ با استفاده از درهم سازی، برای هر زیر رشته به طول $|s|$ در t، هش آن را محاسبه می کنیم و در زمان $O(1)$ نیز در رشته t جلو میرویم و هش را مجددا حساب می کنیم. زمان اجرا نیز برابر با $\Theta(|s| + |t|)$ می باشد.

نسخه دیگر از مسئله بالا، به این صورت است که در هر Query، یک رشته s ورودی داده میشود تا در الگوریتم بررسی شود (مشابه کاری که IDEها هنگام indexing پروژه انجام میدهند.)

۲.۱ مسئله دیکشنری

رشته های $T_1, T_2, ..., T_k$ را داریم. پس از انجام پیش پردازش، در هر Query یک رشته s داده می شود و باید جایگاه s خروجی داده می شود. اگر فرض کنیم T_iها به صورت لغت نامه ای (Lexicographical) مرتب شده باشند، رشته قبل s مورد پرسش است: زیرا لزوما s در T_iها نیست.
یک ایده ساده برای حل این مسئله، جست و جوی دودویی ساده است. در این صورت هر پرسش در $O(|s|.logk)$ پاسخ داده می شود. با انجام درهم سازی روی پیشوند کلمات، این زمان به $O(logk.log|s| + |s|)$ نیز کاهش می یابد.
هدف ما رسیدن به زمان $O(log|\Sigma| + |s|)$ می باشد که Σ تعداد کل حروف الفبا می باشد.

۳.۱ ترای و ترای فشرده (Trie)

Trie یک داده ساختار درختی برای حل مسائل ذکر شده می باشد. اسم درخت نیز از کلمه re**Trie**val به معنی بازیابی می آید.

در این درخت، فرزندان هر راس با حروف القبا (Σ) در ارتباط هستند و کلیدها به جای رئوس روی مسیر از ریشه به برگ ها ذخیره می شوند. در انتهای هر رشته نیز کاراکتر $ به منظور نمایش انتهای رشته می آید.

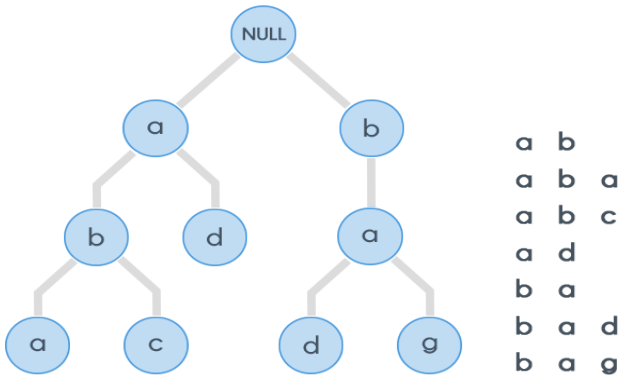


Fig. 1

شکل ۱: مثال یک Trie

درج هر رشته به طول k، در $O(k)$ انجام می شود.
هر جست و جو نیز در $O(n)$ انجام می شود که n طول بزرگترین کلمه در Trie است.
ایراد این داده ساختار حافظه مصرفی زیاد آن در پیاده سازی معمولی است که بیشتر حافظه نیز بلا استفاده می ماند.

۱.۳.۱ فشرده سازی **Trie**

فشرده سازی یکی از راه حل های کاهش حافظه مصرفی می باشد.
در ساده ترین نوع فشرده سازی، فشرده سازی مسیر هایی است که شاخه ندارند. با حفظ ساختار قبلی درخت، فشرده سازی انجام می شود و تعداد فرزندان هر راس تغییری نمی کند و تنها محتویات یال ها تغییر می کند. در صورت نیاز و به هنگام اضافه کردن رشته جدید، این فشرده سازی مجدد به حالت قبلی برمی گردد.
در حالت غیر فشرده مجموع کل یال ها به تعداد حروف رشته ها وابسته است؛ در صورتی که در شکل فشرده، به تعداد رشته ها وابسته است و برای k رشته، تعداد کل یال ها حداکثر $2k-1$ می باشد. تعداد کل راس ها نیز حداکثر $2k$ تا می باشد.
برای حل مسئله دیکشنری با این داده ساختار، با داشتن s، تا جای ممکن در درخت مسیر را از ریشه طی می کنیم. هنگامی که دیگر امکان حرکت نبود، اولین مسیر منتهی

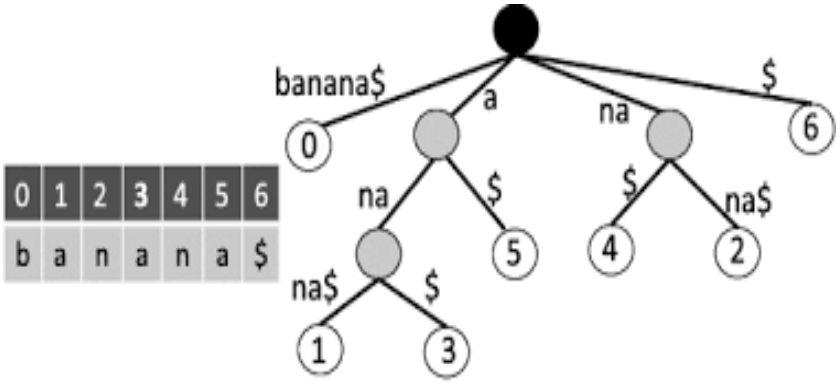
به برگ ($) را خروجی می دهیم. (منظور از اولین راس یعنی در هر راس به اولین راس مراجعه کنیم تا به انتهای رشته برسیم). بدین ترتیب هر Query در زمان $\Theta(|s| + |\Sigma|)$ پاسخ داده می شود.

