Hyvin suuret aakkostot

- Jos σ on hyvin suuri, voidaan kukin merkki jakaa osiin, niin että saadaan pidempiä merkkijonoja pienemmässä aakkostossa. Esimerkiksi:

```
83 111 114 116
      ||
      V
01010011 01101111 01110010 01110100
      ||
      V
0101 0011 0110 1111 0111 0010 0111 0100
      ||
      V
5 3 6 15 7 2 7 4
```

- Jos merkit jaetaan k osaan saadaan algoritmi, jonka aikavaatimus on $\mathcal{O}(k \cdot DP(\mathcal{R}) + \sigma^{1/k})$ ja tilavaatimus $\mathcal{O}(n + \sigma^{1/k})$.
- Jos $\sigma > n$, voidaan valita $k = \lceil \log_n \sigma \rceil$, jolloin $\sigma^{1/k} \leq n$. Tällöin aikavaatimus on $\mathcal{O}(DP(\mathcal{R}) \log_n \sigma)$ ja tilavaatimus $\mathcal{O}(n)$.
- Jos $\sigma = n^{\mathcal{O}(1)}$, eli jos yksi merkki mahtuu $\mathcal{O}(\log n)$ bittiin, k on vakio ja aikavaatimus on $\mathcal{O}(DP(\mathcal{R}))$ eli algoritmi on optimaalinen.

3.5 Hakurakenteet

- Siirrytään nyt järjestämisestä hakurakenteisiin.
- Yleisesti, hakurakenne on tietorakenne, joka tallentaa joukon R ja mahdollistaa seuraavien operaatioiden nopean suorituksen:
  - Lisäys
  - Poisto
  - Haku: Tutki, sisältääkö R alkion x, ja jos sisältää, palauta siihen mahdollisesti liittyvät tiedot.
  - Naapurihaku: Jos R ei sisällä alkiota x, palauta sen lähin edeltäjä tai seuraaja R:ssä.

- Naapurihaku mahdollistaa esim. osavälihaut: Etsi kaikki R:n alkiot, jotka ovat välillä $[x, y]$.
- Kaikki hakurakenteet eivät tue naapurihakua, esimerkkinä hajautustaulu.
- Ylläoleva koskee *dynaamisia* hakurakenteita. *Staattinen* hakurakenne ei mahdollista lisäystä ja poistoa, ainoastaan hakurakenteen muodostamisen annetulle joukolle R. Esimerkiksi järjestetty taulukko on staattinen hakurakenne.

Alkuosahaut

- Merkkijonojoukolle $\mathcal{R}$ voidaan yllämainittujen lisäksi määritellä hakuoperaatiot:
  - Alkuosahaku: Tutki, onko S jonkin $\mathcal{R}$:n jonon alkuosa. Jos on, palauta kaikki sellaiset jonot.
  - Pisin yhteinen alkuosa: Jos S ei ole minkään $\mathcal{R}$:n jonon alkuosa, mikä on S:n pisin alkuosa, joka on.
- Alkuosahaut ovat läheisessä yhteydessä naapurihakuun.
  - Jos $\mathcal{R}$:ssä on jonoja, joiden alkuosa on S, S:n lähin seuraaja $\mathcal{R}$:ssä on ensimmäinen niistä ja loput ovat sen välittömiä seuraajia.
  - Jonolla S on pisin yhteinen alkuosa sen edeltäjän tai seuraajan kanssa.

Perushakurakenteet merkkijonoille

- Perushakurakenteita, kuten binääripuita, voidaan käyttää myös merkkijonoille.
- Kuten edellä nähtiin, myös alkuosahaut voidaan toteuttaa naapurihaun avulla. (Toisaalta esim. hajautustaulu ei tue alkuosahakuja.)
- Operaatiot kuitenkin hidastuvat vertailujen hitauden vuoksi.
- Järjestämisen alarajasta $\Omega(n \log n \log_\sigma n)$ seuraa, että joko lisäyksen tai naapurihaun keskimääräisen aikavaatimuksen alaraja on $\Omega(\log n \log_\sigma n)$, koska järjestäminen voidaan suorittaa lisäämällä alkiot hakurakenteeseen ja suorittamalla naapurihakuja.

- Poisto ei kuitenkaan välttämättä hidastu, koska se ei tarvitse vertailuja.

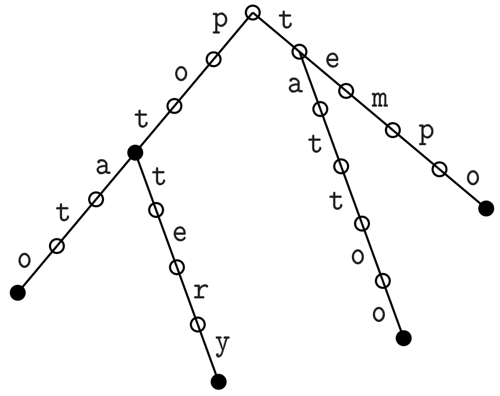
3.6 Trie

Renee de la Briandais: File Searching Using Variable Length Keys. Teoksessa *Proc. AFIPS Western Joint Computer Conference*, 15:295-298, March 1959.

Edward Fredkin: Trie Memory. *Communications of the ACM*, 3(9):490-499, September 1960.

- Trie on juurellinen puu, jonka kaaret on nimetty merkeillä.
- Solmu v esittää merkkijonoa, joka saadaan katenoimalla kaarten nimet polulla juuresta solmuun v.
- Kustakin solmusta sen lapsiin johtavat kaaret on nimetty eri merkeillä. Sen ansiosta kaksi solmua esittää aina eri jonoja.
- Merkkijonojoukon $\mathcal{R}$ trie $trie(\mathcal{R})$ on pienin trie, joka esittää jokaisen $\mathcal{R}$:n jonoista.

Esimerkki 3.6.1 *Joukon* $\mathcal{R} = \{\texttt{pot}, \texttt{potato}, \texttt{pottery}, \texttt{tattoo}, \texttt{tempo}\}$ *trie.*



- Trie nähtiin jo aiemmin Aho-Corasick-automaatin runkona.
- Oletetaan vakioaakkosto.
  - Joukon $\mathcal{R}$ trien tilavaatimus on $\mathcal{O}(||\mathcal{R}||)$, missä $||\mathcal{R}||$ on $\mathcal{R}$:n jonojen kokonaispituus.

