Toteutusdokumentti

Joel Järvinen

February 19, 2014

1 Algoritmien tavoiteanalyysi

Tulen tässä arvioimaan olenko saavuttanut määrittelydokumentissa asetetut tavoitteet, ja olisiko mahdollisesti algoritmeissa ollut parannettavaa.

1.1 Kekojärjestäminen - Heap Sort

Määrittelydokumentissa annoin kekojärjestämiselle aika- ja tilavaativuudeksi
$\mathcal{O}$(log *n*)toteutuksella tämän olen saavuttanut sillä:

Tarkastellaan ensin **heapify** operaatiota, jonka suoritusaika riippuu puun korkeudesta, sillä keon rakennehan on lähes täydellinen binääripuu on keon korkeus puuna kuvattuna $\mathcal{O}$(log *n*) *n* alkiota sisältävällä listalla. Täten **heapify** operaatio pahimmassa tapauksessa saa aikavaativuudekseen $\mathcal{O}$(log *n*).

heapify(A, index)
1: **if** $right \leq heapSize$ **then**
2: **if** $A[left] \geq A[right]$ **then**
3: $largest = left$
4: **else**
5: $largest = right$
6: **end if**
7: **if** $A[index] \leq A[largest]$ **then**
8: $swap(A, index, largest)$
9: $heapify(A, largest)$
10: **end if**
11: **end if**

Toiseksi tarkkailun kohteeksi nousee **buildHeap** operaatio, jonka rungossa kutsutaan **heapify** *n*/2 kertaa listalle jossa on *n* alkiota, täten **buildHeap** käyttää aikaa korkeintaan $\mathcal{O}$(*n* log *n*). Tällöin listasta ollaan saatu muodostettua keko.

Ja lopuksi algoritmissa kutsutaan **sort** operaation aikana **heapify** *n* - 1 kertaa, tällöin kokonais aikavaativuudeksi saadaan $\mathcal{O}$(*n* log *n*). Tilavaativuudeksi kekojärjestämiselle saadaan rekursio kutsujen määrä $\mathcal{O}$(*n* log *n*).

sort(A)
1: **for** $i = heapSize$ down to 2 **do**
2: $swap(1, i)$
3: $heapSize - 1$
4: $heapify(A, 1)$
5: **end for**

1.2 Kuplajärjestäminen - BubbleSort

Määrittely dokumentissa annan kupla järjestämiselle aikavaativuuden $\mathcal{O}(n^2)$, ja toteutetussa koodissa aikavaativuus on $\mathcal{O}(n^2)$ sillä:

- For-loop sisältää valinnan ja vakioajan vieviä sijoitusoperaatiota. For-loopin runko on siis vakioaikainen $\mathcal{O}(1)$
- For-loopin runko sen sijaan suoritetaan pahimmillaan *n* kertaa, sillä for-loop suoritetaan niin kauan että ollaan listan läpi menty kertaalleen ilman että listan sisällä on tehty swappauksia.
- Eli runko suoritetaan pahimmillaan 1 + 2 + 3 + ... + *n* täten aikavaativuudeksi nousee $\mathcal{O}(n^2)$.

Tilavaativuus kuplajärjestämisellä sen sijaan on $\mathcal{O}(1)$, sillä järjestäminen ei vaadi aputilaa ja käytetään ainoastaan yhtä apumuuttujaa.

1.3 Lomitusjärjestäminen - MergeSort

Määrittely dokumentissa annan lomitusjärjestämiselle aikavaativuuden $\mathcal{O}(n \log n)$, ja toteutuksessa tähän on päästy sillä,

Merge-operaation vaativuus, merge operaatiossa käytetään hyvänä luokan alussa luotua aputaulukkoa. **Mergen** aikana lomitetaan aputaulukon *vasen - oikea* välillä olevia alkiota.

- **Mergen** alussa sijoitetaan taulukon A *vasen - oikea* välillä olevat alkiot taulukkoon temp

  alustus
  1: **for** $i = left$ up to $right$ **do**
  2: $temp[i] = A[i]$
  3: **end for**

  Tämän operaation aikavaativuus on täten $\mathcal{O}(k)$, missä *k* = lomitettavan taulukonosan pituus.

