Eksamensdisposition - Amortiseret analyse

Søren Mulvad, rbn601

3. april 2019

- **Motivation**
- **Aggregeret analyse**
  - Stack-eksempel
- **Accounting method**
  - Stack-eksempel
- **Potentiale metode**
  - Generelt
  - Stack-eksempel
  - Dynamisk tabel kun med `Insert`

1 Amortiseret analyse

- **Motivation**
  - Nogle gange er vi for en datastruktur ikke interesseret i dens worst case køretid hver eneste gang en operation udføres, men i stedet interesseret i hvad average case er når vi udfører en række af instruktioner.
  - Jeg vil her illustrere koncepterne på nogle lidt tænkte eksempler, men amortiseret analyse er også relevant i Fibonacci Heaps, og dermed i Dijkstras algoritme og Prims algoritme (MST), samt i disjoint-set forests hvilket benyttes i Kruskals algoritme (MST).
- **Aggregeret analyse**
  - Antag vi har en stack der har operationerne `Push(S, x)`, `Pop(S)`, `Stack-Empty(S)` som alle tager $\Theta(1)$-tid.
    Derudover har vi `MultiPop(S, k)`, som kalder `Pop` og `Stack-Empty` et antal gange der svarer til minimum af antal elementer og en parameter k vi giver den. Denne vil køre i $\Theta(\min(s, k))$-tid.
  - Hvis vi udfører n operationer på en stack der til at starte med er tom ser vi, at vi worst-case får en køretid på $O(n^2)$.
  - Vi beregner et upper bound $T(n)$ for alle n operationer og får derved den amortiserede cost pr. operation til $T(n)/n$.
  - Nu udnytter vi, at der højest kan være n `Push`-operationer og der kan ikke være flere `Pop`-operationer (inklusiv dem i `MultiPop`) end `Push`-operationer.
  - Så får vi at worst-case for alle n operationer er $O(n)$, og herved at den amortiserede cost pr. operaton er $O(n)/n = O(1)$.
- **Accounting method**
  - Den rigtige cost for den i'te operation er c_i
    Den amortiserede cost for den i'te operation er $\hat{c}_i$.
  - Hvis $\hat{c}_i > c_i$, så overcharger vi operationen og har på den måde noget "kredit" at bruge af. Hvis det omvendte gør sig gældende, så undercharger vi og bruger herved noget af den kredit vi har bygget op.
  - Der skal ALTID gælde:
    $$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$$
    Hvis vi nu kan få et upper bound på $\sum_{i=1}^{n} \hat{c}_i$ får vi også et upper bound på den faktiske omkostning.
  - Nu henholdsvis får/vælger vi følgende værdier:

Operation	**Faktisk cost**	**Amortiseret cost**
`Push`	1	2
`Pop`	1	0
`MultiPop`	$\min(s, k)$	0

    Vi kan se på det på den måde, at ved `Push` betaler vi både for selve operationen med 1 samt lægger 1 kredit på elementet.
  - `Pop` bliver undercharged med 1, men denne cost betaler vi med den kredit vi har lagt på elementet. Vi ser at vi aldrig får en kredit der er negativ.

- Vi ser, at for enhver sekvens af n operationer har vi

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \leq 2n$$

og herved bliver den gennemsnitlige cost ≤ 2, hvorved vi får en gennemsnitlig køretid på $O(1)$.

- **Potentialemetoden - Generelt**
  - Minder lidt om accounting method bortset fra at kreditten ikke er gemt på enkelte elementer men i stedet i "banken". Mængden af kredit i banken er udtrykt ved en potentialefunktion Φ.
  - Hvis vi udfører n operationer på en datastruktur hvor D_i er strukturen efter den i'te operation for $i = 1, ..., n$, så skriver vi $\Phi(D_i)$ som symbol for kreditten der er gemt med den nuværende struktur.
  - Vi definerer den amortiserede cost $\hat{c}_i$ som den faktiske cost c_i plus forskellen mellem hvad der er i potentialet nu og hvad der var lige før.

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Hvis $\Phi(D_i) - \Phi(D_{i-1}) > 0$ kan man sige at vi putter kredit i banken, og hvis det er mindre tager vi kredit fra banken.