- Merkkijonon S lisäys, poisto, ja perushaku voidaan toteuttaa ajassa $\mathcal{O}(|S|)$.
- Alkuosahaku jonolla S vie ajan $\mathcal{O}(|S| + len)$, missä len on vastauksena saatujen merkkijonojen yhteispituus.
- Jos riittää tutkia, onko S jonkin $\mathcal{R}$:n alkuosa, se voidaan tehdä ajassa $\mathcal{O}(|S|)$.

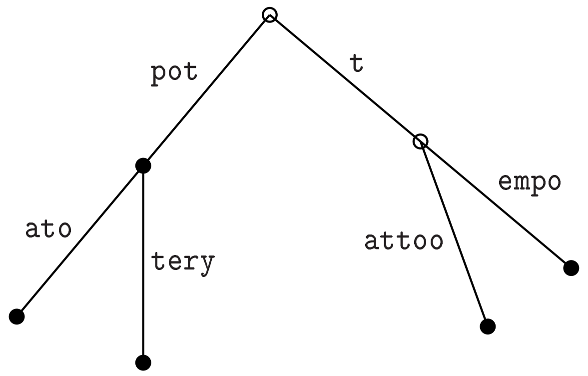
• Jos aakkoston koko ei ole vakio, täytyy määritellä, miten haarautuminen solmussa toteutetaan.

Tiivistetty trie

D. R. Morrison: PATRICIA — Practical Algorithm to Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

• Tiivistetty trie saadaan triestä, korvaamalla jokainen haarautumaton polku yhdellä kaarella, jonka nimetään korvattujen kaarien nimien katenaatiolla.

Esimerkki 3.6.2 *Joukon* $\mathcal{R} = \{$`pot`, `potato`, `pottery`, `tattoo`, `tempo`$\}$ *tiivistetty trie.*



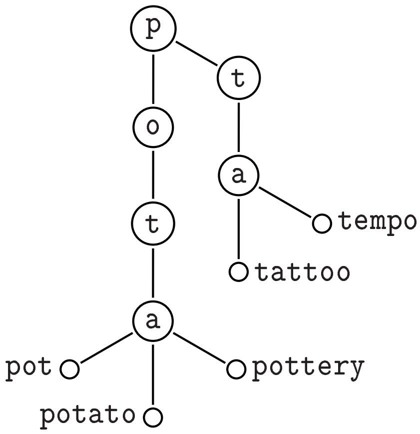
• Alkuosahaku jonolla S vie nyt ajan $\mathcal{O}(|S| + occ)$, missä occ on vastauksena saatujen merkkijonojen lukumäärä (ei yhteispituus).

3.7 Ternääripuu

Jon L. Bentley & Robert Sedgewick: Fast Algorithms for Sorting and Searching Strings. Teoksessa *Proc. 8th Annual ACM–SIAM Symposium on Discrete Algorithms*, sivut 323–350, 1997.

- Solmujen haarautumisen toteuttaminen on trien toteutuksen ongelmakohta.
- Sen vuoksi käytännössä yksinkertaisempi hakurakenne merkkijonoille on *ternääripuu*.
- Ternääripuu on kuin binääripuu, mutta
  - Kussakin sisäsolmussa haaraudutaan kahden sijasta kolmeen: pienempään, yhtä suureen, ja suurempaan.
  - Haarautuminen tapahtuu yhden merkkiposition perusteella.
- Ternääripuun suhde binääripuuhun on siis sama kuin merkkijonopikajärjestämisen suhde tavalliseen pikajärjestämiseen. (Trietä taas voisi verrata kantalukujärjestämiseen.)

Esimerkki 3.7.1 *Joukon* $\mathcal{R} = \{\texttt{pot}, \texttt{potato}, \texttt{pottery}, \texttt{tattoo}, \texttt{tempo}\}$ *ternääripuu*



- Ternääripuu voidaan nähdä trien ja binääripuun yhdistelmänä.
- Alipuun paino on sen lehtien lukumäärä.
- Ternääripuu on tasapainossa, jos jokaisen vasemman ja oikean alipuun paino on enintään puolet sen vanhemman painosta.

- Tasapainotus voidaan tehdä kierroilla aivan kuten binääripuissa.
- Lähelle tasapainoa päästään myös esim. lisäämällä jonot puuhun satunnaisessa järjestyksessä.
- Tarkastellaan merkkijonon S hakua tasapainotetussa joukon $\mathcal{R}$ ternääripuussa.
  - Jokainen askel puussa alaspäin joko etenee jonossa S (keskihaara) tai puolittaa jäljellä olevien $\mathcal{R}$:n jonojen määrän.
  - Siten hakuaika on $\mathcal{O}(|S| + \log |\mathcal{R}|)$.
- Siis n:n jonon tasapainoisessa ternääripuussa:
  - Merkkijonon S lisäys, poisto, ja perushaku voidaan toteuttaa ajassa $\mathcal{O}(|S| + \log n)$.
  - Alkuosahaku jonolla S vie ajan $\mathcal{O}(|S| + \log n + len)$, missä len on vastauksena saatujen merkkijonojen yhteispituus.
  - On myös mahdollista saavuttaa alkuosahaun aikavaatimus $\mathcal{O}(|S| + \log n + occ)$, missä occ on vastauksena saatujen merkkijonojen lukumäärä. (HT)
  - Jos riittää tutkia, onko S jonkin $\mathcal{R}$:n alkuosa, se voidaan tehdä ajassa $\mathcal{O}(|S| + \log n)$.
- Ternääripuun koko on $\mathcal{O}(DP(\mathcal{R}))$.

3.8 Merkkijonobinäärihaku

U. Manber & G. Myers: Suffix Arrays: A New Method for On-line String Searches. *SIAM Journal on Computing* 22(5):935–948, 1993.

- Järjestetty taulukko on yksinkertainen staattinen hakurakenne, jossa haut voidaan tehdä binäärihakuna käyttäen $\mathcal{O}(\log n)$ alkioiden vertailua.

