Luku 24

Lisää segmenttipuusta

Segmenttipuu on monipuolinen tietorakenne, joka mahdollistaa monenlaisten kyselyiden toteuttamisen tehokkaasti. Tähän mennessä olemme käyttäneet kuitenkin segmenttipuuta melko rajoittuneesti. Nyt on aika tutustua pintaa syvemmältä segmenttipuun mahdollisuuksiin.

24.1 Kulku puussa

Tähän mennessä olemme kulkeneet segmenttipuuta alhaalta ylöspäin lehdistä juureen. Vaihtoehtoinen tapa toteuttaa puun käsittely on kulkea ylhäältä alaspäin juuresta lehtiin. Tämä kulkusuunta on usein kätevä silloin, kun kyseessä on perustilannetta monimutkaisempi segmenttipuu.

Esimerkiksi välin $[a,b]$ summan laskeminen segmenttipuussa tapahtuu alhaalta ylöspäin tuttuun tapaan näin (luku 8.4):

```
int summa(int a, int b) {
    a += N; b += N;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += p[a++];
        if (b%2 == 0) s += p[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

Ylhäältä alaspäin toteutettuna funktiosta tulee:

```
int summa(int a, int b, int k, int c, int d) {
    if (a > d || b < c) return 0;
    if (a == c && b == d) return p[k];
    int w = d-c+1;
    return summa(a, min(b,c+w/2-1), 2*k, c, c+w/2-1) +
           summa(max(a,c+w/2), b, 2*k+1, c+w/2, d);
}
```

Nyt välin $[a,b]$ summan saa laskettua kutsumalla funktiota näin:

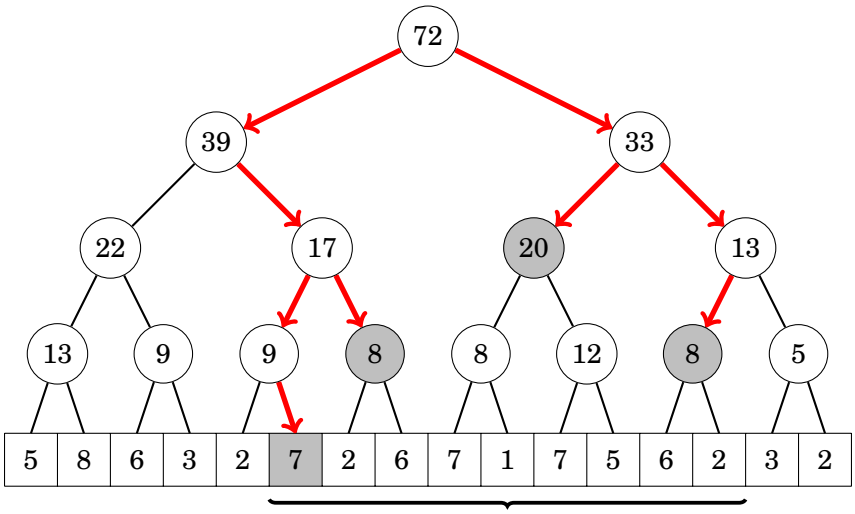
```
int s = summa(a, b, 1, 0, N-1);
```

Parametri k ilmaisee kohdan taulukossa p. Aluksi k:n arvona on 1, koska summan laskeminen alkaa segmenttipuun juuresta.

Väli $[c,d]$ on parametria k vastaava väli, aluksi koko kyselyalue eli $[0,N-1]$. Jos väli $[a,b]$ on välin $[c,d]$ ulkopuolella, välin summa on 0. Jos taas välit $[a,b]$ ja $[c,d]$ ovat samat, summan saa taulukosta p.

Jos väli $[a,b]$ on välin $[c,d]$ sisällä, haku jatkuu rekursiivisesti välin $[c,d]$ vasemmasta ja oikeasta puoliskosta. Kun w on välin $[c,d]$ pituus, vasen puolisko kattaa välin $[c,c+w/2-1]$ ja oikea puolisko kattaa välin $[c+w/2,d]$.

Seuraava kuva näyttää, kuinka haku etenee puussa, kun tehtävänä on laskea puun alle merkityn välin summa. Harmaat solmut ovat kohtia, joissa rekursio päättyy ja välin summan saa taulukosta p.



Myös tässä toteutuksessa kyselyn aikavaativuus on $O(\log n)$.

