Tasattu eli amortisoitu ajoaika

Vektori on joustavarajainen taulukko eli sen kokoa kasvatetaan tarvittaessa.

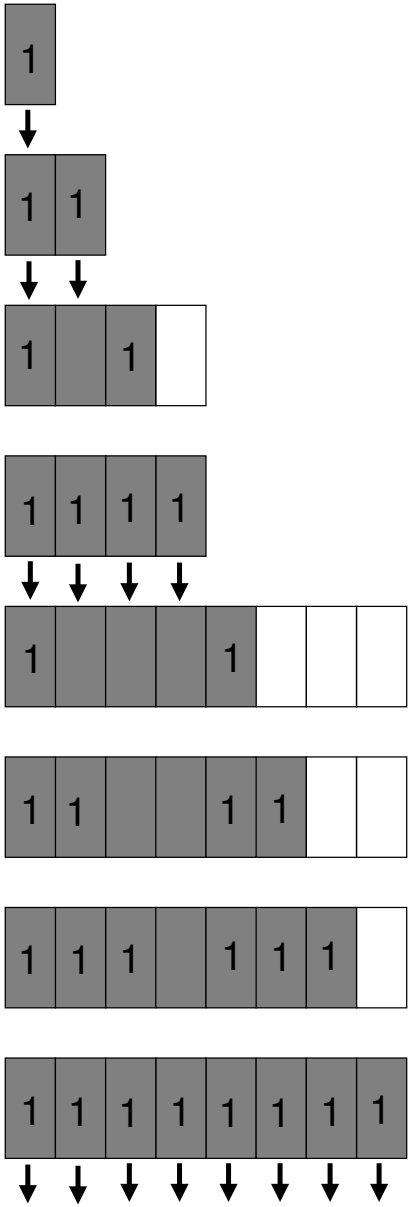
- kun uusi alkio ei enää mahdu taulukkoon, varataan uusi suurempi ja siirretään kaikki alkiot sinne
- taulukko ei koskaan kutistu

  $\Rightarrow$ muistin varaus ei vähene muuten kuin kopioimalla vektoriin kokonaan uusi sisältö

$\Rightarrow$ Alkion lisäämisen vectorin loppuun sanottiin olevan *tasatusti (amortisoidusti)* vakioaikaista.

- Amortisoidusti lähestyttäessä suoritusaikaa tarkastellaan kokonaisuutena, tutkitaan operaatiosarjojen suoritusaikaa yksittäisten operaatioiden sijaan
  - jokaista kallista muistinvarausta vaativaa lisäysoperaatiota edeltää kalliin operaation hintaan suoraan verrannollinen määrä halpoja lisäysoperaatioita
  - kalliin operaation kustannus voidaan jakaa tasan halvoille operaatioille
  - tällöin halvat operaatiot ovat edelleen vakioaikaisia, tosin vakiokertoimen verran hitaampia kuin oikeasti
  - kallis operaatio voidaan maksaa säästöillä

$\Rightarrow$ kaikki lisäysoperaatiot vektorin loppuun ovat tasatusti vakioaikaisia



Tämä voidaan todentaa vaikkapa kirjanpitomenetelmällä:

- laskutetaan jokaisesta lisäyksestä kolme rahaa
- yksi raha käytetään lisäyksen todellisiin kustannuksiin
- yksi raha laitetaan säästöön lisätyn alkion i kohdalle
- yksi raha laitetaan säästöön alkion $i - \frac{1}{2} \cdot vector.capacity()$ kohdalle
- kun tulee tarve laajentaa taulukkoa, jokaisella alkiolla on yksi raha säästössä, ja kallis kopiointi voidaan maksaa niillä

6.2 Suunnitteluperiaate: Satunnaistaminen

Satunnaista on eräs algoritmien suunnitteluperiaatteista.

- Sen avulla voidaan usein estää huonoimpien tapausten patologinen ilmeneminen.
- Parhaan ja huonoimman tapauksen suoritusajat eivät useinkaan muutu, mutta niiden esiintymistodennäköisuus käytännössä laskee.
- Huonot syötteet ovat täsmälleen yhtä todennäköisiä kuin mitkä tahansa muut syötteet riippumatta alkuperäisestä syötteiden jakaumasta.
- Satunnaistaminen voidaan suorittaa joko ennen algoritmin suoritusta satunnaistamalla sen saama syöteaineisto tai upottamalla satunnaistaminen algoritmin sisälle.
  - jälkimmäisellä tavalla päästään usein parempaan tulokseen
  - usein se on myös helpompaa kuin syötteen esikäsittely

- Satunnaistaminen on hyvä ratkaisu yleensä silloin, kun
  - algoritmi voi jatkaa suoritustaan monella tavalla
  - on vaikea arvata etukäteen, mikä tapa on hyvä
  - suuri osa tavoista on hyviä
  - muutama huono arvaus hyvien joukossa ei haittaa paljoa
- Esimerkiksi QUICKSORT voi valita jakoarvoksi minkä tahansa taulukon alkion
  - hyviä valintoja ovat kaikki muut, paitsi lähes pienimmät ja lähes suurimmat taulukossa olevat arvot
  - on vaikea arvata valintaa tehdessä, onko ko. arvo lähes pienin / suurin
  - muutama huono arvaus silloin tällöin ei turmele QUICKSORTin suorituskykyä

  $\Rightarrow$ satunnaistaminen sopii QUICKSORTille

Satunnaistamisen avulla voidaan tuottaa algoritmi RANDOMIZED-QUICKSORT, joka käyttää satunnaistettua PARTITIONIA.

