

Øvingsforelesning 10

Grådighet

Benjamin Bjørnseth

Program

- Denne ukens øving
- Grådighet
- Forrige ukes øving

Øving 10 - Teori

- Oppgave 1 - Generelt
- Oppgave 2 - Antall veier
 - Bruk DP - rekursjon
- Oppgave 3 - 0-1-ryggsekk
 - God idé å bruke DP for de store problemene

Øving 10 - Praksis

- Det jobbes med en å lage en ny øving
- Enten blir den lagt ut snart, eller så fremskyndes en annen øving

Program - Grådighet

- Grådighet vs DP
- Hvordan bevise grådighet
- Huffman-koding
- Eksempler

Grådighet vs DP

- For å bruke DP har problemer (gjerne)
 - Optimal substruktur
 - Overlappende delproblemer
- For å bruke en grådig algoritme har problemer (gjerne)
 - Optimal substruktur
 - 'Grådig valg'-egenskapen
- DP - sjekk konsekvensene av alle valg
 - Gir overlappende delproblemer
- Grådig algoritme - ta det som ser best ut
 - Trenger 'grådig valg'-egenskapen

Eksempel - Korteste vei

- Kan bruke dynamisk programmering
 - $d[t, i] = \min \{ d[x, i-1] + w(x, t) \}$
- Sjekker vekt for alle veilengdebeskrankninger og alle forgjengernoder
 - 'Relax'er en kant $|V| - 1$ ganger
- Kan vi tillate oss å være grådige?
 - Grådig mhp i: Kjør 'Relax' kun fra noder vi kjenner korteste vei til
 - => Dijkstra!

DP - oppfriskning

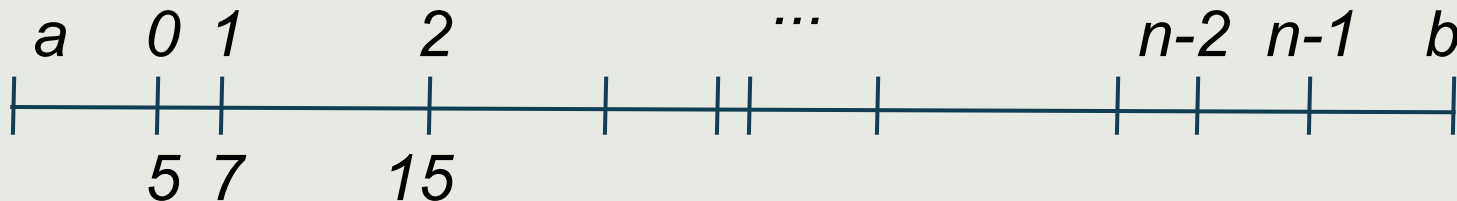
1. Beskriv optimal substruktur
 - a. Beskriv løsning i å bestå av å ta et valg
 - b. Anta at valget som gir optimal løsning er gitt
 - c. Vis at etter valget har vi nye delproblem å løse
 - d. Bevis at disse må løses optimalt
2. Beskriv rekursiv løsning
3. Beregn optimal løsningsverdi
4. Eventuelt: finn optimal løsning

Grådige algoritme

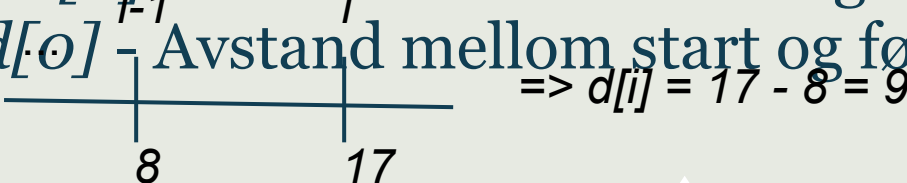
1. Beskriv løsning i å bestå av å ta et valg
2. Bevis at et grådige valg er trygt
3. Bevis optimal substruktur
4. Bruk rekursjonen direkte for å beregne løsning

Eksempel: Matstasjonsbesøk

- Skal gå på ski fra a til b
- Langs veien er det n matstasjoner, angitt av distanse langs løypa
- Vi kan gå m kilometer før vi trenger mat
 - Må ha spist før vi har gått m kilometer
 - Er aldri mer enn m kilometer mellom to stasjoner
- Hvor burde vi stoppe?



Terminologi

- Problem: Hva er færrest antall stopp ved matstasjoner $\{0, \dots, n-1\}$?
 - n - Antall matstasjoner
 - $s[i, l]$ - Færrest antall stopp ved matstasjoner $\{0, \dots, i-1\}$, når det er l kilometer til neste stopp
 - $s[0]$ - Start, $s[n]$ - siste målstasjon
 - $d[i]$ - Avstand mellom matstasjon i og $i - 1$
 - $d[n]$ - Avstand mellom mål og siste matstasjon
 - $d[0]$ - Avstand mellom start og første matstasjon
- 
$$\Rightarrow d[i] = 17 - 8 = 9$$

Matstasjoner - Dynamisk programmering

1. Beskriv optimal substruktur
 - a. Valg: Skal vi stoppe i matstasjon $i-1$ med l kilometer til neste stopp?
 - b. Valg gitt
 - i. $x = 0$: Stopper
 - ii. $x = 1$: Stopper ikke
 - c. Løsning er nå
 - i. x
 - ii. Antall stopp i matstasjoner $\{0, \dots, i-2\}$, gitt valget x
 - d. Antall stopp i matstasjoner $i-1$ med $d[i] + l * x$ kilometer til neste stopp må være optimalt

