Pensum potpurri

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2024

Litt om pseudokode

Pseudokode er formidling

- Pseudokode er *ikke et læringsmål i seg selv* for IN2010
- Det er et verktøy for å kunne formidle egen forståelse
- Pseudokoden er god dersom en (kvalifisert) leser forstår den
- Vi kan se for oss at pseudokode ligger langs en skala:

Python/Java  Naturlig språk

- Slideren er posisjonert *veldig omtrentlig* der vi ønsker (ta det med en klype salt)
- Gode eksempler finnes i slidene fra forelesning, boka og wikipedia

Pseudokode – Dos & don'ts

Gjør dette

- Bruk innrykk
- Bruk kontrollstrukturer
  - `if`, `while`, etc.
- Bruk fornuftige variabelnavn
  - evt. forklar navnene i teksten rundt
- Vær konsekvent
- Vær presis
- Vær konsis

Ikke gjør dette

- Ikke importer biblioteker
  - bruk *antagelser* i stedet
- Ikke vær *vag*
- Ikke vær språkspesifikk
  - Slicing i Python
  - `System.out.println`

Rød-svarte trær

Rød-svarte trær

- Rød-svarte trær er, i likhet med AVL-trær, *balanserte* binære søketrær
- Likhetene mellom rød-svarte- og AVL-trær er:
  - De er begge selvbalanserende binære søketrær
  - De har begge $\mathcal{O}(log(n))$ på innsetting, sletting og oppslag
  - De anvender begge rotasjoner for å bevare et krav om balanse
- Forskjellene mellom rød-svarte- og AVL-trær er:
  - Rød-svarte trær har *svakere* krav om balanse enn AVL-trær
  - Rød-svarte trær bruker *litt* mindre minne, siden vi ikke trenger å lagre høydene
  - Rød-svarte trær gjør færre rotasjoner

Rød-svarte trær vs. AVL-trær

- Rød-svarte trær er raskere enn AVL-trær når innsetting og sletting forekommer ofte, til sammenligning med oppslag
  - Dette er fordi rød-svarte trær trenger færre rotasjoner
- AVL-trær er raskere enn rød-svarte trær når oppslag forekommer ofte, til sammenligning med innsetting og sletting
  - Dette er fordi AVL-trær er mer balanserte

Invarianter for rød-svarte trær

1. Hver node fargelegges *rød* eller *svart* (derav navnet)
2. Roten til treet er svart
3. Alle tomme trær er svarte
4. En rød node kan ikke ha et rødt barn
5. Hver gren fra roten av treet til et tomt tre inneholder *like mange svarte noder*

Intuisjon

- Verste tilfellet (med hensyn til balanse) for et rødsvart tre er at vi har
  - en gren med bare svarte noder
  - en annen gren med annenhver svarte og røde noder
- Da har vi en gren som er dobbelt så lang som en annen!
  - Men dobbelt så langt er ikke så mye lenger
  - Så vi bevarer $\mathcal{O}(log(n))$ på innsetting, sletting og oppslag