Algoritmi 3.8.1 *Binäärihaku*

Syöte: Järjestetty joukko $R = \{k_0, k_1, \ldots k_{n-1}\}$, hakualkio x.
Tuloste: i, jolla $k_{i-1} < x \leq k_i$ tai
0, jos $x \leq k_0$; tai n, jos $k_{n-1} < x$;

(1) $left := 0$; $right := n$; (* invariantti: tulos on välillä $[left, right]$ *)
(* eli $k_{left-1} < x \leq k_{right}$ *)
(2) **while** $left < right$ **do**
(3) $mid := \lfloor (left + right)/2 \rfloor$;
(4) **if** $x \leq k_{mid}$ **then** $right := mid$;
(5) **else** $left := mid + 1$;
(6) **return** $right$;

- Kun alkiot ovat merkkijonoja, pahimman tapauksen hakuaika on $\mathcal{O}(m \log n)$, missä m on hakujonon pituus.
- Hakua voidaan kuitenkin nopeuttaa hyödyntäen yhteisten alkuosien pituuksille päteviä säännönmukaisuuksia.
- Merkitään $lcp(A, B)$ kahden merkkijonon A ja B pisimmän yhteisen alkuosan (longest common prefix) pituutta.

Lemma 3.8.2 *Olkoon $A \leq B \leq C$ merkkijonoja. Tällöin $lcp(A, C) = \min\{lcp(A, B), lcp(B, C)\}$.*

Todistus.

- Oletetaan, että $\ell = lcp(A, B) \leq lcp(B, C)$. (Päinvastainen tilanne on symmetrinen.)
- Tällöin $A[0 \ldots \ell) = B[0 \ldots \ell) = C[0 \ldots \ell)$. Siten $lcp(A, C) \geq \ell$.
- Toisaalta joko $|A| = \ell$ tai $A[\ell] < B[\ell] \leq C[\ell]$. Kummassakin tapauksessa $lcp(A, C) \leq \ell$.

□

Lemma 3.8.3 *Olkoon $A \leq B, B' \leq C$ merkkijonoja. Tällöin $lcp(B, B') \geq lcp(A, C)$.*

Todistus.

- Olkoon $B_{min} = \min(B, B')$ ja $B_{max} = \max(B, B')$.

- Lemman 3.8.2 perusteella

$$
\begin{aligned}
lcp(A,C) &= \min(lcp(A,B_{max}), lcp(B_{max},C) \leq lcp(A,B_{max}) \\
&= \min(lcp(A,B_{min}), lcp(B_{min},B_{max}) \leq lcp(B_{min},B_{max})
\end{aligned}
$$

□

- Tarkastellaan jotakin hetkeä haun aikana, kun haetaan jonoa P järjestetystä merkkijonojoukosta $\mathcal{R} = \{S_0, S_1, \dots S_{n-1}\}$.
- Tällöin $S_{left-1} < P \leq S_{rigth}$. Toisaalta $S_{left-1} \leq S_{mid} \leq S_{right}$, joten lemman 3.8.3 mukaan

  $$lcp(P, S_{mid}) \geq lcp(S_{left-1}, S_{right}) = \min\{lcp(P, S_{left-1}), lcp(P, S_{right})\} \ ,$$

  missä oletetaan, että $lcp(X, S_{-1}) = lcp(X, S_n) = 0$ kaikilla X.
- Siis verrattaessa jonoja P ja S_{mid} voidaan ensimmäiset $\min\{lcp(P, S_{left-1}), lcp(P, S_{right})\}$ merkkiä ohittaa.
- Saadaan seuraava algoritmi (missä $X[i \dots)$ tarkoittaa jonon X positiosta i alkavaa loppuosaa).

Algoritmi 3.8.4 *Merkkijonobinäärihaku*

Syöte: Järjestetty merkkijonojoukko $\mathcal{R} = \{S_0, S_1, \dots S_{n-1}\}$, hakujono P.
Tuloste: i, jolla $S_{i-1} < P \leq S_i$ tai
    0, jos $P \leq S_0$; tai n, jos $S_{n-1} < P$;

$left := 0$; $right := n$;
$llcp := 0$; $rlcp := 0$;   (* $llcp = lcp(P, S_{left-1})$, $rlcp = lcp(P, S_{right})$ *)
while $left < right$ **do**
    $mid := \lfloor (left + right)/2 \rfloor$;
    $mlcp := \min(llcp, rlcp)$;
    $mlcp := mlcp + lcp(P[mlcp \dots), S_{mid}[mlcp \dots))$;
    if $mlcp = |P|$ tai $P[mlcp] < S_{mid}[mlcp]$ **then**
        $right := mid$; $rlcp := mlcp$;
    else
        $left := mid + 1$; $llcp := mlcp$;
return $right$;

- Hakuaika nopeutuu, mutta pahimmassa tapauksessa se on edelleen $\mathcal{O}(m \log n)$.

- Edellä käytettiin hyväksi sitä, että $lcp(P, S_{left-1})$ ja $lcp(P, S_{right})$ tunnettiin.

- Jos myös $lcp(S_{mid}, S_{left-1})$ ja $lcp(S_{mid}, S_{right})$ tunnetaan, algoritmia voidaan edelleen tehostaa. Ne voidaan laskea etukäteen ja tallettaa taulukkoon.

  – Arvo mid määrää yksikäsitteisesti vastaavat arvot $left$ ja $right$.

  – Olkoon $LCP_l(mid) = lcp(S_{mid}, S_{left-1})$ ja $LCP_r(mid) = lcp(S_{mid}, S_{right})$.

Lemma 3.8.5 *Olkoon $A \leq B, B' \leq C$ merkkijonoja.*

a) Jos $lcp(A, B) > lcp(A, B')$, niin $B < B'$.

b) Jos $lcp(B, C) > lcp(B', C)$, niin $B > B'$.

Todistus. Todistetaan a)-kohta. Kohdan b) todistus on vastaava.

- Tehdään vastaoletus $B \geq B'$. Tällöin $lcp(A, B) = \min(lcp(A, B'), lcp(B', B)) \leq lcp(A, B')$, mikä on ristiriita.

$\square$

- Lemman perusteella jonojen P:n ja S_{mid} keskinäinen järjestys pystytään päättelemään pelkästään lcp-arvoja vertaamalla, jos $lcp(P, S_{left-1}) \neq lcp(S_{mid}, S_{left-1})$ tai $lcp(P, S_{right}) \neq lcp(S_{mid}, S_{right})$.