Matstasjoner - Dynamisk programmering

2. Rekursjon

$$s[i, l] = \begin{cases} 0, & i = 0 \\ \infty, & l > m \\ \min \left\{ \begin{array}{l} s[i-1, l + d[i-1]], \\ s[i-1, d[i-1]] \end{array} \right\} + 1, & \text{ellers} \end{cases}$$

Matstasjoner - Dynamisk programmering

3. Beregn løsning

- a. Bruk rekursjon for hver matstasjon for hver mulige l

Matstasjoner - Grådig

1. Beskriv valg
2. Vis at et grådig valg er trygt

To situasjoner:

- Hvis $l + d[i] \leq m$, stopper vi ikke i i
- $l + d[i] > m$, stopper vi i i

Situasjon #1

- Anta at én optimal løsning stopper her
- La neste stopp for optimal løsning være k



Situasjon #1

- Anta at én optimal løsning stopper her
- La neste stopp for optimal løsning være k
- Kan konstruere ny optimal løsning ved å bytte ut stopp i stasjon i med et tillatt stopp blant stasjoner $\{k+1, \dots, i-1\}$



Situasjon #2

- Anta at vi har en optimal løsning som ikke stopper her
 - Optimal løsning = ∞
 - Ikke mulig optimal løsning

Matstasjoner - Grådig

- Optimal substruktur
 - Etter grådig valg, står vi igjen med delproblem $\{0, \dots, i-2\}$
 - Optimal løsning består nå av
 - Det grådige valget
 - Optimal løsning av delproblemet
 - Bevis: Klipp-og-lim

DP vs grådig

```
$ time python dp_food.py < foodinput.txt  
input size 100000  
number of stations: 67928  
real 0m1.233s
```

```
$ time python greedy_food.py < foodinput.txt  
input size 100000  
number of stations: 67928  
real 0m0.079s
```

Standard bevis av grådlig egenskap

- Anta at du har en optimal løsning hvor det grådige valget ikke er tatt
- Vis hvordan du kan lage en ny optimal løsning ved å bytte ut et annet valg med det grådige valget
 - Litt mer involvert enn klipp-og-lim

Eksempel: Huffman-koding

- Komprimeringsalgoritme
 - Har en sekvens av bokstaver
 - Ønsker en plasseffektiv bitrepresentasjon
- Standard bokstavkoding: ASCII, UTF-8
 - Samme antall bits per bokstav
 - E.g. a = 1100001, b = 1100010, ...
- Huffman:
 - Kjenner forekomst av hver bokstav
 - Ønsker koder som gir færrest bits totalt
 - Må kunne dekode tilbake til bokstaver
=> prefikskoder

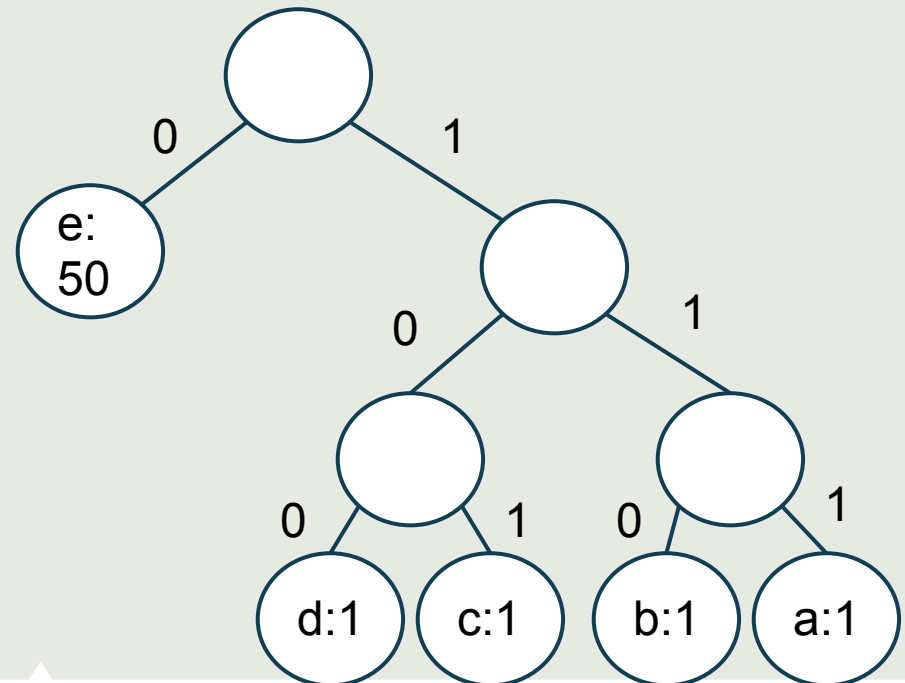
Grådig koding

- Hva med å gi den hyppigst forekommende bokstaven bitlengde 1, nest hyppigst 2, etc?
 - Tekst har forekomster a: 1, b: 1, c: 1, d: 1, e: 50
 - $e = 0$, $d = 10$, $c = 110$, $b = 1110$, $a = 1111$
 $\Rightarrow 50 + 2 + 3 + 4 + 4 = 63$ bits totalt
 - $e = 0$, $d = 100$, $c = 101$, $b = 110$, $a = 111$
 $\Rightarrow 50 + 3 + 3 + 3 + 3 = 62$ bits totalt
- Fungerer altså ikke

Huffmans grådighet

- Huffman er omvendt grådig
 - Gi lengste streng til de som forekommer sjeldnest
 - Kjenner ikke lengste streng før slutten
 - => Konstruerer trerepresentasjon av kodingen, fra bunnen og opp