Heapsort

Heapsort – Idé

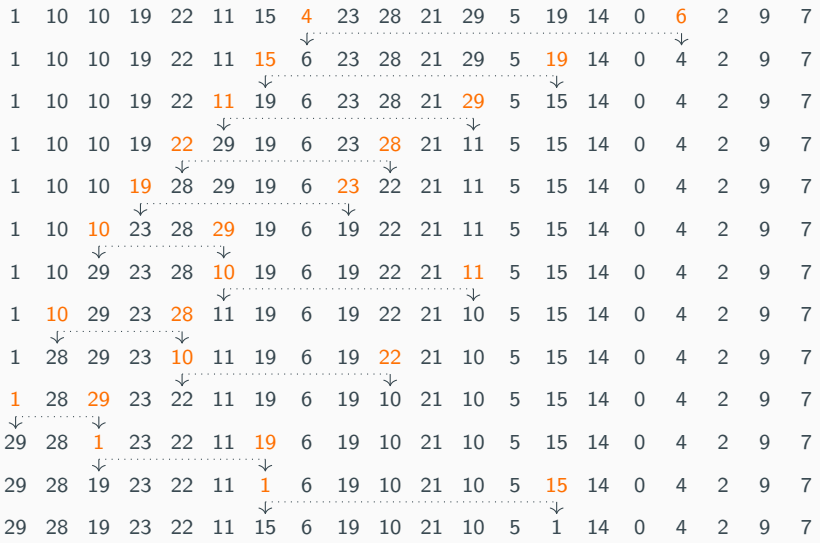


- Idéen bak heapsort er å bygge en heap og poppe elementer av heapen
- Fordi en heap kan implementeres med et array, gjør vi arrayet om til en heap
- Litt mer presist skal vi
  1. gjør arrayet om til en max-heap
  2. la i være $n - 1$ der n er størrelsen på arrayet
  3. pop fra max-heapen og plasser elementet på plass i
  4. senk i og gå til 3. frem til i blir 0

Heapsort – Bygge en max-heap

- Vi trenger å kunne gjøre et array om til en max-heap
- En max-heap er en heap der hver node er *større* enn begge barna
  - I motsetning til en min-heap der hver node er *mindre* enn begge barna
- En node svarer til en posisjon i arrayet
  - der roten ligger på plass 0,
  - venstre barn for noden på plass i ligger på plass $2i + 1$
  - og høyre barn for noden på plass i ligger på plass $2i + 2$
- `BuildMaxHeap` gjør et array om til en max-heap
  - Den starter i midten av arrayet og jobber seg mot venstre
  - Den bytter plass på foreldre- og barnenoder rekursivt der foreldrenoden er mindre enn (minst) en av barnenodene
  - Etter siste iterasjon vil det største elementet ligge i roten, og alle foreldrenoder er større (eller lik) begge barnenoder

Heapsort – Bygge en max-heap (eksempel)



Heapsort – Bygge en max-heap (implementasjon)

Algorithm: Hjelpeprosedyre for å bygge en max-heap

Input: En (uferdig) heap A med n elementer der i er roten
Output: En mindre uferdig heap

```
1  Procedure BubbleDown(A, i, n)
2      largest ← i
3      left ← 2i + 1
4      right ← 2i + 2
5
6      if left < n and A[largest] < A[left] then
7          largest ← left
8
9      if right < n and A[largest] < A[right] then
10         largest ← right
11
12     if i ≠ largest then
13         A[i], A[largest] ← A[largest], A[i]
14         BubbleDown(A, largest, n)
```

Algorithm: Bygg en max-heap

Input: Et array A med n elementer
Output: A som en max-heap

```
1  Procedure BuildMaxHeap(A, n)
2      for i ← ⌊n/2⌋ down to 0 do
3          BubbleDown(A, i, n)
```

Heapsort – Implementasjon

Algorithm: Heapsort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure HeapSort(A)
2     BuildMaxHeap(A, n)
3     for i ← n − 1 down to 0 do
4         A[0], A[i] ← A[i], A[0]
5         BubbleDown(A, 0, i)
6     return A
```

Heapsort – Kjøretidsanalyse (`BubbleDown`)

- Merk at linje 2–13 er konstanttidsoperasjoner
- Vi bryr oss bare om antall rekursive kall
- Algoritmen terminerer garantert når $i \geq \frac{n}{2}$
- Hvert rekursive kall øker i til $2i + 1$ eller $2i + 2$
- Altså *dobles* i for hvert rekursive kall
- Hvis en heap har høyde h inneholder den mindre enn 2^{h+1}
  - h er i $\mathcal{O}(\log(n))$, siden en heap er et *komplett* binærtre
- Dermed gjør vi maksimalt h rekursive kall
- Så `BubbleDown` er i $\mathcal{O}(\log(n))$

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
1  Procedure BubbleDown(A, i, n)
2      largest ← i
3      left ← 2i + 1
4      right ← 2i + 2
5
6      if left < n and A[largest] < A[left] then
7          largest ← left
8
9      if right < n and A[largest] < A[right] then
10         largest ← right
11
12     if i ≠ largest then
13         A[i], A[largest] ← A[largest], A[i]
14         BubbleDown(A, largest, n)
```