  - Samme egenskab som for accounting method skal gælde:

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i \tag{1}$$

  - Vi får summen af den amortiserede cost til at blive:

$$\begin{aligned}
\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} \Big(c_i + \Phi(D_i) - \Phi(D_{i-1}) \Big) \\
&= c_1 + \Phi(D_1) - \Phi(D_0) \\
&+ c_2 + \Phi(D_2) - \Phi(D_1) \\
&+ \vdots \\
&+ c_n + \Phi(D_n) - \Phi(D_{n-1}) \\
&= \left(\sum_{i=1}^{n} c_i \right) + \Phi(D_n) - \Phi(D_0)
\end{aligned}$$

Da vi ser at alle potentiale-led pånær det første og sidste går ud med hinanden i summen. Hvis vi sørger for at $\Phi(D_n) \geq \Phi(D_0)$ ser vi, at vi vil opfylde Eq. (1).

- **Potentialemetoden - Stack-eksempel**
  - Vi vælger en potentialefunktion $\Phi(D_i) =$ antallet af elementer på stacken.
    Vi ser tydeligt at Eq. (1) er overholdt, da $D_0 = 0$ og $\Phi(D_i) \geq 0$ for alle i.
  - Nu skal vi upper bound'e dette.
  - `Push`:
    - $\Phi(D_i) - \Phi(D_{i-1}) = 1$ (der tilføjes et element til stacken)
    - Derved fås $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$
  - `Pop`:
    - $\Phi(D_i) - \Phi(D_{i-1}) = -1$ (der fjernes et element fra stacken)
    - Derved fås $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$

- `MultiPop`:
  - Lad os sige vi popper $k' > 0$ elementer.
  - $\Phi(D_i) - \Phi(D_{i-1}) = -k'$ (da der fjernes k' elementer fra stacken)
  - Derved fås $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$
- For alle operationer $i = 1, ..., n$ gælder, at $\hat{c}_i \leq 2$. Herved får vi følgende ulighed:

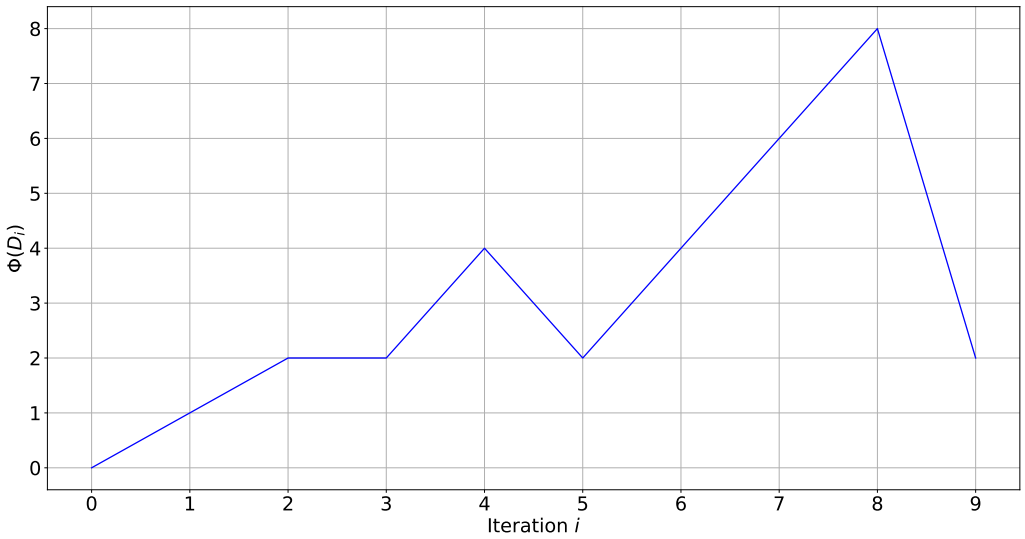
$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \leq 2n$$

  Og derved er den gennemsnitlige tid brugt pr. operation $2n/n = O(1)$.