- Muussa tapauksessa $lcp(P, S_{left-1}) = lcp(S_{mid}, S_{left-1})$ ja $lcp(P, S_{right}) = lcp(S_{mid}, S_{right})$, ja seuraavan lemman perusteella $lcp(P, S_{mid}) \geq \max(lcp(P, S_{left-1}), lcp(P, S_{right}))$.

Lemma 3.8.6 *Olkoon A, B ja C jonoja, joilla $lcp(A, B) = lcp(A, C)$. Tällöin $lcp(B, C) \geq lcp(A, B) = lcp(A, C)$.*

Todistus. Tehdään vastaoletus $lcp(B, C) < lcp(A, B) = lcp(A, C)$. Tällöin mikään A:n, B:n ja C:n keskinäisen järjestyksen valinta ei toteuta lemma 3.8.2 ja seuraa ristiriita. $\square$

Algoritmi 3.8.7 *Merkkijonobinäärihaku (tehostettu)*
Syöte: Järjestetty merkkijonojoukko $\mathcal{R} = \{S_0, S_1, \ldots S_{n-1}\}$, hakujono P.
Taulukot LCP_l ja LCP_r.
Tuloste: i, jolla $S_{i-1} < P \leq S_i$ tai
0, jos $P \leq S_0$; tai n, jos $S_{n-1} < P$;

(1) $left := 0$; $right := n$;
(2) $llcp := 0$; $rlcp := 0$;
(3) **while** $left < right$ **do**
(4) $\quad mid := \lfloor (left + right)/2 \rfloor$;
(5) $\quad$ **if** $llcp < LCP_l(mid)$ **then**
(6) $\quad\quad left := mid + 1$;
(7) $\quad$ **else if** $rlcp < LCP_r(mid)$ **then**
(8) $\quad\quad right := mid$;
(9) $\quad$ **else if** $llcp > LCP_l(mid)$ **then**
(10) $\quad\quad right := mid$; $rlcp := LCP_l(mid)$;
(11) $\quad$ **else if** $rlcp > LCP_r(mid)$ **then**
(12) $\quad\quad left := mid + 1$; $llcp := LCP_r(mid)$;
(13) $\quad$ **else**
(14) $\quad\quad mlcp := \max(llcp, rlcp)$;
(15) $\quad\quad mlcp := mlcp + lcp(P[mlcp \ldots), S_{mid}[mlcp \ldots))$;
(16) $\quad\quad$ **if** $mlcp = |P|$ tai $P[mlcp] < S_{mid}[mlcp]$ **then**
(17) $\quad\quad\quad right := mid$; $rlcp := mlcp$;
(18) $\quad\quad$ **else**
(19) $\quad\quad\quad left := mid + 1$; $llcp := mlcp$;
(20) **return** $right$;

Lause 3.8.8 *Järjestetty merkkijonojoukko $\mathcal{R} = \{S_0, S_1, \ldots, S_{n-1}\}$ voidaan esikäsitellä ajassa $\mathcal{O}(DP(\mathcal{R}))$ ja tilassa $\mathcal{O}(n)$ niin, että binäärihaku jonolla P, $|P| = m$, voidaan toteuttaa ajassa $\mathcal{O}(|P| + \log n)$.*

Todistus.

- Arvot $LCP_l(mid)$ ja $LCP_r(mid)$ voidaan laskea ajassa $dp_{\mathcal{R}}(S_{mid})$. Koko taulukot voidaan siis laskea ajassa $\mathcal{O}(DP(\mathcal{R}))$ ja tallettaa tilaan $\mathcal{O}(n)$.
- **while** -silmukka suoritetaan $\log n$ kertaa. Riviä 15 lukuunottamatta kaikki toimii vakioajassa.
- Jos rivillä 15 $mlcp$ kasvaa k:n verran, aikaa kuluu $\mathcal{O}(k)$. Sen jälkeen joko $rlcp$ tai $llcp$ kasvaa vähintään k:n verran (rivi 17 tai 19).
- Koska $llcp$ ja $rlcp$ eivät koskaan pienene ja ne kasvavat enintään arvoon $|P|$, rivillä 15 kuluva aika koko algoritmissa on $\mathcal{O}(|P|)$.

□

3.9 Lcp-vertailut

- Alkioiden vertailuihin perustuvat järjestämisalgoritmit ja hakurakenteet eivät ole optimaalisia merkkijonoille, koska vertailu voi viedä enemmän kuin vakioajan, mutta tuloksena on kuitenkin vain bitti tai tritti.

- Binäärihakua lukuunottamatta edellä esitetyt, merkkijonoihin erikoistuneet algoritmit ja tietorakenteet ratkaisivat ongelman suorittamalla kahden merkkijonon vertailu yhden merkkiposition perusteella, siis vakioajassa.

- Binäärihaku käyttää toisenlaista tekniikkaa, *lcp-vertailuja*:
  - Kahden merkkijonon vertailu suoritetaan aina loppuun asti.
  - Vertailua ei kuitenkaan aloiteta aina alusta, vaan tunnettu yhteinen alkuosa ohitetaan.
  - Vertailun tuloksena ei ole vain keskinäinen järjestys vaan myös pisimmän yhteisen alkuosan pituus, jota voidaan hyödyntää tulevissa vertailuissa.

- Samaa tekniikkaa voidaan hyödyntää myös muissa algoritmeissa ja tietorakenteissa.

Järjestäminen

- Lomitusjärjestäminen (mergesort) on tunnettu $\mathcal{O}(n \log n)$-ajassa toimiva järjestämisalgoritmi. Sen etuja ovat mm. vakaus ja soveltuvuus linkitetyn listan järjestämiseen.

- Toistuvana perusoperaationa on kahden järjestetyn listan lomitus (yhdistäminen yhdeksi järjestetyksi listaksi).

- Merkkijonoille soveltuva muunnelma saadaan pitämällä kirjaa järjestettyjen listojen peräkkäisten alkioiden pisimmän yhteisen alkuosan pituuksista.