Heapsort – Kjøretidsanalyse (`BuildMaxHeap`)

- Vi gjør $\frac{n}{2}$ kall på `BubbleDown`
- Siden `BubbleDown` er $\mathcal{O}(\log(n))$ og vi gjør $\frac{n}{2}$ kall
  - *virker* dette som $\mathcal{O}(n \log(n))$
  - Men det *er faktisk* $\mathcal{O}(n)$!
  - Å vise dette er utenfor pensum
- Intuitivt er det fordi
  - $\frac{n}{2}$ noder vil ikke ha noen kall på `BubbleDown`
  - 2^{h-1} noder vil ha kun ett kall på `BubbleDown`
  - 2^{h-2} noder vil ha kun to kall på `BubbleDown`
  - ...
  - kun rotnoden vil kunne treffe verste tilfellet til `BubbleDown`

Algorithm: Bygg en max-heap

```
1 Procedure BuildMaxHeap(A, n)
2     for i ← ⌊n/2⌋ down to 0 do
3         BubbleDown(A, i, n)
```

Heapsort – Kjøretidsanalyse (`HeapSort`)

- Vi vet at `BuildMaxHeap` er i $\mathcal{O}(n)$
- Etter det gjør vi n iterasjoner
- For hver iterasjon kaller vi på `BubbleDown`
  - som er i $\mathcal{O}(\log(n))$
  - Her vil vi alltid kalle `BubbleDown` fra rotnoden
- Dette gir $\mathcal{O}(n \log(n))$ i kjøretidskompleksitet

Algorithm: Heapsort

```
1 Procedure HeapSort(A)
2     BuildMaxHeap(A, n)
3     for i ← n − 1 down to 0 do
4         A[0], A[i] ← A[i], A[0]
5         BubbleDown(A, 0, i)
6     return A
```

Huffman-koding

Huffman-koding

- Huffman-koding brukes for å *komprimere* data
- Du er gitt en mengde med symboler, kalt et *alfabet*
  - der hvert symbol har en gitt relativ *frekvens*
- Vi ønsker å representere hvert symbol med en bitstreng
  - slik at strenger over alfabetet blir så korte så mulig
- Vi kaller en slik mapping fra symboler til bitstrenger en *enkoding*
- Vi kaller disse bitstrengene *kodeord*

Huffman-koding – fast lengde

- Anta at vi jobber med bitstrenger av (fast) lengde n
- Med n bits kan vi representere 2^n forskjellige symboler
  - For å kunne representere m symboler trenger vi $\lceil log_2(m) \rceil$ bits
- Med en fast lengde n og en gitt streng s
  - brukes det $|s| \cdot n$ bits for å representere den

Huffman-koding – variabel lengde

- Som regel vil noen symboler forekomme oftere enn andre
- En Huffman-koding konstrueres på bakgrunn av den relative frekvensen til symbolene i alfabetet, slik at
  - symboler som forekommer ofte representeres med en relativt kort bitstreng
  - symboler som forekommer sjeldent representeres med en relativt lang bitstreng

Huffman-koding – variabel lengde (prefiks)

- Med bitstrenger av variabel lengde er det vanskeligere å vite når et symbol slutter og et annet begynner
- For å unngå tvetydighet må vi sørge for at ingen kodeord er *prefiks* av et annet
  - Ingen kodeord kan være en forlengelse av et annet
  - Hvis 010 er et kodeord kan 0001 være et kodeord
  - Hvis 010 er et kodeord kan *ikke* 0101 være et kodeord

Huffman-koding – frekvenstabell

«det er veldig vanskelig å finne på en eksempelsetning»