24.2 Laiska päivitys

Laiska päivitys mahdollistaa segmenttipuun, jossa sekä päivitykset että kyselyt kohdistuvat väleihin. Ideana on tallentaa jokaiseen solmun varsinaisen arvon lisäksi tietoa, miten solmun alipuuta tulee päivittää. Päivitys on kuitenkin laiska eikä se tapahdu, jos alipuuta ei tarvitse käsitellä.

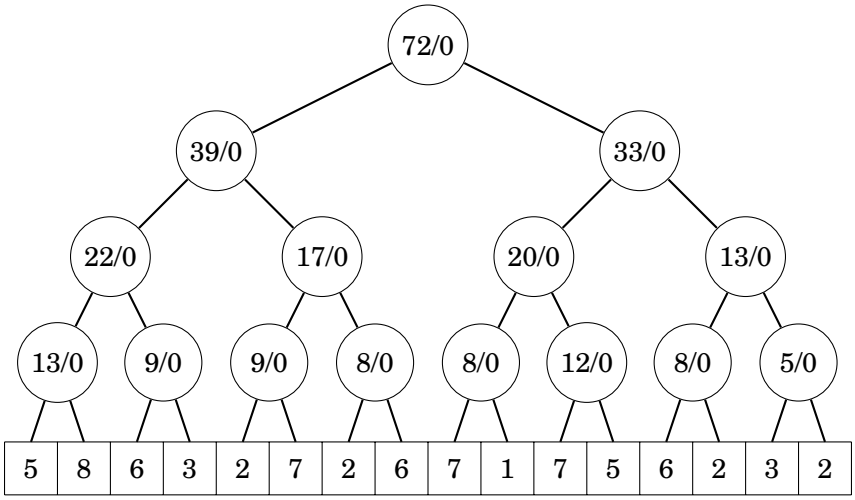
Tarkastellaan esimerkkinä segmenttipuuta, jonka operaatiot ovat:

- kasvata jokaista välin $[a,b]$ arvoa x:llä
- laske välin $[a,b]$ arvojen summa

Molemmat puun operaatiot toteutetaan ylhäältä alaspäin. Jokaisessa solmussa on kaksi arvoa: solmua vastaavan alipuun summa, kuten tavallisessa segment-

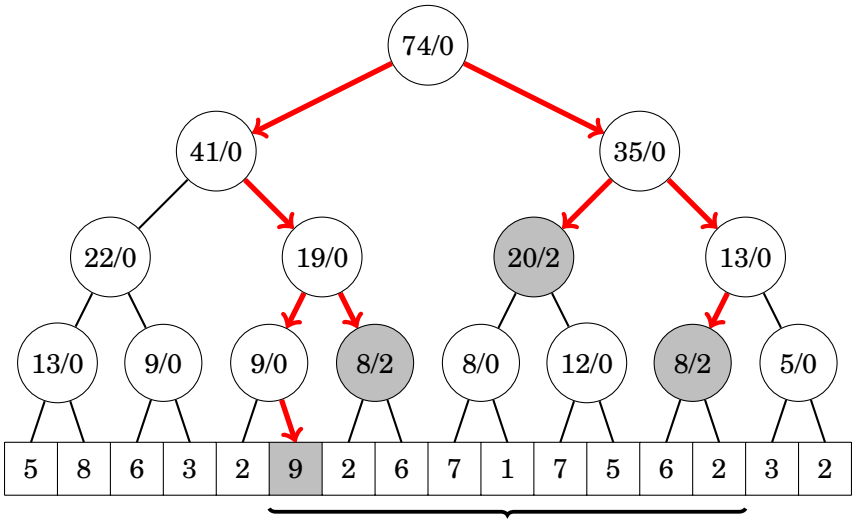
tipuussa, sekä laiskaan päivitykseen liittyvä arvo z. Arvo z solmussa tarkoittaa, että kaikkiin solmun alipuun solmuihin tulee lisätä arvo z.

Perustilanteessa jokaisessa solmussa $z = 0$, mikä tarkoittaa, että mitään päivityksiä ei ole kesken. Näin on seuraavassa kuvassa:



Nyt jokaisen solmun sisältönä on lukupari s/z, missä s on alipuun summa ja z on alipuuhun lisättävä arvo.