Lause 3.9.1 *Merkkijonojoukon $\mathcal{R}$, $|\mathcal{R}| = n$, lomitusjärjestäminen voidaan toteuttaa ajassa $\mathcal{O}(DP(\mathcal{R}) + n \log n)$.*

Todistus. Tarkastellaan yhtä lomitusaskelta:

- Olkoon T viimeisin tuloslistalle lisätty jono ja olkoon S ja S' syötelistojen kärjessä olevat jonot. Siis $T \leq S, S'$ ja seuraavana tehtävänä on verrata jonoja S ja S'.

- Voidaan olettaa, että $lcp(T, S)$ ja $lcp(T, S')$ tunnetaan. Yksi niistä saadaan suoraan toisesta syötelistasta ja toinen laskettiin edellisessä vertailussa.

- Jos $lcp(T, S) \neq lcp(T, S')$, järjestys saadaan vakioajassa lemmasta 3.8.5.

- Jos $lcp(T, S) = lcp(T, S') = \ell$, ℓ ensimmäistä merkkiä voidaan ohittaa.

- Olkoon $\ell(X)$ jonon X ja sen lähimmän tunnetun edeltäjän pisimmän yhteisen alkuosan pituus. Lomituksen aikana $\ell(X)$ ei koskaan laske, eikä nouse suuremmaksi kuin $dp_{\mathcal{R}}(X)$.

- Jos S:n ja S':n vertailu vaatii k merkkivertailua, joko $\ell(S)$ tai $\ell(S')$ kasvaa $k-1$:llä.

Koska $\sum_{S \in \mathcal{R}} \ell(S) \leq DP(R)$, merkkivertailujen kokonaismäärä on $\mathcal{O}(DP(\mathcal{R}) + n \log n)$. □

- Lomitusjärjestämisen lisäksi lcp-vertailutekniikka soveltuu mm. keko- ja lisäysjärjestämisen merkkijonoversioille.

Hakurakenteet

R. Grossi & G. F: Italiano: Efficient Techniques for Maintaining Multidimensional Keys in Linked Data Structures. Teoksessa *Proc. ICALP'99*, sivut 372–381, 1999.

- Binäärihaku järjestetystä taulukosta vastaa hakua täydellisesti tasapainotetusta binääripuusta.

- Lcp-vertailutekniikkaa voidaan soveltaa myös yleiseen binääripuuhun.

- Oletetaan, että puun solmu i tallettaa jonon S_i. Sen lisäksi se tallettaa lcp-arvot $LCP_l(i) = lcp(S_i, S_<)$ ja $LCP_r(i) = lcp(S_i, S_>)$, missä $S_<$ ja $S_>$ ovat S_i:n lähin edeltäjä ja seuraaja juuresta solmuun i johtavalla polulla.

- Haku toimii kuten binäärihaku, mutta jonojen S_{mid}, S_{left-1} ja S_{right} tilalla on nyt S_i, $S_<$ ja $S_>$.
- Myös lisäys- ja poisto-operaatioissa voidaan hyödyntää lcp-vertailujen tekniikkaa.
- Grossi ja Italiano ovat osoittaneet seuraavan tuloksen.

Lause 3.9.2 *Merkkijonobinääripuu voidaan toteuttaa niin, että jonon P, $|P| = m$, lisäys, poisto ja haku n:n jonon binääripuusta voidaan tehdä ajassa $\mathcal{O}(m + \log n)$.*

- Ternääripuulla on samat operaatioiden aikavaativuudet. Ternääripuun koko on kuitenkin $\mathcal{O}(DP(\mathcal{R}))$, mutta binääripuun koko on vain $\mathcal{O}(n)$.
- Grossi ja Italiano ovat osoittaneet, että sama tekniikka soveltuu kaikille binääripuiden versioille (esim. puna-musta-puu) ja yleistyksille (esim. B-puu).

Luku 4

Loppuosarakenteet

- Olkoon $S = S[0 \ldots n)$ merkkijono aakkostossa Σ.
- Merkitään S_i:llä, $0 \le i \le n$, loppuosaa $S[i \ldots n)$.
- Yleistetään merkintä joukkoihin. Kaikilla $C \subseteq [0, n]$, $S_C = \{S_i \mid i \in C\}$.
- Jonon S *loppuosarakenne* on joukon $S_{[0,n]}$ hakurakenne.

Loppuosarakenteet poikkevat monessakin mielessä tavanomaisista hakurakenteista.

- Loppuosarakenteilla on monia sovelluksia, ehkä tärkeimpänä niistä tarkka hahmonsovitus:
  - Hahmolla P on esiintymä S:ssä alkaen positiosta i, jos ja vain jos P on S_i:n alkuosa.
  - Hahmonsovitus S:ssä voidaan siis toteuttaa alkuosahakuna S:n loppuosarakenteessa.
- Loppuosien joukon kokonaispituus $||S_{[0,n]}||$ on $\Theta(n^2)$. Myös $DP(S_{[0,n]})$ voi olla $\Theta(n^2)$ (esim. $S = XX$).
  - Esim. trie ja ternääripuu soveltuvat huonosti loppuosarakenteeksi. Ne ovat liian isoja.
- Loppuosarakenteilla on omia muodostusalgoritmeja.
  - Kaikkien hakurakenteiden muodostusalgoritmit vaativat ajan $\Theta(DP(\mathcal{R}))$ yleiselle jonojoukolle $\mathcal{R}$.

– Loppuosarakenteille on monia muodostusalgoritmeja, jotka toimivat ajassa $\mathcal{O}(n)$ tai $\mathcal{O}(n \log n)$.

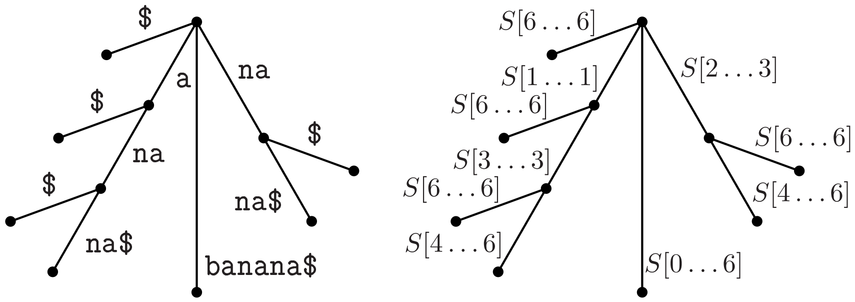
4.1 Loppuosapuu

P. Weiner: Linear Pattern Matching Algorithm. Teoksessa *Proc. 14th Symposium on Switching and Automata Theory*, sivut 1–11, 1973.