- Setningen over har følgende frekvenstabell

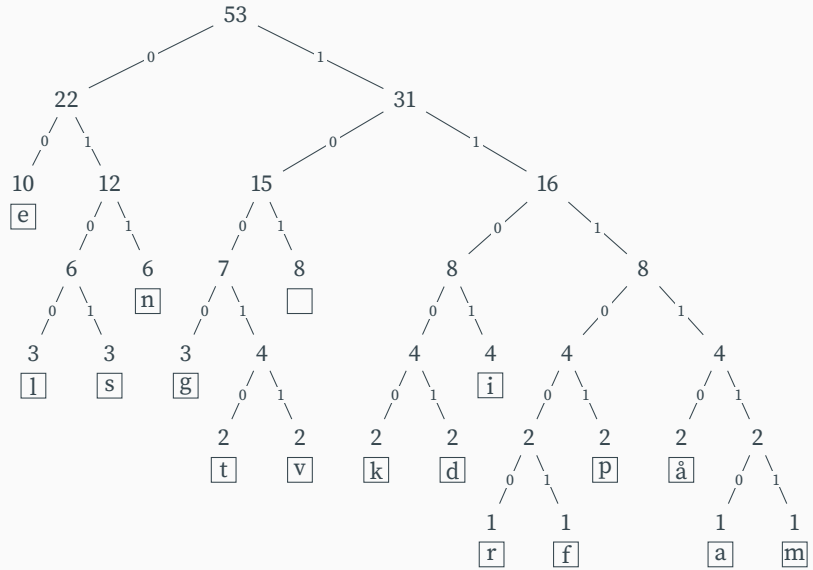
Symbol		a	d	e	f	g	i	k	l	m	n	p	r	s	t	v	å
Frekvens	8	1	2	10	1	3	4	2	3	1	6	2	1	3	2	2	2

- Med fast lengde trenger vi 5 bits for hvert symbol
  - Det gir $53 \cdot 5 = 265$ bits for å representere hele setningen
- Med huffman-koding trenger vi bare 198 bits
  - Dette er *optimalt*

Huffman-koding – Huffman trær

- En Huffman-koding konstrueres ved å bygge et binært tre
  - der hvert symbol i alfabetet forekommer som en løvnode
- Hver sti fra rot til løv gir opphav til et kodeord for det aktuelle symbolet
  - En gren mot venstre tolkes som 0
  - En gren mot høyre tolkes som 1
- Hver node v har en assosiert frekvens
  - som er gitt av summen av alle løvnoder som er etterfølgere av v sine frekvenser
- Symboler som forekommer sjeldent vil ligge dypere i det Huffman-treet enn symbolene som forekommer ofte

Huffman-koding – Bygge Huffman-tre (eksempel)



Huffman-koding – Bygge Huffman-trær

- Å bygge et Huffman-tre er ganske enkelt når man har en prioritetskø!
- Noder i et Huffman-tre har
  - Et element, samt venstre og høyre som vanlig
  - I tillegg en frekvens `freq` som nodene ordnes etter
- Algoritmen er som følger:
  - Opprett en tom prioritetskø
  - For hvert par av symbol og frekvens
    - Opprett en node (uten barn) og sett noden inn i køen
  - Så lenge det er mer enn ett element i køen
    - Fjern de to minste nodene v_1 og v_2
    - Lag en ny node u der v_1 og v_2 er barn av u og $u.\texttt{freq} = v_1.\texttt{freq} + v_2.\texttt{freq}$
    - Plasser u på køen
  - Til slutt returneres (den eneste) noden som ligger på køen

Huffman-koding – Bygge Huffman-trær (implementasjon)

ALGORITHM: BYGGE HUFFMAN TRÆR

Input: En mengde C med par (s, f) der s er et symbol og f er en frekvens
Output: Et Huffman-tre

```
1  Procedure Huffman(C)
2      Q ← new PriorityQueue
3      for (s, f) ∈ C do
4          Insert(Q, new Node(s, f, null, null))
5      while Size(Q) > 1 do
6          v1 ← RemoveMin(Q)
7          v2 ← RemoveMin(Q)
8          f ← v1.freq + v2.freq
9          Insert(Q, new Node(null, f, v1, v2))
10     return RemoveMin(Q)
```