Et slikt tre er også
kjekt til dekodning
av komprimert
tekst



Huffmankoding - Fremgangsmåte

- Ta de to nodene med lavest forekomst
 - Kall dem a og b
- Konstruer en ny node, med venstre barn a og høyre barn b og forekomst lik summen av forekomster av a og b
- Repeter til vi bare har en node igjen

Huffmankoding - Fremgangsmåte

e:50

d:1

c:1

b:1

a:1

Huffmankoding - Fremgangsmåte

e:50

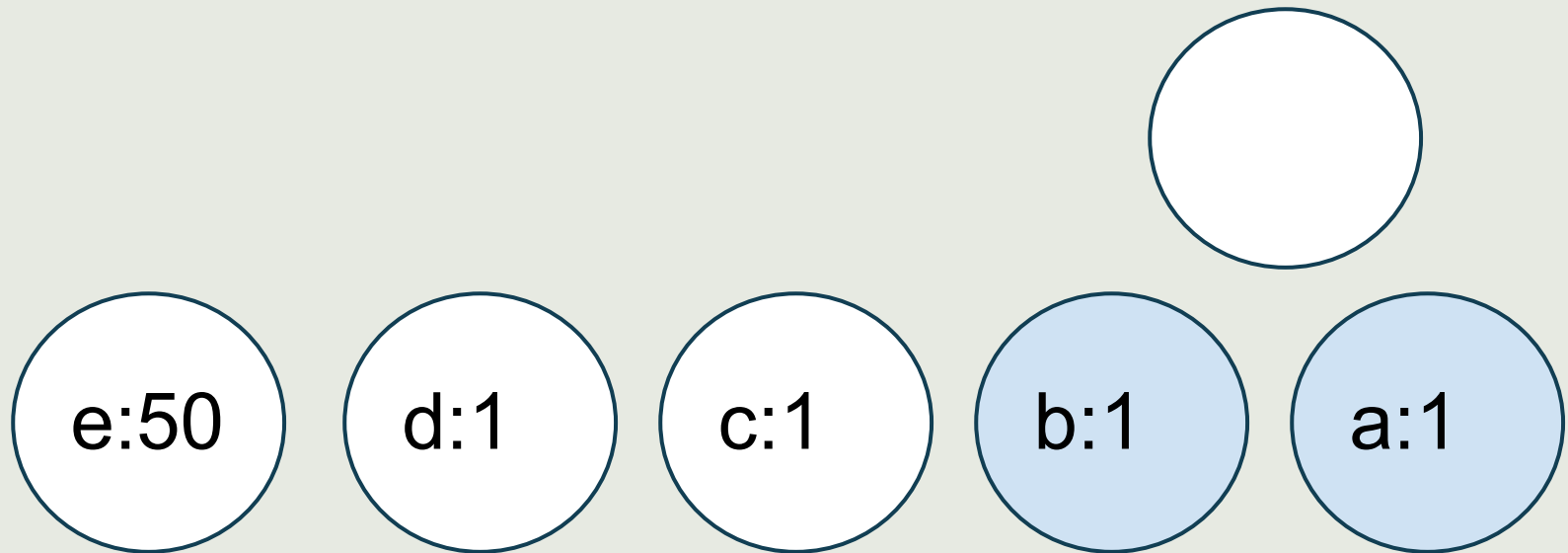
d:1

c:1

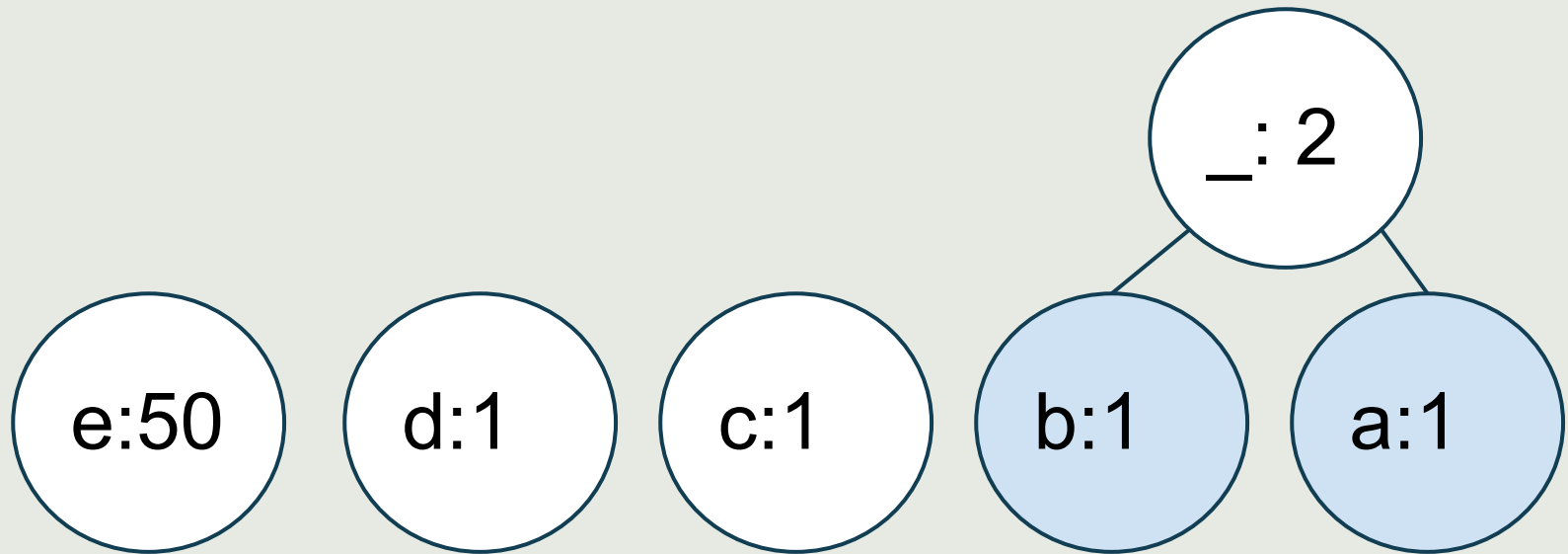
b:1

a:1

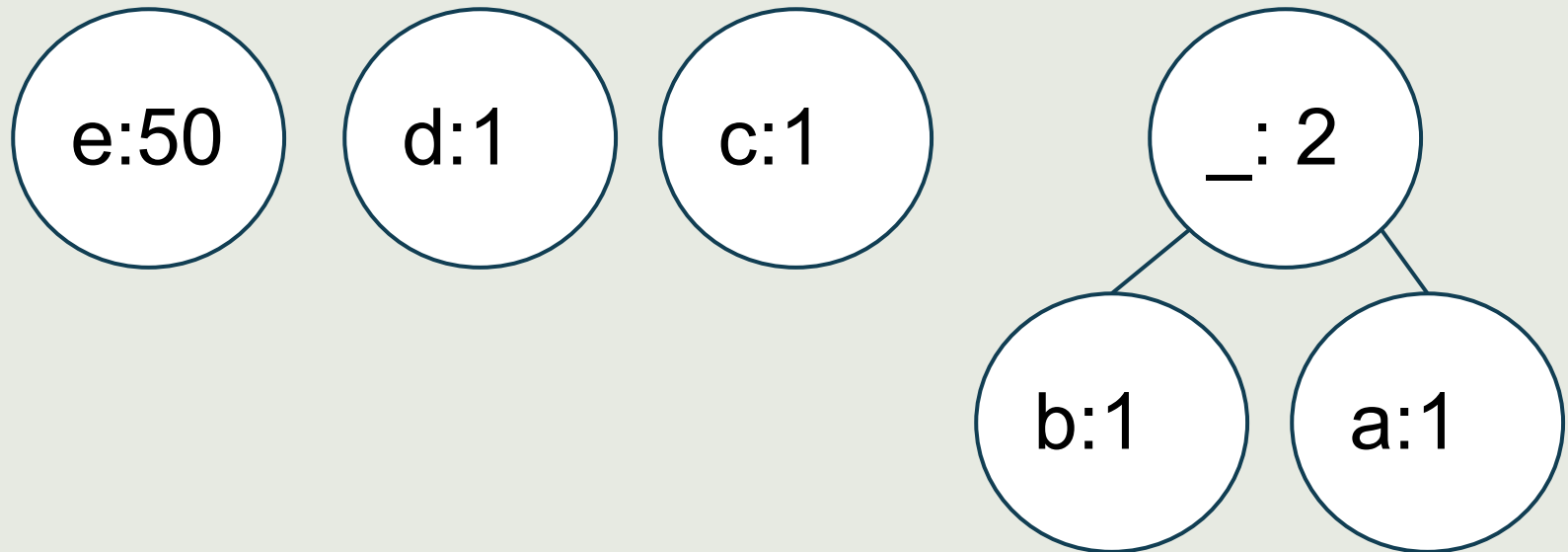
Huffmankoding - Fremgangsmåte



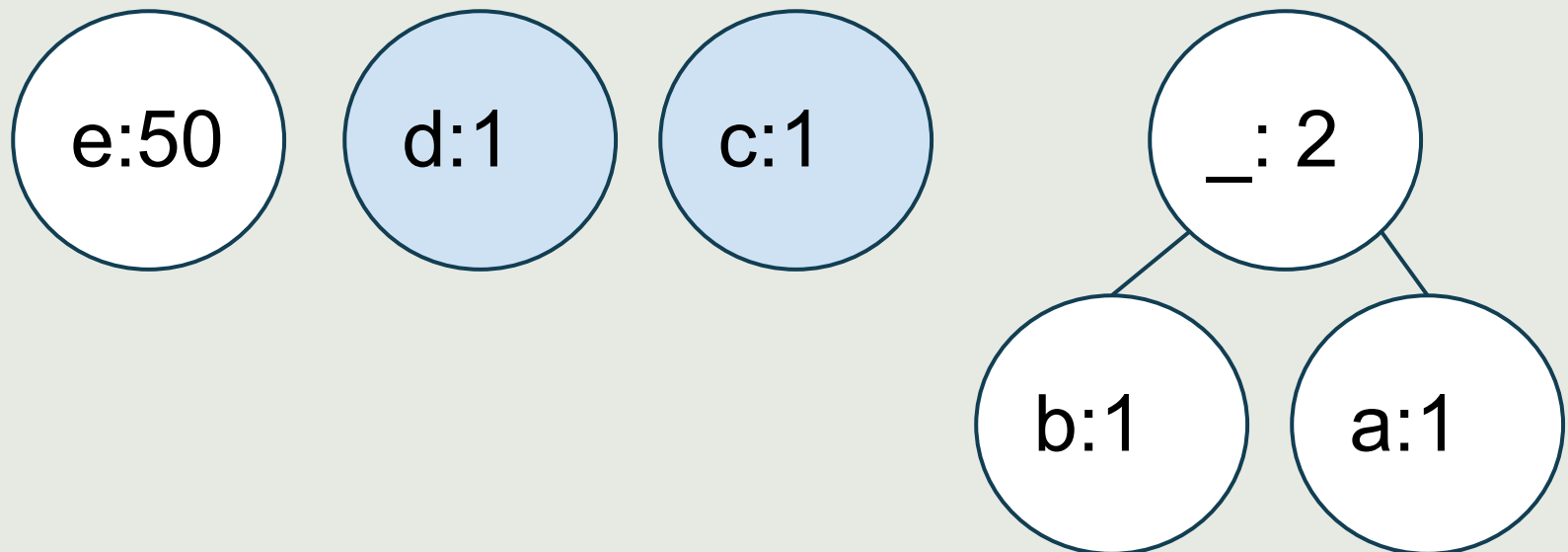
Huffmankoding - Fremgangsmåte



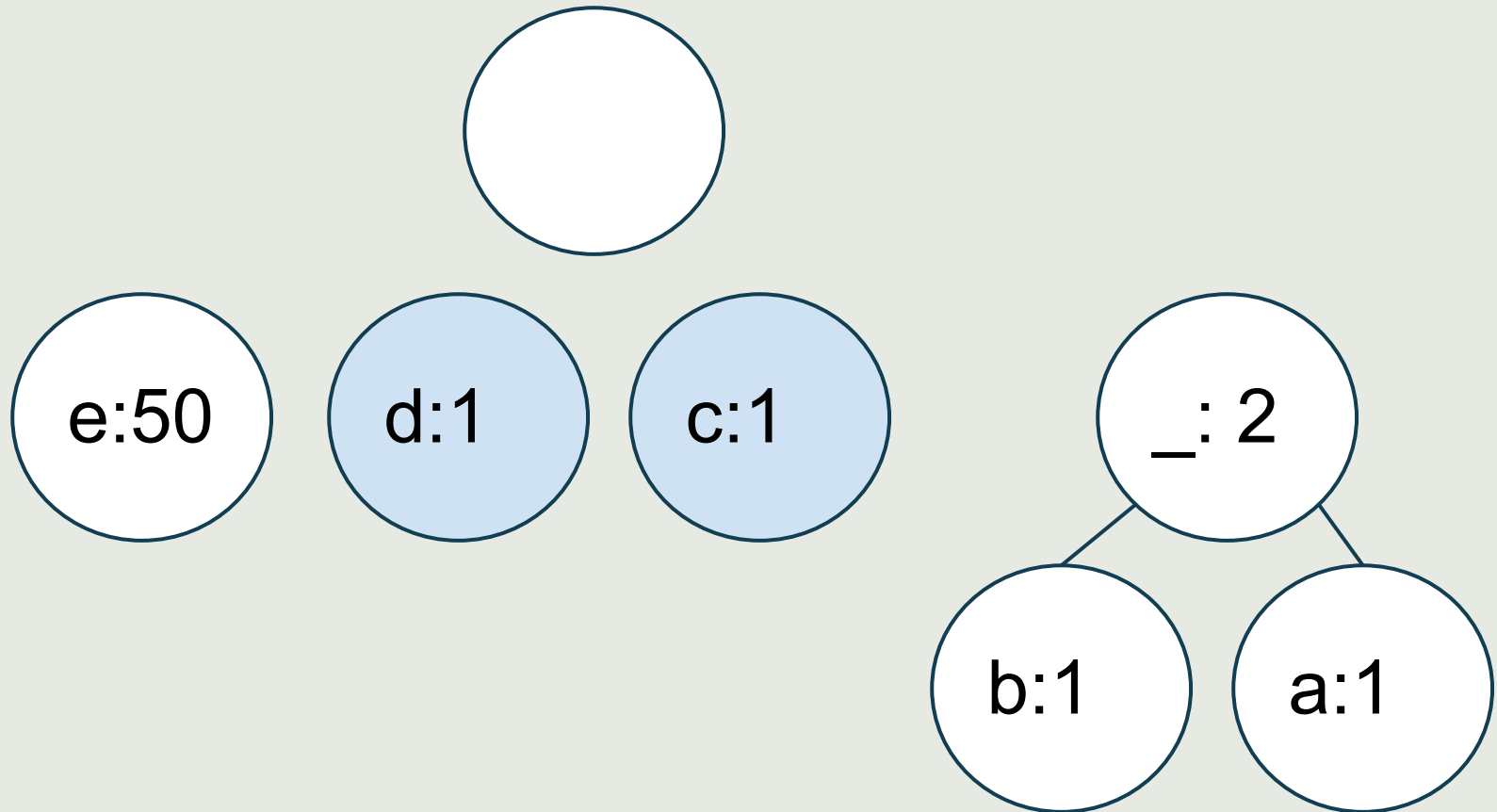
Huffmankoding - Fremgangsmåte