- Jonon S *loppuosapuu* (suffix tree) on loppuosajoukon $S_{[0,n]}$ tiivistetty trie.

- Oletetaan, että jonon S ja siis jokaisen loppuosan lopussa on ylimääräinen merkki $S[n] = \texttt{\$}$, missä $\texttt{\$} \notin \Sigma$. Tällöin loppuosia esittävät solmut ovat kaikki lehtiä.

Esimerkki 4.1.1 *Jonon $S =$ `banana$` loppuosapuu.*



Loppuosapuun koko on $\mathcal{O}(n)$.

- Loppumerkin `$` ansiosta puussa on täsmälleen $n+1$ lehteä, yksi kutakin loppuosaa kohden.

- Jokaisessa puussa on vähemmän haarautumissolmuja kuin lehtiä. Koska loppuosapuussa jokainen sisäsolmu on haarautumissolmu, sisäsolmuja on enintään n kappaletta.

- Puissa on aina kaaria yksi vähemmän kuin solmuja. Kaaria on siten enintään $2n - 2$. Kaarten nimet ovat S:n osajaonoja ja ne voidaan esittää vakiotilassa osoittamalla alku- ja loppukohta.

Hahmonsovitus eli alkuosahaku hahmolle P voidaan tehdä ajassa $\mathcal{O}(|P| + occ)$, missä occ on tuloksen koko eli hahmon esiintymien lukumäärä.

Loppuosa-automaatti

- Tiivistetty trie saatiin tavallisesta triestä korvaamalla haurautumattomat polut yhdellä kaarella.
- Toinen mahdollisuus tiivistää trietä erityisesti loppuosatrien tapauksessa saadaan tulkitsemalla trie automaatiksi ja soveltamalla automaatin minimointia. Tämä tarkoittaa isomorfisten alipuiden yhdistämistä.
- Lopputulos on loppuosa-automaatti eli DAWG (directed acyclic word graph).
- Voidaan osoittaan, että loppuosa-automaatin koko on $\mathcal{O}(n)$. Tulos seuraa esimerkiksi loppuosapuun ja -automaatin dualiteetista:
  - Jonon S loppuosapuussa on solmu, joka esittää jonoa X, jos ja vain jos X on S:n loppuosa tai on olemassa kaksi eri merkkiä a ja b, joilla Xa ja Xb ovat S:n osajonoja.
  - Jonon S loppuosa-automaatissa on tila, jonka tunnistamista jonoista pisin on X, jos ja vain jos X on S:n alkuosa tai on olemassa kaksi eri merkkiä a ja b, joilla aX ja bX ovat S:n osajonoja.
- Soveltamalla haarautumattomien polkujen tiivistämistä loppuosa-automaattiin saadaan tiivistetty loppuosa-automaatti, CDAWG (Compact DAWG). Se saadaan myös soveltamalla automaatin minimointia loppuosapuuhun. Kovin merkittävää tilansäästöä näin ei kuitenkaan yleensä saavuteta.
- Useimmat loppuosapuun sovelluksista voidaan toteuttaa myös loppuosa-automaatilla. Keskitytään jatkossa kuitenkin loppuosapuuhun.

4.2 Loppuosapuun muodostaminen

E. M. McCreight: A Space-Economic Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

- Ensimmäinen lineaariaikainen loppuosapuun muodostusalgoritmi oli jo Weinerin loppuosapuun esitelleessä artikkelissa.

- Esitetään tässä kuitenkin McCreightin algoritmi, joka on yksinkertaisempi ja käytännössä tehokkaampi.
- Perusideana on lisätä loppuosat rakenteeseen yksi kerrallaan järjestyksessä S_0, S_1, ...
- Syksyn 2003 muistiinpanoissa on esitetty Ukkosen algoritmi [Ukk95], joka muodostaa loppuosapuun ensin jonolle $S[0\ldots 1)$ sitten jonolle $S[0\ldots 2)$, jne..
- Näennäisestä erilaisuudestaan huolimatta, McCreightin ja Ukkosen algoritmit ovat läheistä sukua.
- Kaikki yllämainitut algoritmit toimivat lineaarisessa ajassa vakioaakkostolla. Farachin algoritmi [Far97] toimii lineaarisessa ajassa, kun merkit ovat kokonaislukuja lineaariselta osaväliltä. Farachin algoritmi on erittäin monimutkainen, mutta siihen pohjautuva yksinkertaisempi loppuosataulukon muodostusalgoritmi nähdään myöhemmin.

Loppuosalinkit

- McCreightin algoritmissa (ja kaikissa edellämainituissa algoritmeissa) keskeisessä osassa ovat *loppuosalinkit*.
- Jos solmu u esittää jonoa $S[i\ldots j]$ ja solmu v jonoa $S[i+1\ldots j]$, niin u:sta on loppuosalinkki v:hen, merkitään $\mathsf{slink}(u) = v$.
- Loppuosalinkki on hyvin määritelty kaikille solmuille paitsi juurelle.

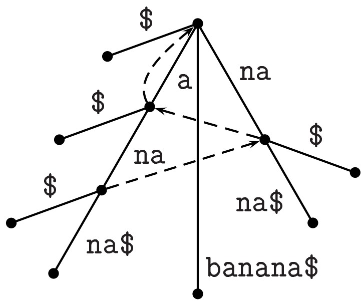
Lemma 4.2.1 *Jos jonon S loppuosapuussa on solmu u, joka esittää jonoa $S[i\ldots j]$, $0 \leq i \leq j \leq n$, niin siinä on myös solmu v, joka esittää jonoa $S[i+1\ldots j]$.*

Todistus. Jos u on lehti, väite on selvä.