۲.۳.۱ داده ساختارهای هر راس **Trie**

- آرایه به طول Σ: زمان هر کوئری برابر با $O(|s|+|\Sigma|)$ می باشد اما حافظه مصرفی آن زیاد است.
- لیست مرتبط یا درخت دودویی جست و جو: حافظه این روش کمتر از روش اول است و زمان هر کوئری نیز به $O(|s|.log(|\Sigma|))$ می رسد.
- جدول درهم سازی: حافطه این روش نیز از روش اول بهتر است و زمان هر کوئری نیز مانند روش اول، $O(|s| + |\Sigma|)$ می باشد.
- درخت ون امدبواس: حافظه این روش نیز بهینه است و زمان پاسه به هر کوئری نیز به $O(|s|.loglog|\Sigma|)$ می رسد.
- درخت دودویی جست و جوی متوازن وزن دار: زمان هر پاسخ به هر پرسش برابر با $O(|s|+logk)$ می باشد که این زمان را تا $O(|s|+log|\Sigma|)$ نیز کاهش می یابد.

۴.۱ درخت پسوندی یا **Suffix Tree**

درخت پسوندی، یک ترای فشرده است که شامل همه پسوندهای یک رشته می باشد.



شکل ۲: مثال یک Suffix Tree

این داده ساختار قابلیت حل هردو مسئله ذکر شده در ابتدا را دارد. برای پیدا کردن زیر

رشته s در t، هم زمان با پیمایش درخت و خواندن s، آنرا پیدا می کند.
درخت پسوندی در زمان $\Theta(|t|)$ ساخته می شود!

کاربردهای درخت پسوندی:

- تطابق رشته ها
- در مسئله تطابق رشته ها، به جای پیدا کردن یک مورد خاص، تعداد تکرار و حتی محل آن را گزارش می دهیم.
- طولانی ترین زیر رشته تکراری در یک رشته
- طولانی ترین زیر رشته مشترک kتا زیر رشته (این کار در زمان خطی $O(\Sigma(k_i))$ (جمع طول زیر رشته ها) انجام می شود.) بدین منظور از k رشته داده شده یک رشته به صورت $T_1\$_1T_2\$_2...T_k\$_k$ می سازیم و درخت پسوندی آنرا می سازیم. برای هر راس می فهمیم آیا در صورت ادامه دادن از آن به تمام $\$_i$ها میرسیم. اگر راسی با این شرایط وجود داشته باشد، از ریشه تا آن راس در تمام رشته های T_i وجود دارد. برای کل درخت این عملیات را انجام میدهیم و رشته با طول ماکسیمم را خروجی می دهیم.

۱.۴.۱ آرایه پسوندی یا **Suffix Array**

آرایه پسوندی معادل فشرده و ساده تر درخت پسوندی است. درخت پسوندی را از روی ظاهرش می توان فهمید اما آرایه پسوندی اینگونه نیست.
برای ساختن آرایه پسوندی، ابتدا به هر پیشوند یک رقم از صفر تا $|t|$ میدهیم و سپس پیشوندها را مرتب می کنیم و ایندکس پیشوند های مرتب شده را در آرایه پیشوندی قرار می دهیم.
مثال برای Banana:

$0.Banana\$$
$1.anana\$$
$2.nana\$$
$3.ana\$$
$4.na\$$
$5.a\$$
$6.\$$
$\implies SuffixArray = [6, 5, 3, 1, 0, 4, 2]$

از روی درخت پیشوندی و آرایه پیشوندی، هرکدام را می توان سریع ساخت.
آرایه کمکی دیگری به نام LCP وجود دارد که نشان می دهد در حالت مرتب شده پسوندها که از روی آنها آرایع پسوندی را ساختیم، در چند حرف باهم مشترک هستند.

پیشوندها به صورت متوالی بررسی می شوند.
برای مثال بالا داریم:
$LCP = [0, 1, 3, 0, 0, 2]$
داشتن آرایه LCP باعث می شود حل مسئله تطابق رشته ها از $O(|s|.log|t|)$ به $O(|s| + log|t|)$ برسد.
خود آرایه پسوندی را می توان مستقیما از روی رشته t در زمان $O(|t|.log^2|t|)$ ساخت که این زمان به $O(|t|.log|t|)$ نیز کاهش می یابد.

مسئله تعداد زیر رشته های متفاوت یک رشته:
به کمک آرایه LCP می توان این مسئله را حل کرد. بدین منظور از تعداد کل زیر رشته ها باید تعداد زیر رشته های تکراری را کم کرد. تعداد زیررشته های تکراری نیز برابر با جمع عناصر آرایه LCP می باشد.
تعداد کل زیر رشته های متفاوت یک رشته = $\binom{n}{2}$ - $\Sigma LCP[i]$

۲.۴.۱ تبدیل باروز-ویلر

برای کاهش حافظه مصرفی، با دریافت ورودی، همه rotation های آن را تولید می کنیم و به ترتیب Lexicographical مرتب می کنیم. سپس آخرین حرف هر rotation را ذخیره می کنیم. در خروجی نیز تعداد تکرار حروف ورودی مشخص است. همچنین به جای ذخیره سازی کل حروف، تعداد تکرار را برای هر کدام ذخیره می کنیم.
برای مثال اگر ورودی $Banana\$$ باشد، خروجی به شکل $Bnn\hat{}aa\$a$ می شود.

عکس تبدیل باروز-ویلر:
با داشتن ستون آخر و مرتب سازی آنها، ستون اول به دست می آید. حال با یک شیفت، ۲ ستون اول به صورت نامرتب ساخته می شوند. حال می توان با مرتب کردن آن ها و ادامه این فرایند ورودی اولیه را ساخت.

این الگوریتم به جز رشته ها و متن ها، برای فشرده سازی فایل ها نیز استفاده می شود.