Kasvatetaan sitten välin arvoja 2:llä:

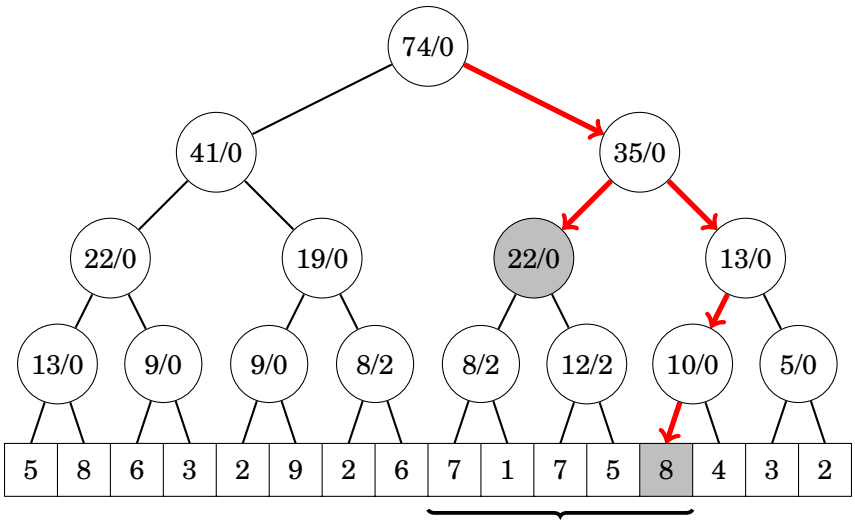


Puussa tapahtui nyt kahdenlaisia muutoksia. Kussakin rekursion välisolmussa alipuun summa kasvoi 2:lla. Rekursion päätesolmuihin taas tuli laiska päivitys: arvo z kasvoi 2:lla mutta alipuun summat eivät muuttuneet vielä.

Laiska päivitys jatkuu puussa alaspäin vasta sitten, kun on todellinen syy mennä alemmas alipuuta toisen päivityksen tai summakyselyn takia. Useita laiskoja päivityksiä voi olla päällekkäin niin, että z kertoo niiden summan.

Laiskan päivityksen ansiosta päivitysoperaation aikavaativuus on vain $O(\log n)$, koska operaatio käsittelee puusta samaa osaa kuin välikysely.

Katsotaan lopuksi, miten laiska päivitys etenee kyselyoperaatiossa:



Laiskan päivityksen z-arvo siirtyi kerrosta alemmas kaikista solmuista, joiden alipuun summaa tarvittiin kyselyissä. Laiska päivitys eteni siis juuri sen verran eteenpäin, että haluttu summa saatiin laskettua.

24.3 Harva lukuväli

Tavallisen segmenttipuun rajoituksena on, että lukuvälinä on yhtenäinen alue $0,1,2,\ldots,N-1$. Tämä ratkaisu ei toimi, jos N on niin iso, että muisti ei riitä segmenttipuun tallentamiseen. Segmenttipuun yleistämiseen harvalle lukuvälille on kuitenkin monia mahdollisuuksia.

24.3.1 Koordinaattien pakkaus

Yksinkertaisin tapa toteuttaa harvan lukuvälin segmenttipuu on käyttää koordinaattien pakkausta (luku 9.3). Tällöin lukuväliksi tulee $0,1,2,\ldots$ ja tavallinen segmenttipuu riittää. Koordinaattien pakkausta voi kuitenkin käyttää vain, jos kaikki lukuvälin arvot tunnetaan algoritmin alussa.

24.3.2 `map`-rakenne

Tavallinen tapa toteuttaa taulukko, joka on vapaasti indeksoitavissa, on käyttää `map`-rakennetta. Yllättävää kyllä, idea on mahdollista yleistää myös harvan lukuvälin segmenttipuuhun.

Tällaisen harvan segmenttipuun voi toteuttaa kuten tavallisen segmenttipuun, mutta taulukko `p` on korvattu `map`-rakenteella. Tällöin muistissa ovat vain ne segmenttipuun solmut, joita on tarvittu segmenttipuun käsittelyn aikana, ja muiden solmujen tulkitaan olevan oletusarvoisia.