- **Potentialemetoden - Dynamisk tabel kun med `Insert`**
  - Nu indfører vi en abstrakt datastruktur T, som vi kan tænke på som en tabel. Den understøtter `Insert` og `Delete`, bruger et array til at gemme sin data og bruger $O(k)$ tid på at allokere/free'e et array af størrelse k. For nu antager vi, at T kun understøtter `Insert`.
  - Vi ønsker at T dynamisk allokerer et nyt større array til sig selv når det gamle array er for småt til at indeholde alle elementerne.
  - Indfør følgende notation:
    - num_i: Antallet af elementer i T efter den i'te operation
    - size_i: Størrelsen af arrayet for T efter den i'te operation
    - $\alpha_i = \text{num}_i/\text{size}_i$, loadfaktoren af T efter den i'te operation (hvis $\text{size}_i = 0$, definer $\alpha_i = 1$).
  - Vi starter med at T er tom. Herefter siger vi, at lige før vi indsætter det i'te element, hvis $\text{num}_{i-1} = \text{size}_{i-1}$ (svarende til loadfaktoren $\alpha_{i-1} = 1$) så ekspanderer vi T til et array der er dobbelt så stort, 2size_{i-1} og kopierer de gamle elementer over.
  - Vi ser at tabelekspandering tager $O(\text{num}_i)$ worst-case tid. Så hvis den i'te operation kræver en ekspandering kan vi sætte $c_i = \text{num}_i$.
  - Hvis ingen ekspandering er krævet kan vi sætte $c_i = 1$.

  - Definer potentialefunktionen

$$\Phi(D_i) = 2\text{num}_i - \text{size}_i$$



Figur 1: Potentialet $\Phi(D_i)$ efter hver iteration i

  - Først vis den er valid ved at vise $\Phi(D_0) = 0$ og $\Phi(D_i) \geq 0$ for alle i.

- Tydeligvis er $\Phi(D_0) = 0$ da T er tom til at starte med. Og da vi har, at T altid som minimum er halvt fyld, så har vi også $\Phi(D_i) \geq 0$ for alle i. Derfor gælder Eq. (1).
- Da Eq. (1) gælder kan vi få et upper bound for $\sum_{i=1}^{n} c_i$ ved at upper bound'e $\sum_{i=1}^{n} \hat{c}_i$.
- Hvis der IKKE sker tabelekspandering får vi følgende amortiserede cost:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (2\text{num}_i - \text{size}_i) - (2\text{num}_{i-1} - \text{size}_{i-1}) \\ &= 1 + (2\text{num}_i - \text{size}_i) - (2(\text{num}_i - 1) - \text{size}_i) \\ &= 3\end{aligned}$$

  Vi får dette da size ikke ændrer sig, og det forrige antal elementer svarer til det nuværende antal elementer minus en, da vi jo netop har tilføjet en. 3-tallet fås ved at se, at ting går ud med hinanden.
- Hvis der SKER tabelekspandering OG $i = 1$ får vi den amortiserede cost 2.
- Hvis der SKER tabelekspandering OG $i > 1$ får vi:

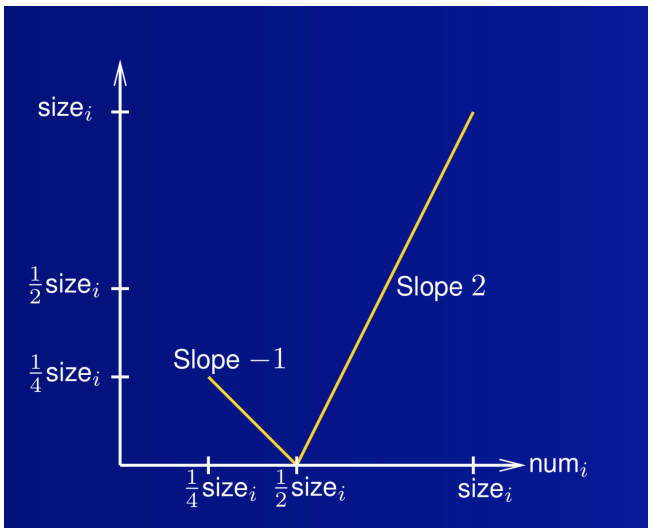
$$\begin{aligned}\hat{c}_i &= c_i\Phi(D_i) - \Phi(D_{i-1}) \\ &= \text{num}_i + (2\text{num}_i - \text{size}_i) - (2\text{num}_{i-1} - \text{size}_{i-1}) \\ &= \text{num}_i + \Big(2\text{num}_i - 2(\text{num}_i - 1)\Big) - \Big(2(\text{num}_i - 1) - (\text{num}_i - 1)\Big) \\ &= 3\end{aligned}$$