- Tämän jälkeen sijoitetaan alkiot temp taulukosta takaisin A taulukkoon valittaen aina temp[*left*] ja temp[*middle*] väliltä pienempi arvo. Tässä *middle* kuvastaa oikean lohkon valinnan sijaintia.

  takaisin sijoitus
  1: **while** $left \leq leftEnd$ and $middle \leq right$ **do**
  2: **if** $temp[left] \leq temp[middle]$ **then**
  3: $A[i] = temp[left]$
  4: $i++$ and $left++$
  5: **else**
  6: $A[i] = temp[middle]$
  7: $i++$ and $middle++$
  8: **end if**
  9: **end while**

- Koska on mahdollista että lohkot eivät mene tasan lopussa vielä asetetaan loput alkiot väliltä *left - leftEnd* takaisin taulukkoon A.

  lopullinen sijoitus
  1: **while** $left \leq leftEnd$ **do**
  2: $A[i] = temp[left]$
  3: $i++$ and $left++$
  4: **end while**

  Näiden kahden yllä mainitun while loopin läpi käyntiin kätetään jälleen aikaa $\mathcal{O}(k)$. Täten koko **merge**-operaation aikavaativuudeksi sadaan $\mathcal{O}(k)$.

- Lopullisen lomitusjärjestämisen aikavaativuuden saamiseen täytyy ottaa huomioon rekursio kutsujen määrä. Jokainen sort kutsu puolittaa syötteensä koon, täten kun syöte on aluksi *n*. Niin kun puolitus on tehty $\log_2$ *n*-kertaa on syötteen koko enään 1, eli kun mukaan luetaan alin taso, jolla ei enään rekursio kutsua tehdä on rekursiotasoja $\log_2$ *n* + 1 kappaletta.

  Tästä nähdään että lomitusjärjestämisen kokonaisaikavaativuus on $\mathcal{O}(n \log n)$.

Lomitusjärjestämisen tilavaativuus, kuten yllätodettiin on rekursiopuun koko korkeintaan $\log_2$ *n*, ja aputila joka luodaan järjestämisen luokan alussa on kokonaisuudessaan $\mathcal{O}(n)$. On tilavaativuus $\mathcal{O}(n)$.

1.4 Pikajärjestäminen - Quick Sort

Määrittelydokumentissa annan quick sortille pahimmantapauksen aikavaativuudeksi $\mathcal{O}(n^2)$ ja keskimääräinen aikavaativuudeksi $\mathcal{O}(n \log n)$ja tilavaativuudeksi $\mathcal{O}(\log n)$. Omaan toteutukseen lisäsin lineaarisessa ajassa toimivan tarkistuksen joka tarkistaa ettei taulukko ole järjestyksessä. Näin se pikajärjestämisen aivan huonoin aikavaativuus saadaan eliminoitua joka jakaisi taulukon aina kahtia jättäen toiseen osaan vain 1 alkion, suuremmilla syötteillä tämä aiheuttaisi myös sen että muisti loppuu kesken suuren rekursiopuun takia.

Tiran materiaaleissa on pikajärjestämisen aikavaativuudesta hyvin laajat analyysit tässä otos todistuksessa jossa todistetaan että pikajärjestäminen toimii $\mathcal{O}(n \log n)$ ajassa jopa silloin kun partition jakaa alkioita melko huonosti. Jossa oletetaan että k:n alkion taulukko jakautuisi pahimmillaan siten että pienemmässä osassa on vain $k/10$ alkiota ja suuremmassa $9k/10$.

$T_u(n) \leq cn(\log_{10/9} n + 1) = cn\ (\frac{log_2 n}{log_2 10/9} + 1) \leq cn\ (7 \times \log_2 n + 1) =$
$7 \times cn\ (\log_2 n + \frac{1}{7}) = \mathcal{O}(n \log n)$

partition(A, left, right)

1: $pivot = A[left]$
2: $left = left - 1$
3: $right = right - 1$
4: **while** $left < right$ **do**
5: $left++$ and $right--$
6: **while** $A[right] > pivot$ **do**
7: $right--$
8: **end while**
9: **while** $A[left] < pivot$ **do**
10: $left++$
11: **end while**
12: **if** $left < right$ **then**
13: $swap(A, left, right)$
14: **end if**
15: **end while**

Tilavaativuudeksi pikajärjestämisessä omassa toteutuksessani jäi $\mathcal{O}(n)$, rekursiopuun koon takia.