- Jos u ei ole lehti, sillä on ainakin kaksi lasta. Olkoon a ja b kaksi lapsiin johtavien kaarten alkumerkkiä. Tällöin $S[i\ldots j]a$ ja $S[i\ldots j]b$ esiintyvät tekstissä S.
- Siis myös $S[i+1\ldots j]a$ ja $S[i+1\ldots j]b$ esiintyvät tekstissä S, joten loppuosapuussa täytää olla haarautumissolmu, joka esittää jonoa $S[i+1\ldots j]$.

□

Esimerkki 4.2.2 *Jonon S = `banana$` loppuosapuu. Sisäsolmujen loppuosalinkit on esitetty katkoviivoilla.*



- Loppuosapuuta voidaan myös käyttää loppuosien joukon Aho-Corasick-automaattina. Loppuosalinkit toimivat tällöin korjauslinkkeinä.

- Aiemmin mainittiin, että loppuosapuun solmut ja käänteisjonon loppuosa-automaatin tilat vastaavat toisiaan. Tällöin loppuosalinkit vastaavat automaatin tilasiirtymiä. Myös loppuosa-automaatissa on loppuosalinkkejä, jotka taas vastaavat käänteisjonon loppuosapuun varsinaisia kaaria.

Triviaalialgoritmi

Esitetään loppuosapuu seuraavasti:

- $\mathsf{child}(u, a)$ on u:n lapsi v, johon u:sta menevän kaaren nimen ensimmäinen merkki on a. Jos tällaista lasta ei ole, $\mathsf{child}(u, a) = \perp$.

- $\mathsf{parent}(u)$ on solmun u vanhempi.

- $\mathsf{start}(u)$ on solmun u esittämän osajonon alkukohta ja $\mathsf{depth}(u)$ on u:n syvyys eli sen esittämän jonon pituus. Siis u esittää jonoa $S[\mathsf{start}(u) \ldots \mathsf{start}(u) + \mathsf{depth}(u))$.

Triviaalialgoritmi muodostaa loppuosapuun lisäämällä loppuosat yksi kerrallaan.

Algoritmi 4.2.3 *Triviaali loppuosapuun muodostus*
Syöte: Merkkijono $S = S[0, n]$, $S[n] = \$$.
Tuloste: S:n loppuosapuu.

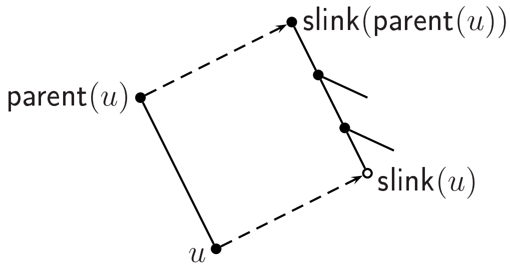
luo juurisolmu r; $\mathsf{depth}(r) := 0$;
$u := r$; (* sijainti puussa on solmussa u tai siihen johtavalla kaarella *)
$d := 0$; (* syvyys *)
for $i := 0$ **to** n **do** (* Lisätään loppuosa S_i *)
  (* invariantti: $S[i \ldots i + d) = S[\mathsf{start}(u) \ldots \mathsf{start}(u) + d)$ *)
  while $d = \mathsf{depth}(u)$ **and** $\mathsf{child}(u, S[i + d]) \neq \perp$ **do**
    $u := \mathsf{child}(u, S[i + d])$; $d := d + 1$;
    while $d < \mathsf{depth}(u)$ **and** $S[\mathsf{start}(u) + d] = S[i + d]$ **do** $d := d + 1$;
  if $d < \mathsf{depth}(u)$ **then** (* pysäys keskelle kaarta *)
    luo uusi solmu v;
    $\mathsf{start}(v) := i$; $\mathsf{depth}(v) := d$;
    $p := \mathsf{parent}(u)$;
    $\mathsf{child}(v, S[\mathsf{start}(u) + d]) := u$; $\mathsf{parent}(u) := v$;
    $\mathsf{child}(p, S[i + \mathsf{depth}(p)]) := v$; $\mathsf{parent}(v) := p$;
    $u := v$;
  luo uusi lehti w; (* w edustaa loppuosaa S_i *)
  $\mathsf{start}(w) := i$; $\mathsf{depth}(w) := n - i + 1$;
  $\mathsf{child}(u, S[i + d]) := w$; $\mathsf{parent}(w) := u$;
  $d := 0$; $u := r$;

Loppuosalinkkien laskeminen

- Tarkastellaan triviaalialgoritmissa tilannetta, jossa loppuosaa S_i edustava lehti on juuri lisätty solmun u lapseksi.
- Oletetaan, että loppuosalinkki on olemassa kaikille sisäsolmuille paitsi mahdollisesti u:lle. Jos solmulla u ei ole loppuosalinkkiä, se voidaan laskea seuraavalla proseduurilla.

  Prosedure Set-slink(u)
  $v := \mathsf{slink}(\mathsf{parent}(u))$;
  while $\mathsf{depth}(v) < \mathsf{depth}(u) - 1$ **do**
    $v := \mathsf{child}(v, S[\mathsf{start}(u) + 1 + \mathsf{depth}(v)])$;
  if $\mathsf{depth}(v) > \mathsf{depth}(u) - 1$ **then**
    luo uusi solmu w;
    $\mathsf{start}(w) := \mathsf{start}(u) + 1$; $\mathsf{depth}(w) := \mathsf{depth}(u) - 1$;
    $p := \mathsf{parent}(v)$;
    $\mathsf{child}(w, S[\mathsf{start}(v) + \mathsf{depth}(w)]) := v$; $\mathsf{parent}(v) := w$;
    $\mathsf{child}(p, S[\mathsf{start}(w) + \mathsf{depth}(p)]) := w$; $\mathsf{parent}(w) := p$;
    $v := w$;
  $\mathsf{slink}(u) := v$;

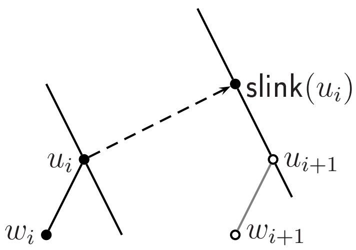
- Proseduuri etsii oikean kohdan puusta u:n vanhemman loppuosalinkkiä käyttäen ja lisää siihen tarvittaessa uuden solmun.