  Hvordan disse numre fås kan ræsonneres ud fra grafen, bare husk at alle variabler skal være num_i *til sidst.*

- **Potentialemetoden - High-level bevis når vi har `Insert`/`Delete`**
  - Forklaring af `Delete`: Vi ønsker at kunne fjerne elementer fra tabellen igen, og såfremt vi når ned under en hvis loadfaktor α_i at bruge et mindre array til at holde dem. Naiv implementation vil være at gå til mindre array når $\alpha_i < 1/2$, men så kunne vi potentielt få et problem hvis vi indsætter og sletter mange gange meget tæt på en 2-tals potens.
  - I stedet vælger vi at gøre det når vi når ned under en loadfaktor $\alpha_i < 1/4$, og så kan man faktisk vise at vi understøtter `Insert` og `Delete` i $O(1)$ amortiseret tid.
  - Vi vælger nu potentialemetoden:

$$\Phi(D_i) = \begin{cases} 2\text{num}_i - \text{size}_i & \text{if } \alpha_i \geq 1/2 \\ \text{size}_i/2 - \text{num}_i & \text{if } \alpha_i < 1/2 \end{cases} \tag{2}$$

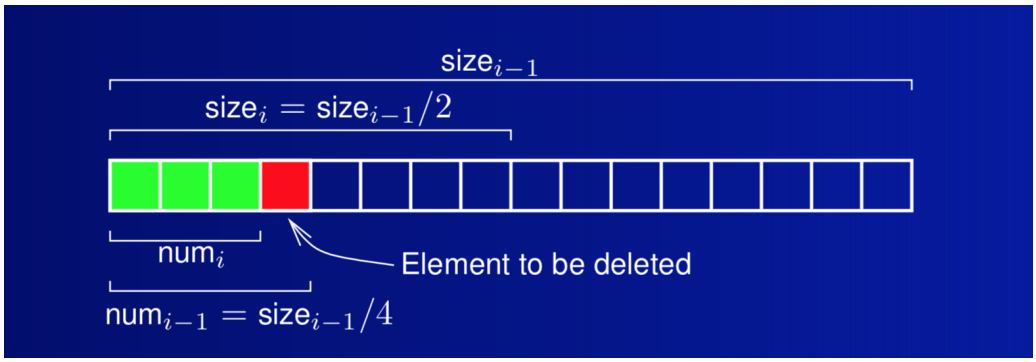
    Da får vi følgende graf:



Figur 2: Hvordan grafen vokser

- Vi observerer, at lige før en tabel ekspandering/forkortelse har Φ værdien num_i. Lige efter er tabellen cirka halv fuld og derfor dropper potentialet til omkring 0. Dette drop i potentiale "betaler" for ekspanderingen/forkortelsen.
- Når der ikke sker en ekspandering/forkortelse, så er den amortiserede cost højest 3 siden den absolutte hældning af Φ højest er 2.

- **Potentialemetoden - Bevis for amortiseret cost af `Delete`**
  - Hvis der ikke sker nogen tabelforkortelse, så kan vi ud fra grafen argumentere for, at $\hat{c}_i \leq 2$ da $c_i = 1$.
  - Hvis der SKER en tabelforkortelse, så vil ét element slettes og num_i elementer flyttes til det forkortede array, så den faktiske cost er $c_i = \text{num}_i + 1$. Vi har at loadfaktoren vil være under $1/2$ for både før og efter operationen, så vi bruger case 2 i Eq. (2) for begge.
  - Da får vi:



$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= \text{num}_i + 1 + (\text{size}_i/2 - \text{num}_i) - (\text{size}_{i-1}/2 - \text{num}_{i-1}) \\
&= \text{num}_i + 1 + ((\text{num}_i + 1) - \text{num}_i) - (2(\text{num}_i + 1) - (\text{num}_i + 1)) \\
&= 1
\end{aligned}$$

Husk igen på at vi skal have num_i *som eneste variabel, og ræsonner da ud fra figuren.*