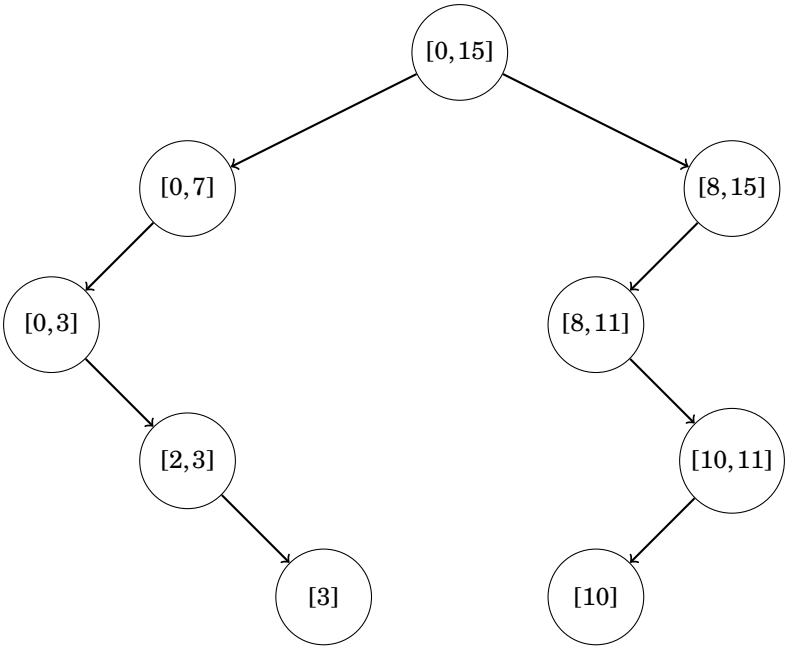
Harvan lukuvälin segmenttipuu on mukavaa toteuttaa `map`-rakenteena, mutta huonona puolena on, että ratkaisu on usein käytännössä melko hidas.

24.3.3 Dynaaminen puu

Tehokas tapa toteuttaa yleinen harvan lukuvälin segmenttipuu on luoda puun solmuja dynaamisesti ja toteuttaa puun käsittely ylhäältä alaspäin.

Aluksi puun ainoa solmu on väliä $[0, N-1]$ vastaava puu. Kun puuhun tapahtuu kyselyitä ja päivityksiä, siihen lisätään tarvittavia solmuja. Jokainen kysely ja päivitys lisää enintään $O(\log n)$ uutta solmua puuhun.

Esimerkiksi jos $N = 16$ ja lukuvälin arvoja 3 ja 10 on muutettu, puu näyttää muistissa seuraavanlaiselta:



Solmut on kätevää tallentaa tietueina tähän tapaan:

```
struct solmu {
    int x;
    int a, b;
    solmu *l, *r;
    solmu(int x, int a, int b) : x(x), a(a), b(b) {}
};
```

Tässä x on solmussa oleva arvo, $[a,b]$ on solmua vastaava väli ja l ja r osoittavat solmun vasempaan ja oikeaan alipuuhun.

Tämän jälkeen solmuja voi käsitellä seuraavasti:

```
// uuden solmun luonti
solmu *s = new solmu(0, 0, 15);
// kentän muuttaminen
s->x = 5;
```

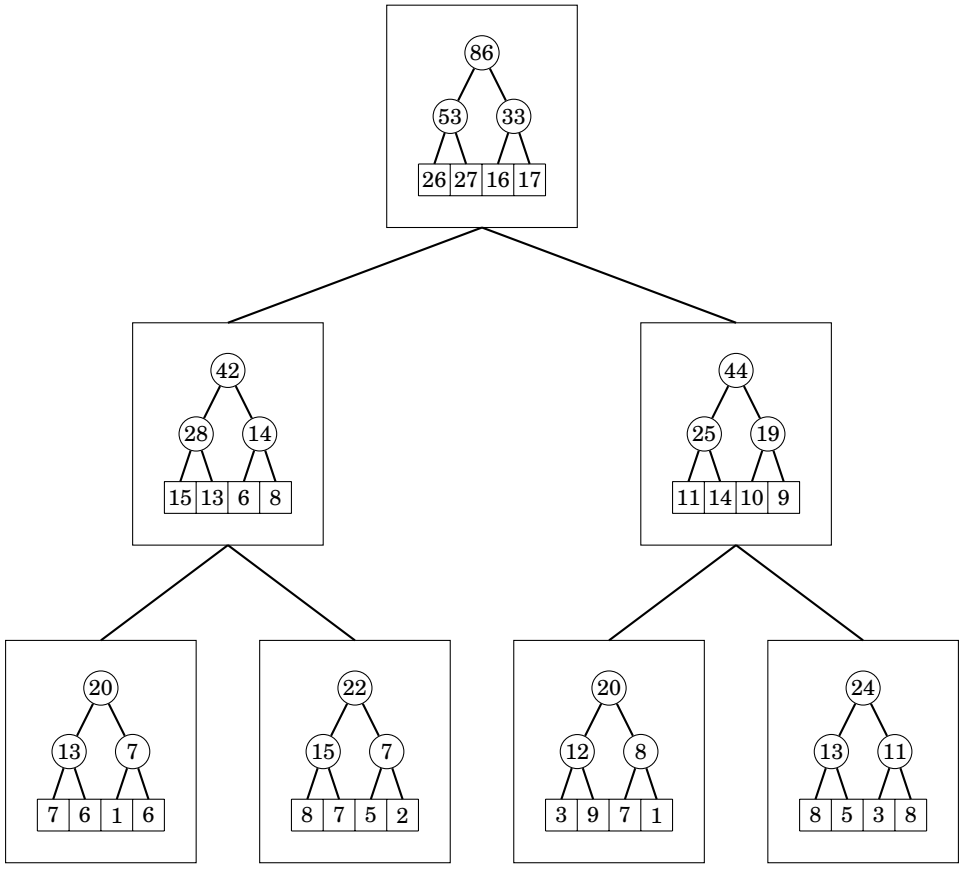
24.4 Kaksiulotteisuus

Kaksiulotteinen segmenttipuu mahdollistaa kyselyt kaksiulotteisen taulukon alueella. Kaksiulotteisen segmenttipuun voi toteuttaa rakentamalla segmenttipuun, jonka jokaisessa solmussa on segmenttipuu. Nyt suuri segmenttipuu vastaa taulukon rivejä ja pienet segmenttipuut vastaavat sarakkeita.

Esimerkiksi taulukosta

7	6	1	6
8	7	5	2
3	9	7	1
8	5	3	8

syntyy seuraavanlainen segmenttipuu:

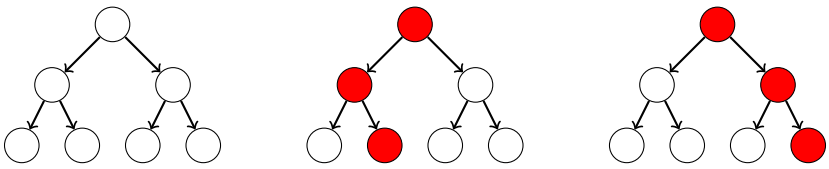


Tässä segmenttipuussa operaatioiden aikavaativuus on $O(\log^2 n)$, koska suuressa segmenttipuussa käsitellään $O(\log n)$ solmua, ja jokaisessa solmussa pienen segmenttipuun käsittely vie aikaa $O(\log n)$.

24.5 Muutoshistoria

Segmenttipuuta on myös mahdollista laajentaa niin, että muistissa on kaikki puun vaiheet historian aikana. Tämä perustuu siihen, että jokainen muutos vaikuttaa vain pieneen osaan puun solmuista.

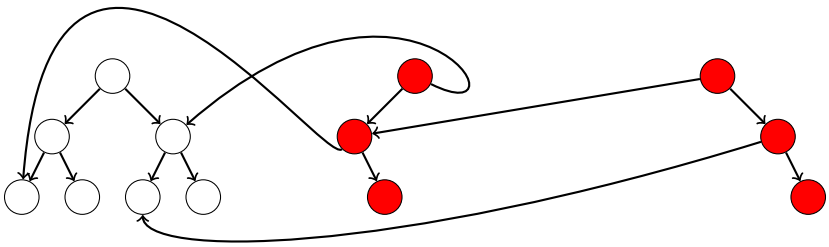
Tarkastellaan esimerkiksi seuraavaa muutossarjaa:



vaihe 1 vaihe 2 vaihe 3

Punaiset solmut muuttuvat päivityksessä, ja muut solmut säilyvät ennallaan.

Muistia säästävä tapa tallentaa muutoshistoria on käyttää mahdollisimman paljon hyväksi puun vanhoja osia joka muutoksen jälkeen. Tässä tapauksessa muutoshistorian voisi tallentaa seuraavasti:



vaihe 1 vaihe 2 vaihe 3

Jokainen muutos tuo vain $O(\log n)$ uutta solmua puuhun, joten koko muutoshistorian pitäminen muistissa on mahdollista.