Tilavaativuudessa olisikin voinut päästä parempaan jos olisi osan partitionista tehnyt iteraatiolla pelkän rekursion sijaan.

1.5 Laskemisjärjestäminen - Counting Sort

Laskemisjärjestämisellä päästään lineaariseen aikavaativuuteen, mutta ehtona on että tiedetään kuinka suuria lukuja listalla on. Pseudokoodina, missä *k* = suurin mahdollinen luku listalla

aputaulukon alustus

for $i = 0$ up to k **do**
    $C[i] = 0$
end for

lasketaan kuinka usein kukin numero esiintyy listalla

for $i = 0$ up to n **do**
    $x = A[i]$
    $C[x] = C[x] + 1$
end for

Tarkistetaan kuinka monta *korkeintaan i*:n suuruista lukua on taulukossa A

for $i = 2$ up to k **do**
    $C[i] = C[i] + C[i - 1]$
end for

Järjestetään taulukon A alkiot taulukkoon B tarkistamalla kuinka monta lukua *i* on taulukossa A jäljellä

for $i = n$ down to 1 **do**
    $x = A[i]$
    $B[C[x]] = x$
    $C[x] = C[x] - 1$
end for

Laskemisjärjestämisen aikana käydään taulukot A ja C kahdesti läpi sekä taulukko B kertaalleen, täten aikavaativuudeksi saadaan $\mathcal{O}(n + k)$.

Tilavaativuus saadan katsomalla aputaulukkojen B ja C koko jotka ovat B = *n* ja C = *k*, missä k = suurimman mahdollisen luvun määrä, täten tilavaativuus on $\mathcal{O}(n + k)$.

2 Tietorakenteen analyysi

Osa keon analyyseistä on tehty jo algoritmien kohdassa kekojärjestäminen, sillä keko tietorakenteessa käytetään samanlaista **heapify** toimintoa. Joten tässä arvioin muiden keon operaatioiden toteutusta.

- **peek**-operaatio, jolla voidaan tarkastella keon päällimmäistä / ensimmäistä alkiota. Maksimi keossahan tämä on suurin, ja minimi keossa pienin. **peek** on hyvinkin triviaali ja käytännössä vain palauttaa ensimmäisen alkion:

  peek

  if $currentSize == 0$ **then**
      return $null$
  end if
  return $heap[1]$

  Tästä näemmä että operaation aikavaativuus on selvästi vakio

- **remove**/**pop**-operaatio joka palauttaa ensimmäisen alkion keosta, ja poistaa sen sieltä. Metodi kutsuu **heapify** korjaamaan keko ehto joten metodilla on samat vaativuusluokat kuin **heapifyllä** $\mathcal{O}$(log *n*)

  pop

  if $currentSize == 0$ **then**
      return $null$
  end if

  $E\ item$ = $(E)\ heap[1]$

  $heap[1]$ = $heap[--currentSize]$
  $heap[currentSize]$ = $null$
  $heapify(1)$ and return $item$

Alkion lisääminen kekoon tapahtuu kasvattamalla keon kokoa yhdellä, eli teemme paikan uudelle alkiolle. Tämän jälkeen kuljetaan keon uudesta alkiosta ylöspäin ja samalla siirretään vanhoja alkiota alaspäin kunnes löydämme paikan uudelle alkiolle joka ei riko keko-ehtoja.
Pahimmassa tapauksessa joudumme kuljettamaan uuden alkion aina juureen asti ja tällöin vanhoja alkioita on kuljetettu puun korkeuden verran alaspäin. Ja koska keko oli lähes täydellinen binääripuu on sen korkeun $\mathcal{O}$(log *n*) täten siis operaation aikavaativuus on sama. Tässä pseudokoodissa on kyseessä maksimikeon insert

insert(E node)

1: i = $currentSize++$
2: **while** $i > 0$ **do**
3: $parentIndex$ = $getParentIndex(i-1)$
4: $E\ parent$ = $(E)\ heap[parentIndex]$
5:
6: **if** $node \leq parent$ **then**
7: $break$
8: **end if**
9:
10: $heap[i] = parent$
11: $i = parentIndex$
12: **end while**