- Ei valita jakoarvoksi aina $A[\, right\,]$:tä, vaan valitaan jakoarvo satunnaisesti koko osataulukosta.
- Jotta PARTITION ei menisi rikki, sijoitetaan jakoarvo silti kohtaan $right$ taulukkoa

  $\Rightarrow$ Nyt jako on todennäköisesti melko tasainen riippumatta siitä, mikä syöte saatiin ja mitä taulukolle on jo ehditty tehdä.

```
RANDOMIZED-PARTITION( A, left, right )
1   p := RANDOM(left, right)          (valitaan satunnainen alkio pivotiksi)
2   exchange A[ right ] ↔ A[ p ]      (asetetaan se taulukon viimeiseksi)
3   return PARTITION( A, left, right )  (kutsutaan tavallista partitiointia)
```

```
RANDOMIZED-QUICKSORT( A, left, right )
1   if left < right then
2       p := RANDOMIZED-PARTITION( A, left, right )
3       RANDOMIZED-QUICKSORT( A, left, p − 1 )
4       RANDOMIZED-QUICKSORT( A, p + 1, right )
```

RANDOMIZED-QUICKSORTIN ajoaika on keskimäärin $\Theta(n \lg n)$
samoin kuin tavallisenkin QUICKSORTIN.

- RANDOMIZED-QUICKSORTILLE kuitenkin varmasti pätee keskimääräisen ajankäytön analyysin yhteydessä tehtävä oletus, jonka mukaan pivot-alkio on osataulukon pienin, toiseksi pienin jne. aina samalla todennäköisyydellä.
- Tavalliselle QUICKSORTILLE tämä pätee ainoastaan, jos aineisto on tasaisesti jakautunutta.

$\Rightarrow$ RANDOMIZED-QUICKSORT on yleisessä tapauksessa tavallista QUICKSORTIA parempi.

QUICKSORTIA voidaan tehostaa myös muilla keinoilla:

- Voidaan järjestää pienet osataulukot pienille taulukoille tehokkaalla algoritmilla (esim. INSERTIONSORT) avulla.
  - voidaan myös jättää ne vain järjestämättä ja järjestää taulukko lopuksi INSERTIONSORTIN avulla
- Jakoarvo voidaan valita esimerkiksi kolmen satunnaisesti valitun alkion mediaanina.
- On jopa mahdollista käyttää aina mediaanialkiota jakoalkiona.

Mediaani on mahdollista etsiä nopeasti niin sanotun laiskan QUICKSORTIN avulla.

- Jaetaan taulukko "pienten alkioiden" alaosaan ja "suurten alkioiden" yläosaan kuten QUICKSORTissa.
- Lasketaan, kumpaan osaan i:s alkio kuuluu, ja jatketaan rekursiivisesti sieltä.
- Toiselle osalle ei tarvitse tehdä enää mitään.

```
RANDOMIZED-SELECT( A, left, right, goal )
1   if left = right then                          (jos osataulukko on yhden kokoinen...)
2       return A[ left ]                          (... palautetaan ainoa alkio)
3   p := RANDOMIZED-PARTITION( A, left, right )   (jaetaan taulukko pieniin ja isoihin)
4   k := p − left + 1                             (lasketaan monesko jakoalkio on)
5   if i = k then                                 (jos jakoalkio on taulukon i:s alkio...)
6       return A[ p ]                             (...palautetaan se)
7   else if i < k then                            (jatketaan etsintää pienten puolelta)
8       return RANDOMIZED-SELECT( A, left, p − 1, goal )
9   else                                          (jatketaan etsintää suurten puolelta)
10      return RANDOMIZED-SELECT( A, p + 1, right, goal − k )
```

RANDOMIZED-SELECTIN suoritusajan alaraja:

- Jälleen kaikki muu on vakioaikaista paitsi RANDOMIZED-PARTITION ja rekursiivinen kutsu.
- Parhaassa tapauksessa RANDOMIZED-PARTITIONIN valitsema jakoalkio on taulukon i:s alkio, ja ohjelman suoritus loppuu.
- RANDOMIZED-PARTITION ajetaan kerran koko taulukolle.

$\Rightarrow$ Algoritmin suoritusaika on $\Omega(n)$.

RANDOMIZED-SELECTIN suoritusajan yläraja:

- RANDOMIZED-PARTITION sattuu aina valitsemaan pienimmän tai suurimman alkion, ja i:s alkio jää suuremmalle puoliskolle
- työmäärä pienenee vain yhdellä askeleella joka rekursiotasolla.

$\Rightarrow$ Algoritmin suoritusaika on $O(n^2)$.

Keskimääräisen tapauksen ajoaika on kuitenkin $O(n)$.

Algoritmi löytyy esimerkiksi STL:stä nimellä `nth_element`.

Algoritmi on mahdollista muuttaa myös toimimaan aina lineaarisessa ajassa.