- Huomaa, että algoritmi ei tutki kaikkia merkkejä polulla solmusta $\mathsf{slink}(\mathsf{parent}(u))$ solmuun $\mathsf{slink}(u)$, vaan ainoastaan haaraumakohtien merkit. Tämä toimii oikein:
  - Olkoon $d = \mathsf{depth}(u)$. Solmu u on haaraumasolmu, joten $S[i \ldots i+d) = S[j \ldots j+d)$ jollakin $j < i$.
  - Siten polulla lehteen, joka edustaa loppuosaa S_{j+1}, täytyy olla solmun $\mathsf{slink}(u)$ esittämää jonoa $S[i+1 \ldots i+d) = S[j+1 \ldots j+d)$ vastaava kohta.
  - Koska oikea kohta on olemassa, sen löytämiseksi riittää valita oikea haara haaraumakohdissa.

McCreightin algoritmi

- Tarkastellaan triviaalialgoritmissa tilannetta, jossa loppuosaa S_i edustava lehti w_i on juuri lisätty solmun u_i lapseksi.
- Seuraavaksi pitää lisätä lehti w_{i+1} solmun u_{i+1} lapseksi Triviaalialgoritmi etsii solmun u_{i+1} tai sitä vastaavan kohdan juuresta alkaen.
- McCreightin algoritmi oikaisee käyttäen loppuosalinkkiä.



Algoritmi 4.2.4 *McCreight*

Syöte: Merkkijono $S = S[0, n]$, $S[n] = \$$.
Tuloste: S:n loppuosapuu.

```
(1)  luo juurisolmu r; depth(r) := 0; slink(r) := r
(2)  u := r;          (* sijainti puussa on solmussa u tai siihen johtavalla kaarella *)
(3)  d := 0;          (* syvyys *)
(4)  for i := 0 to n do        (* Lisätään loppuosa S_i *)
         (* invariantti: S[i...i+d) = S[start(u)...start(u)+d) *)
(5)      while d = depth(u) and child(u, S[i+d]) ≠ ⊥ do
(6)          u := child(u, S[i+d]); d := d + 1;
(7)          while d < depth(u) and S[start(u)+d] = S[i+d] do d := d + 1;
(8)      if  d < depth(u) then        (* pysäys keskelle kaarta *)
(9)          luo uusi solmu v;
(10)         start(v) := i; depth(v) := d; slink(v) := ⊥;
(11)         p := parent(u);
(12)         child(v, S[start(u)+d]) := u; parent(u) := v;
(13)         child(p, S[i+depth(p)]) := v; parent(v) := p;
(14)         u := v;
(15)     luo uusi lehti w;      (* w edustaa loppuosaa S_i *)
(16)     start(w) := i; depth(w) := n − i + 1;
(17)     child(u, S[i+d]) := w; parent(w) := u;
(18)     if slink(u) = ⊥ then Set-slink(u);
(19)     d := d − 1; u := slink(u);
```

Lause 4.2.5 *Olkoon S, $|S| = n$, merkkijono vakioaakkostossa. McCreightin algoritmi muodostaa S:n loppuosapuun ajassa $\mathcal{O}(n)$.*

Todistus.

- Yhden loppuosa lisäys vie vakioajan lukuunottamatta rivejä 5–7, jossa kuljetaan solmusta $\mathsf{slink}(u_i)$ solmuun u_{i+1}, ja proseduurin Set-$\mathsf{slink}(u)$ kutsua.

- Jokainen askel riveillä 5–7 kasvattaa d:tä. Ainoa kohta, jossa d pienenee on rivi 19 ja sielläkin vain yhdellä (toisin kuin triviaalialgoritmissa). Koska d ei voi kasvaa suuremmaksi kuin n, on riveillä 5–7 kuluva aika yhteensä $\mathcal{O}(n)$.

- Proseduuri Set-$\mathsf{slink}(u)$ toimii vakioajassa lukuunottamatta rivien 2 ja 3 silmukkaa. Olkoon d' kulloisenkin solmun u vanhemman syvyys. d' voi pienentyä rivillä 19 loppuosalinkkiä seurattaessa mutta enintään yhdellä. Jos kuitenkin seurattu loppuosalinkki luotiin juuri edellisen rivin proseduurikutsulla ja kutsun aikana proseduurin silmukka suoritettiin $k > 0$ kertaa, d' ei pienene vaan kasvaa $k - 1$:llä. Siten proseduurin silmukka suoritetaan enintään $\mathcal{O}(n)$ kertaa.

□

4.3 Loppuosapuun sovelluksia

Aiemmin todettiin jo, että loppuosapuu mahdollistaa nopean hahmonsovituksen.

- Vaikka itse loppuosapuun muodostus laskettaisiin, on tarkka hahmonsovitus asymptoottisesti yhtä nopeaa kuin Knuth–Morris–Pratt-algoritmilla yhden hahmon tapauksessa tai Aho–Corasick-algoritmilla monen hahmon tapauksessa.

- Varsinainen hyöty saadaan, jos loppuosapuu voidaan muodostaa etukäteen.

- Loppuosapuu soveltuu myös likimääräiseen hakuun käyttäen seulontamenetelmiä, esimerkiksi luvussa 2.8 kuvattua menetelmää, joka perustuu hahmon jakoon $k + 1$ palaseen ja palasten tarkkaan hahmonsovitukseen.

Loppuosapuu paljastaa myös paljon tietoa jonon S sisäisestä rakenteesta.

- Jonon S erilaisten osajonojen lukumäärä on täsmälleen loppuosatrien (siis loppuosien joukon tiivistämättömän trien) solmujen lukumäärä. Loppuosapuusta tämä luku saadaan laskemalla yhteen kaarten nimien pituudet ja lisäämällä yksi (juuri). Aikavaatimus on $\mathcal{O}(n)$, vaikka tulos on yleisesti $\Theta(n^2)$.

- Merkkijonon S pisin toistuva osajono on pisin jono, joka esiintyy vähintään kahdesti S:ssä. Loppuosapuun syvin sisäsolmu esittää pisintä toistuvaa osajonoa.

Loppuosapuu soveltuu myös kahden tai useamman jonon vertailemiseen. Esimerkiksi kahden jonon S ja T *pisin yhteinen osajono* löydetään lineaarisessa ajassa ainakin seuraavilla kahdella tavalla: