- Merkkijonopikajärjestäminen on siis optimaalinen merkkivertailuihin perustuva merkkijonojen järjestämisalgoritmi.
- Se on myös käytännössä nopea.

3.3 Kantalukujärjestäminen

- Kokonaislukuja on mahdollista järjestää alarajaa $\Omega(n \log n)$ nopeammin, jopa lineaarisessa ajassa.
- Jos aakkosto koostuu kokonaisluvuista, myös merkkijonoja voi järjestää alarajaa $\Omega(DP(\mathcal{R}) + n \log n)$ nopeammin.
- $\Omega(DP(\mathcal{R}))$ on kuitenkin aina alaraja, kun merkkejä on käsiteltävä yksittäin.

Laskemisjärjestäminen

- Laskemisjärjestäminen järjestää pienet kokonaisluvut lineaarisessa ajassa.

Algoritmi 3.3.1 *Laskemisjärjestäminen*

Syöte: (Moni)joukko $R = \{k_0, k_1, \ldots k_{n-1}\}$ kokonaislukuja väliltä $[0, \sigma)$.
Tuloste: Joukko R nousevassa järjestyksessä taulukossa $J[0 \ldots n)$.

```
(1) for i := 0 to σ − 1 do C[i] := 0;
(2) for i := 0 to n − 1 do C[k_i] := C[k_i] + 1;
(3) sum := 0;
(4) for i := 1 to σ − 1 do
(5)       tmp := C[i]; C[i] := sum; sum := sum + tmp;
(6) for i := 0 to n − 1 do
(7)       J[C[k_i]] := k_i;
(8)       C[k_i] := C[k_i] + 1;
(9) return J;
```

- Aika- ja tilavaatimus ovat $\mathcal{O}(n + \sigma)$.
- Algoritmi 3.3.1 on *vakaa* eli se säilyttää samanarvoisten alkioiden keskinäisen järjestyksen.

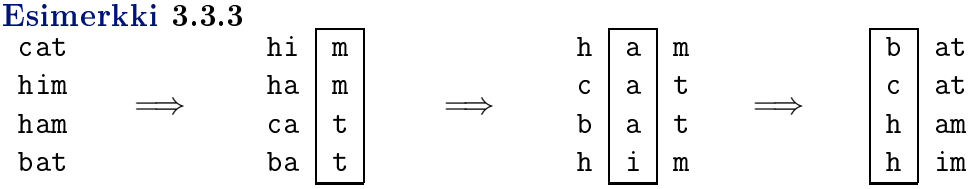
Lopusta alkuun kantalukujärjestäminen

- Kantalukujärjestämisessä (radix sort) laskemisjärjestämistä sovelletaan yhteen positioon kerrallaan.
- Olkoon Laskemisjärjestä($\mathcal{R}$,ℓ) kutsu, joka järjestää merkkijonojoukon $\mathcal{R}$ position ℓ perusteella käyttäen laskemisjärjestämistä.
- Viimeisestä positiosta alkuun päin etenevä järjestäminen on erittäin yksinkertainen.

Algoritmi 3.3.2 *Lopusta alkuun kantalukujärjestäminen*

Syöte: Joukko $\mathcal{R} = \{S_0, S_1, \ldots S_{n-1}\}$ m:n pituisia merkkijonoja aakkostossa $[0, \sigma)$.
Tuloste: Joukko $\mathcal{R}$ aakkosjärjestyksessä
(1) **for** $\ell := m - 1$ **to** 0 **do** Laskemisjärjestä($\mathcal{R}$,ℓ);
(2) **return** $\mathcal{R}$;

Esimerkki 3.3.3



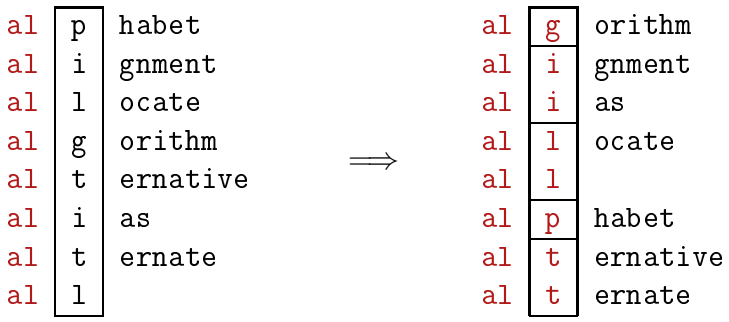
- Algoritmissa oletettiin, että kaikki jonot ovat samanpituisia. Se on kuitenkin helppo yleistää eripituisille merkkijonoille (HT).

Lause 3.3.4 *Olkoon $\mathcal{R} = \{S_0, S_1, \ldots S_{n-1}\}$ joukko merkkijonoja aakkostossa $[0, \sigma)$. Lopusta alkuun kantalukujärjestäminen järjestää $\mathcal{R}$:n ajassa $\mathcal{O}(||\mathcal{R}|| + m\sigma)$ ja tilassa $\mathcal{O}(n + \sigma)$, missä $||\mathcal{R}||$ on merkkijonojen kokonaispituus ja m on pisimmän jonon pituus.*

- Algoritmin heikkous on, että se tutkii merkkijonot kokonaan eli vie $\Omega(||\mathcal{R}||)$ ajan silloinkin, kun $DP(\mathcal{R})$ on paljon pienempi kuin $||\mathcal{R}||$.
- Se soveltuukin lähinnä lyhyille samanmittaisille merkkijonoille.
- Tärkeä sovellusalue on myös kokonaislukujen järjestäminen esittämällä ne σ-kantajärjestelmän jonoina.

Alusta loppuun kantalukujärjestäminen

- Kantalukujärjestämisen voi myös aloittaa alkupäästä. Tällöin tarvitaan kuitenkin hieman monimutkaisempaa kirjanpitoa.
- Tuloksena oleva algoritmi on itseasiassa paljon merkkijonopikajärjestämisen kaltainen. Kolmeenjaon sijasta tehdäänkin nyt jako σ:aan osaan.



- Oletetaan nyt, että Laskemisjärjestä($\mathcal{R}$,ℓ) palauttaa joukot $\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_{\sigma-1}$, missä $\mathcal{R}_i = \{S \in \mathcal{R} \mid S[\ell] = i\}$.
- Rekursiiviset kutsut johtavat lopulta hyvin pieniin joukkoihin. Laskemisjärjestäminen kuluttaa kuitenkin aina ajan $\Omega(\sigma)$. Algoritmin nopeuttamiseksi alle σ:n kokoisiin joukkoihin käytetään merkkijonopikajärjestämistä.

Algoritmi 3.3.5 *Kantalukujärjestä($\mathcal{R}$,ℓ)*

Syöte: Joukko $\mathcal{R}$ merkkijonoja ja niiden yhteisen alkuosan pituus ℓ.
Tuloste: Joukko $\mathcal{R}$ aakkosjärjestyksessä.

(1) **if** $|\mathcal{R}| < \sigma$ **then return** Merkkijonopikajärjestä($\mathcal{R}$,ℓ);
(2) $\mathcal{R}_\perp := \{S \in \mathcal{R} \mid |S| = \ell\}$; $\mathcal{R} := \mathcal{R} \setminus \mathcal{R}_\perp$
(3) $(\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_{\sigma-1}) :=$ Laskemisjärjestä($\mathcal{R}$,ℓ);
(4) **for** $i := 0$ **to** $\sigma - 1$ **do** Kantalukujärjestä($\mathcal{R}_i$,$\ell + 1$);
(5) **return** $\mathcal{R}_\perp \cdot \mathcal{R}_0 \cdot \mathcal{R}_1 \cdots \mathcal{R}_{\sigma-1}$;

Lause 3.3.6 *Olkoon $\mathcal{R} = \{S_0, S_1, \ldots S_{n-1}\}$ joukko merkkijonoja aakkostossa $[0, \sigma)$. Alusta loppuun kantalukujärjestäminen järjestää $\mathcal{R}$:n ajassa $\mathcal{O}(DP(\mathcal{R}) + n \log \sigma)$ ja tilassa $\mathcal{O}(n + \sigma)$.*

Todistus.

- Tarkastellaan aikavaatimusta, kun algoritmia kutsutaan osajoukolla $\mathcal{R}'$, $|\mathcal{R}'| = k$.

- Kun $k \geq \sigma$, ei-rekursiiviset osat, mukaanlukien laskemisjärjestäminen, toimivat ajassa $\mathcal{O}(k)$. Samalla tutkitaan k merkkiä ja samoja merkkejä ei tutkita enää uudelleen.
- Yksilöivien alkuosien ulkopuolisia merkkejä ei tutkita. Siis merkkijonopikajärjestäminen poisluettuna aikavaatimus on yhteensä $\mathcal{O}(DP(\mathcal{R}))$.
- Merkkijonopikajärjestämistä kutsutaan vain σ:a pienemmille joukoille ja kukin jono on mukana enintään yhdessä kutsussa. Siten merkkijonopikajärjestämisessä kuluva kokonaisaika on $\mathcal{O}(DP(\mathcal{R}) + n \log \sigma)$.

□

- Jos σ on vakio, algoritmi toimii optimaalisessa ajassa $\mathcal{O}(DP(\mathcal{R}))$.
- Käytännössä, tavuaakkostolla, se on suunnilleen yhtä nopea kuin merkkijonopikajärjestäminen.
- Kaksivaiheinen kantalukujärjestäminen (Paige ja Tarjan, SIAM J. Comput. 16(6):973–989, 1987) pääsee aikavaatimukseen $\mathcal{O}(DP(\mathcal{R})+\sigma)$.

3.4 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.

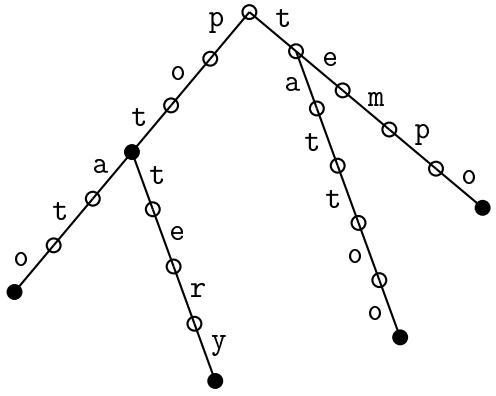
3.5 Trie

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

Edward Fredkin: Trie Memory. *Communications of the ACM*, 3(9):490-499, 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 3.5.1 *Joukon* $\mathcal{R} = \{pot, potato, pottery, tattoo, tempo\}$ *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 $\mathcal{O}(|S|)$.

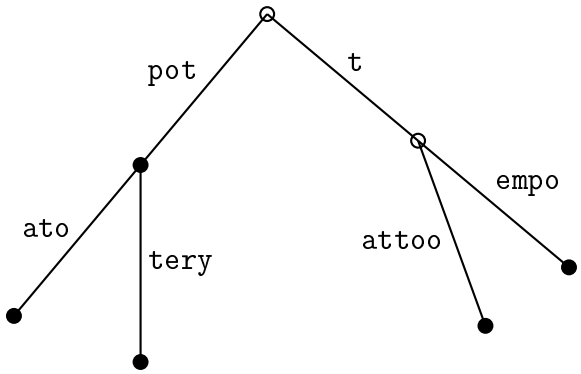
- Jos aakkoston koko ei ole vakio, täytyy määritellä, miten haarautuminen solmussa toteutetaan.

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



- Kunkin kaaren nimi voidaan esittää vakiotilassa osoittimilla merkkijonoihin. Tällöin tilavaatimus on $\mathcal{O}(|\mathcal{R}|)$.

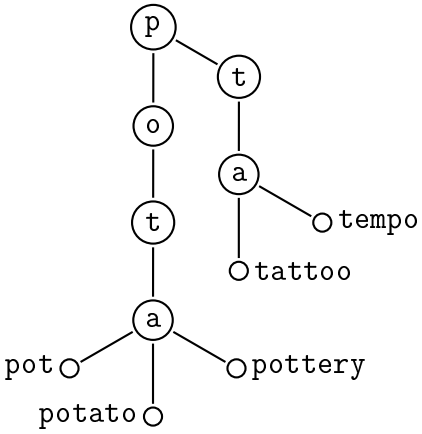
- Alkuosahaku jonolla S vie nyt ajan $\mathcal{O}(|S| + occ)$, missä occ on vastauksena saatujen merkkijonojen lukumäärä (ei yhteispituus).

3.6 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. Juuressa positio on 0. Syvemmälle mentäessä positio kasvaa aina keskihaaraa kuljettaessa.
- Ternääripuun suhde binääripuuhun on siis sama kuin merkkijonopikajärjestämisen suhde tavalliseen pikajärjestämiseen. (Trietä taas voisi verrata kantalukujärjestämiseen.)

Esimerkki 3.6.1 *Joukon* $\mathcal{R} = \{pot, potato, pottery, tattoo, tempo\}$ *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 $\mathcal{O}(DP(\mathcal{R}))$.

3.7 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 3.7.1 *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 ≤ k_{right} *)
(2)  while left < right do
(3)      mid := ⌊(left + right)/2⌋;
(4)      if x ≤ 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 3.7.2 *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 3.7.3 *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 3.7.2 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 $\mathcal{R} = \{S_0, S_1, \ldots S_{n-1}\}$.
- Tällöin $S_{left-1} < P \leq S_{rigth}$. Toisaalta $S_{left-1} \leq S_{mid} \leq S_{right}$, joten lemman 3.7.3 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 $X[i\ldots)$ tarkoittaa jonon X positiosta i alkavaa loppuosaa).

Algoritmi 3.7.4 *Merkkijonobinäärihaku*

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

```
(1)  left := 0; right := n;
(2)  llcp := 0; rlcp := 0;          (* llcp = lcp(P, S_left-1), rlcp = lcp(P, S_right) *)
(3)  while left < right do
(4)      mid := ⌊(left + right)/2⌋;
(5)      mlcp := min(llcp, rlcp);
(6)      mlcp := mlcp + lcp(P[mlcp...), S_mid[mlcp...));
(7)      if mlcp = |P| tai P[mlcp] < S_mid[mlcp] then
(8)          right := mid; rlcp := mlcp;
(9)      else
(10)         left := mid + 1; llcp := mlcp;
(11) 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 $LLCP[mid] = lcp(S_{mid}, S_{left-1})$ ja $RLCP[mid] = lcp(S_{mid}, S_{right})$.

Lemma 3.7.5 *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.

□

- 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 3.7.6 *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 lemmaa 3.7.2 ja seuraa ristiriita. □

Algoritmi 3.7.7 *Merkkijonobinäärihaku (tehostettu)*

Syöte: Järjestetty merkkijonojoukko $\mathcal{R} = \{S_0, S_1, \ldots S_{n-1}\}$, hakujono P.
Taulukot $LLCP$ ja $RLCP$.
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;          (* llcp = lcp(P, S_{left-1}), rlcp = lcp(P, S_right) *)
(3)  while left < right do
(4)      mid := ⌊(left + right)/2⌋;
(5)      if llcp < LLCP[mid] then
(6)          left := mid + 1;
(7)      else if rlcp < RLCP[mid] then
(8)          right := mid;
(9)      else if llcp > LLCP[mid] then
(10)         right := mid; rlcp := LLCP[mid];
(11)     else if rlcp > RLCP[mid] then
(12)         left := mid + 1; llcp := RLCP[mid];
(13)     else
(14)         mlcp := max(llcp, rlcp);
(15)         mlcp := mlcp + lcp(P[mlcp...), S_mid[mlcp...));
(16)         if mlcp = |P| tai P[mlcp] < S_mid[mlcp] then
(17)             right := mid; rlcp := mlcp;
(18)         else
(19)             left := mid + 1; llcp := mlcp;
(20) return right;
```

Lause 3.7.8 *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 $\mathcal{O}(m + \log n)$.*

Todistus.

- Arvot $LLCP[mid]$ ja $RLCP[mid]$ 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 $\mathcal{O}(|P|)$.

□

3.8 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 3.8.1 *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 3.7.5.
- 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 $\sum_{S \in \mathcal{R}} \ell(S) \leq DP(\mathcal{R})$, 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 $LLCP[i] = lcp(S_i, S_<)$ ja $RLCP[i] = lcp(S_i, S_>)$, 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 3.8.2 *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, $0 \leq i \leq n$, 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 $S_{[0,n]} = \{S_0, S_1, \ldots, S_n\}$ hakurakenne.

Loppuosarakenteet poikkevat monessakin mielessä tavanomaisista hakurakenteista.

- Loppuosarakenteilla on monia sovelluksia, keskeisimpänä 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 $\Omega(DP(\mathcal{R}))$ yleiselle jonojoukolle $\mathcal{R}$.

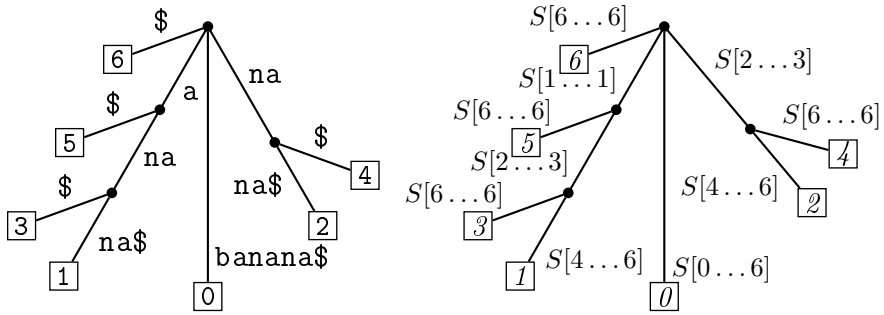
– Loppuosarakenteille on monia muodostusalgoritmeja, jotka toimivat ajassa $\mathcal{O}(n)$ tai $\mathcal{O}(n \log n)$.

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 $S[n] = \$$, missä $\$ \notin \Sigma$. 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 (vakioaakkostolla) ajassa $\mathcal{O}(|P| + occ)$, missä occ on tuloksen koko eli hahmon esiintymien lukumäärä.

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 (Algorithmica 14(3):249–260, 1995), 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 (FOCS, 137–143, 1997) 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.

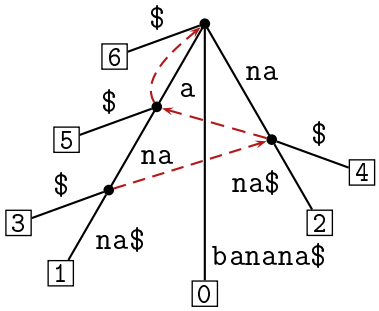
Lemma 4.2.1 *Jos jonon S loppuosapuussa on solmu u, joka esittää jonoa $S[i \ldots j]$, $0 \leq i \leq j \leq n$, niin siinä on myös solmu v, joka esittää jonoa $S[i+1 \ldots j]$.*

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 loppuosalinkeillä.*



- Loppuosapuuta voidaan myös käyttää loppuosien joukon Aho-Corasick-automaattina. Loppuosalinkit toimivat tällöin korjaussiirtyminä. (ks. Luku 4.3)

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.

```
(1)  luo juurisolmu r; depth(r) := 0
(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
(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)     d := 0; u := r
```

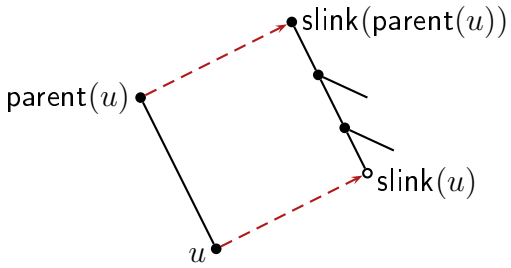
Loppuosalinkkien laskeminen

Triviaalialgoritmi ei laske loppuosalinkkejä, mutta ne voidaan modostaa kutsumalla seuraavaa proseduuria jokaiselle solmulle.

Proseduuri Set-slink(u)

```
(1)  v := slink(parent(u))
(2)  while depth(v) < depth(u) - 1 do
(3)      v := child(v, S[start(u)+1+depth(v)])
(4)  if depth(v) > depth(u) - 1 then
(5)      luo uusi solmu w
(6)      start(w) := start(u)+1; depth(w) := depth(u)-1
(7)      p := parent(v)
(8)      child(w, S[start(v)+depth(w)]) := v; parent(v) := w
(9)      child(p, S[start(w)+depth(p)]) := w; parent(w) := p
(10)     v := w
(11) slink(u) := v
```

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



- Solmun u vanhemman loppuosalinkki täytyy laskea ennen u:n loppuosalinkkiä. Muuten laskentajärjestys on vapaa.

- Rivin 4 tarkistusta ja rivien 5–10 uuden solmun luontia ei tarvita, jos proseduuria kutsutaan valmiissa puussa, koska lemman 4.2.1 perusteella linkin kohdesolmu on aina olemassa. Nämä rivit kuitenkin mahdollistavat proseduurin kutsun myös triviaalialgoritmin aikana esimerkiksi heti u:n luomisen jälkeen.

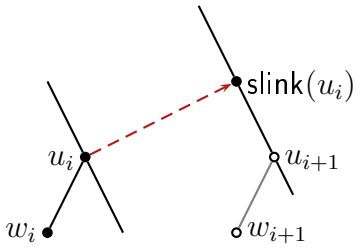
- Huomaa, että algoritmi tutkii vain haaraumakohtien merkit polulla solmusta $\mathsf{slink}(\mathsf{parent}(u))$ solmuun $\mathsf{slink}(u)$.
  - Tämä toimii oikein, jos $v = \mathsf{slink}(u)$ on olemassa tai ainakin kaari, johon se luodaan, on olemassa.
  - Näin on myös triviaalialgoritmin suorituksen aikana: Koska u on haaraumasolmu, sen kautta kulkee polku ainakin kahteen lehteen S_i ja S_j. Tällöin v:n kautta kulkee polku lehtiin S_{i+1} ja S_{j+1}, joista ainakin toinen on jo puussa. (Jos toinen puuttuu, se lisätään heti seuraavaksi.)

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

Esitetään seuraavaksi joitakin esimerkkejä loppuosapuiden monista sovelluksista.

Hahmonsovitus

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.

Loppuosapuuta käytetään myös likimääräiseen hahmonsovitukseen.

- Useat pahimmassa tapauksessa $\mathcal{O}(kn)$ ajassa toimivat algoritmit perustuvat loppuosapuuhun.

- Luvussa 2.5 kuvattiin seulontamenetelmä, joka perustuu hahmon jakoon $k+1$ palaseen ja palasten tarkkaan hahmonsovitukseen. Palasten esiintymät löydetään nopeasti loppuosapuun avulla.

- Voidaan myös muodostaa hahmon k-ympäristö eli kaikki jonot, joiden etäisyys hahmosta on enintään k, ja etsiä ne loppuosapuun avulla.

Merkkijonon sisäinen rakenne

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/tyhjä jono). 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.

Yleistetty loppuosapuu

Loppuosapuu soveltuu myös kahden tai useamman jonon vertailemiseen.

- Kahden merkkijonon S ja T *yleistetty loppuosapuu* on jonon $S£T\$$ loppuosapuu. Huomaa, että S:n alueelta alkavien loppuosien yksilöivä alkuosa päättyy erotinmerkkiin £. Usemman jonon tapauksessa kaikkien erotinmerkkien täytyy olla erilaisia. Yksittäisen merkin sijasta erottimina voi olla myös jonoja, jotka takaavat, että loppuosien yksilöivät alkuosat eivät ulotu erottimien yli.

- Merkitään jokaiseen lehteen, alkaako vastaava loppuosa S:stä vai T:stä. Syvyyssuuntaisella läpikäynnillä voidaan jokaiselle sisäsolmulle laskea, sisältääkö sen alipuu S-lehtiä, T-lehtiä vai molempia.

- Nyt syvin sisäsolmu, jonka alipuu sisältää molempia, esittää S:n ja T:n *pisintä yhteistä osajonoa*.

- Siis pisin yhteinen osajono voidaan laskea lineaarisessa ajassa.

Loppuosapuu Aho-Corasick-automaattina

- Jonon S loppuosalinkeillä varustettu loppuosatrie on Aho-Corasick-automaatti, kun loppuosalinkit toimivat korjaussiirtyminä.
- Myös loppuosapuuta voidaan käyttää AC-automaattina. Korjaussiirtymä keskeltä kaarta tehdään kuten proseduurissa Set-slink() (mutta luomatta uusia solmuja). Tämän operaation tasattu aikavaatimus on $\mathcal{O}(1)$ samalla argumentilla kuin loppuosapuun muodostusalgoritmissa.
- Muodostetaan jonon S loppuosapuu ja selataan sillä jonoa T kuten AC-automaatilla.
- Jos ennen merkin $T[j]$ lukemista ollaan syvyydellä d, se tarkoittaa, että jollakin i, $S[i \ldots i+d) = T[j-d \ldots j)$. Jos $T[j]$ aiheuttaa korjaussiirtymän, $T[j-d \ldots j)$ on pisin T:n positiosta $j-d$ alkava osajono, joka esiintyy S:ssä.
- Korjaussiirtymä kasvattaa $j-d$:tä yhdellä ja siirtymä puussa alaspäin ei muuta $j-d$:tä. Siis selauksen aikana saadaan kaikkien T:n loppuosien pisimmät alkuosat, jotka esiintyvät S:ssä. Tämä on ns. *täsmäystilasto*.
- Täsmäystilastosta saadaan helposti mm. pisin yhteinen osajono. Sitä käytetään myös esim. eräissä $\mathcal{O}(kn)$ ajassa toimivissa likimääräisen hahmonsovituksen algoritmeissa.

LCA-esikäsittely

Loppuosapuun lisäkäsittelyllä saadaan vielä lisää sovelluksia.

- Kahden solmun u ja v *lähin (alin/syvin) yhteinen esi-isä* (LCA, lowest common ancestor, tai NCA, nearest common ancestor) on syvin solmu, joka on sekä u:n että v:n esi-isä. Mikä tahansa puu on mahdollista esikäsitellä siten, että kahden solmun lähin yhteinen esi-isä löydetään vakioajassa. Ohitetaan tässä yksityiskohdat, mutta myöhemmin esitetään vastaava tulos loppuosataulukolle tarkemmin.
- Loppuosapuussa kahden lehden lähin yhteinen esi-isä esittää lehtien edustamien loppuosien *pisintä yhteistä alkuosaa*. Se voidaan siis laskea vakioajassa mille tahansa loppuosille, kun loppuosapuu on LCA-esikäsitelty.