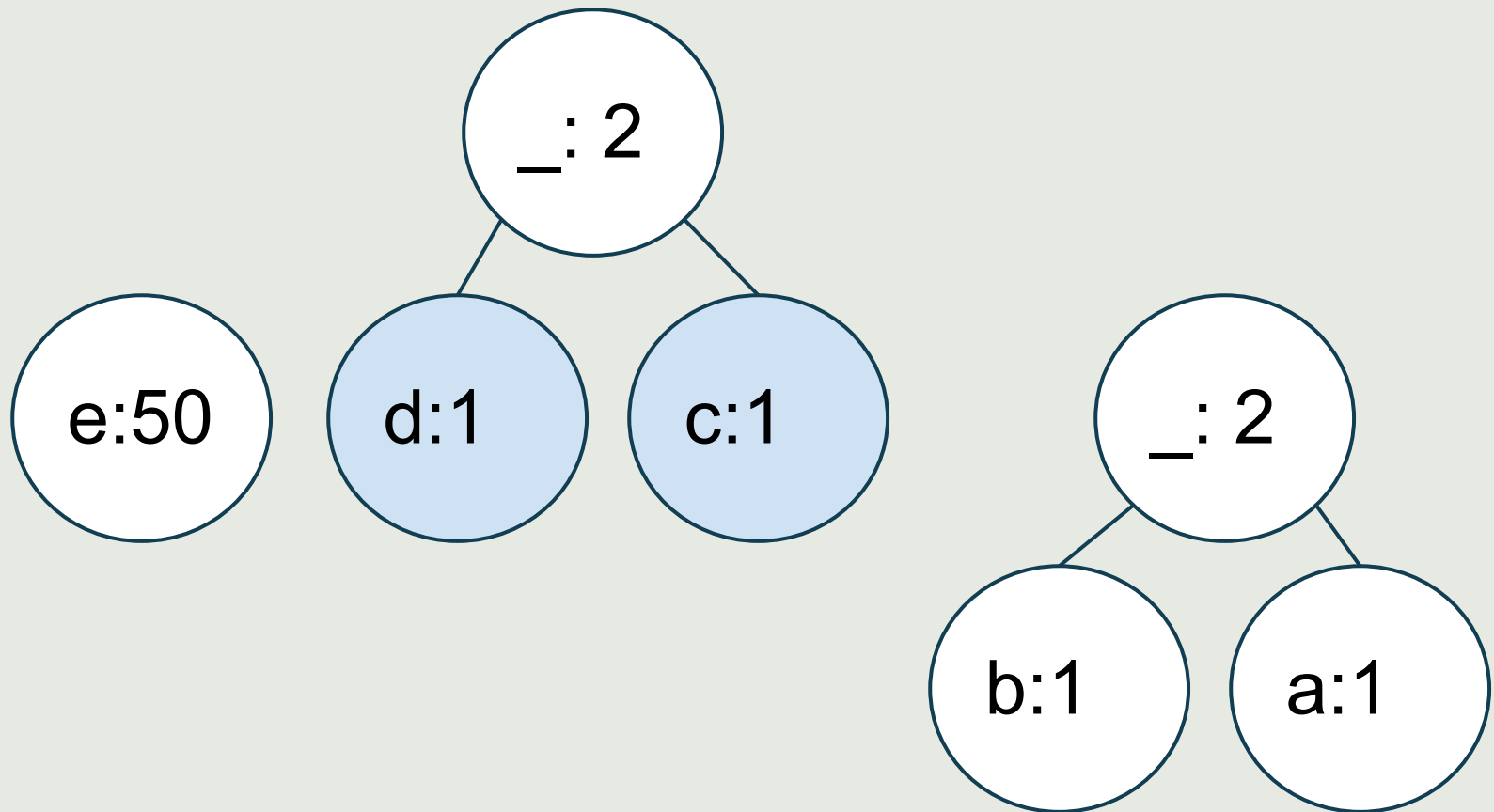
Huffmankoding - Fremgangsmåte



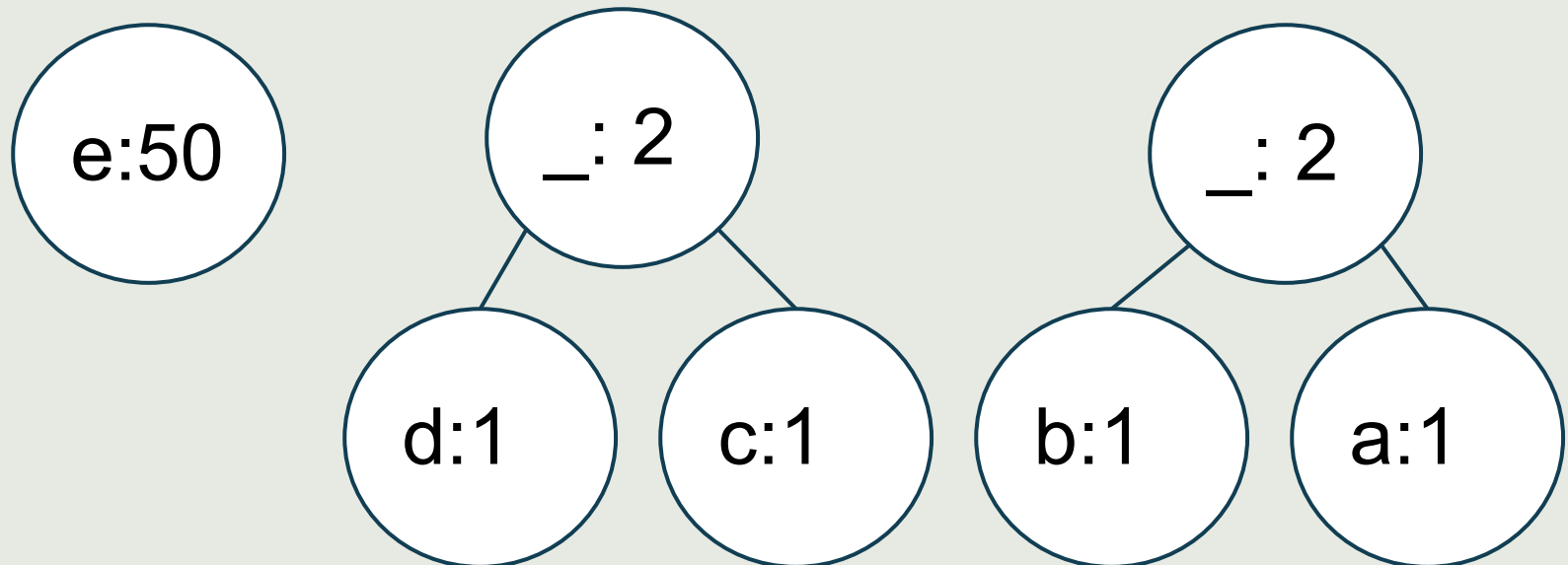
Huffmankoding - Fremgangsmåte



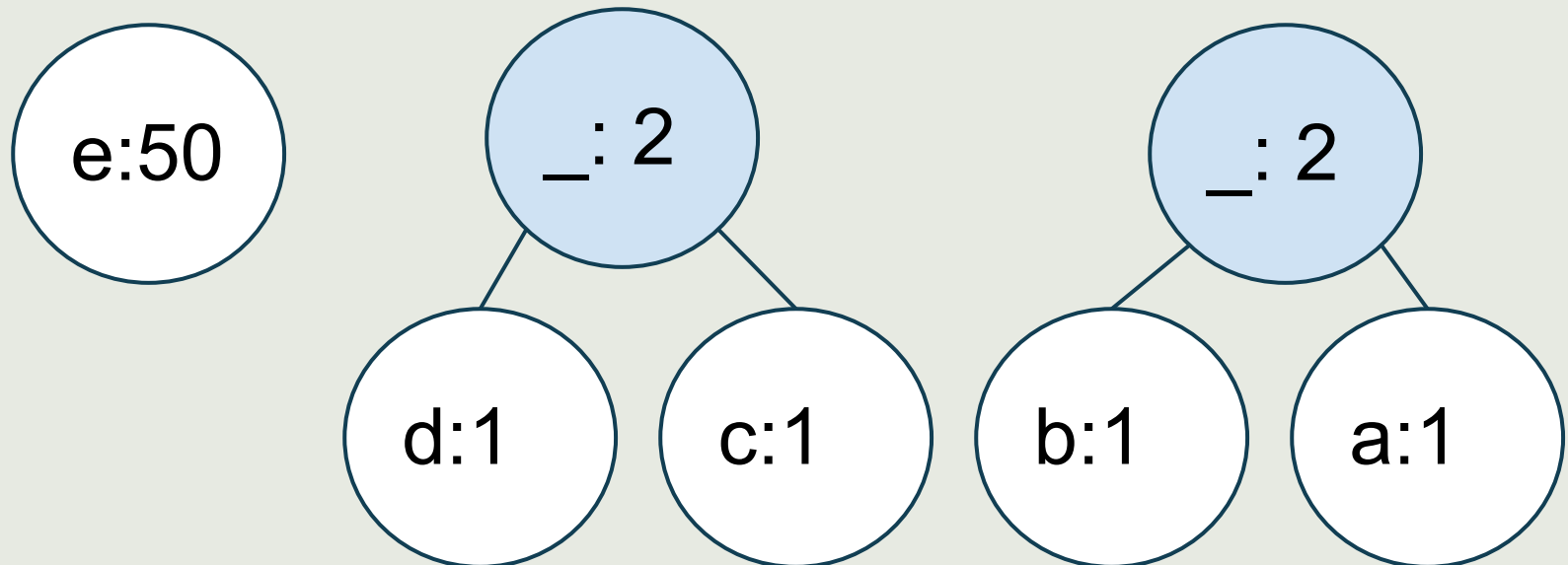
Huffmankoding - Fremgangsmåte



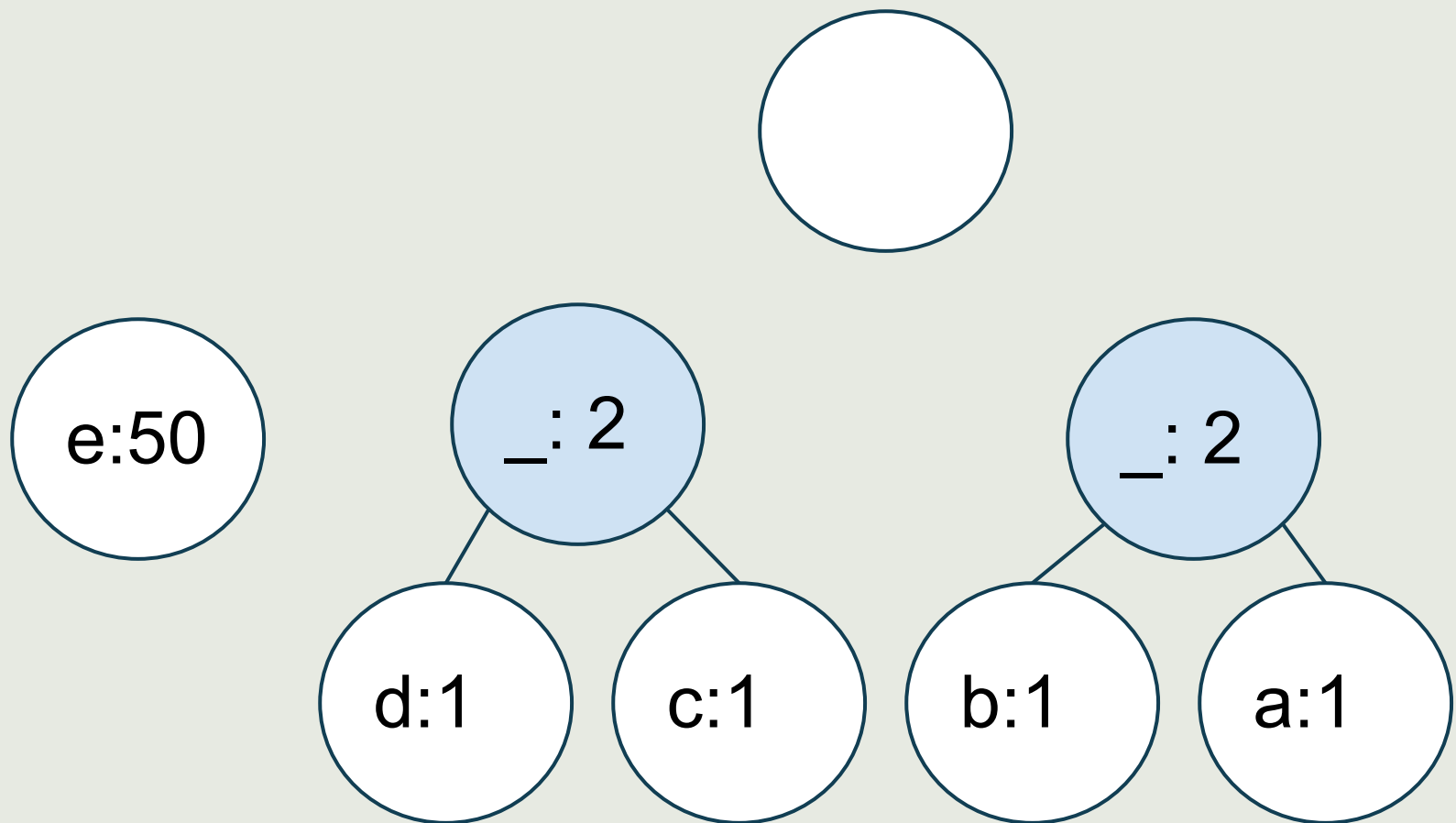
Huffmankoding - Fremgangsmåte



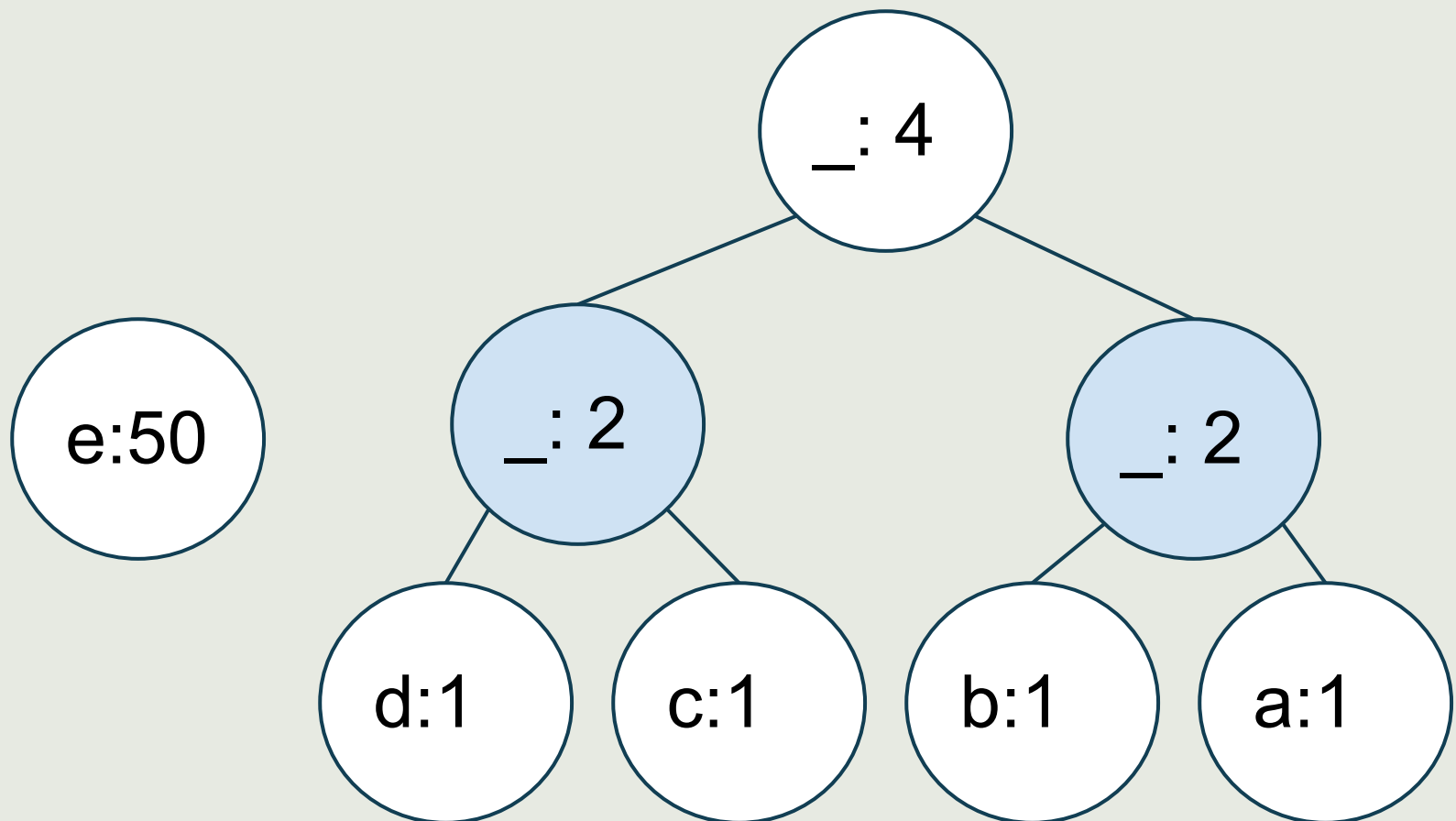
Huffmankoding - Fremgangsmåte



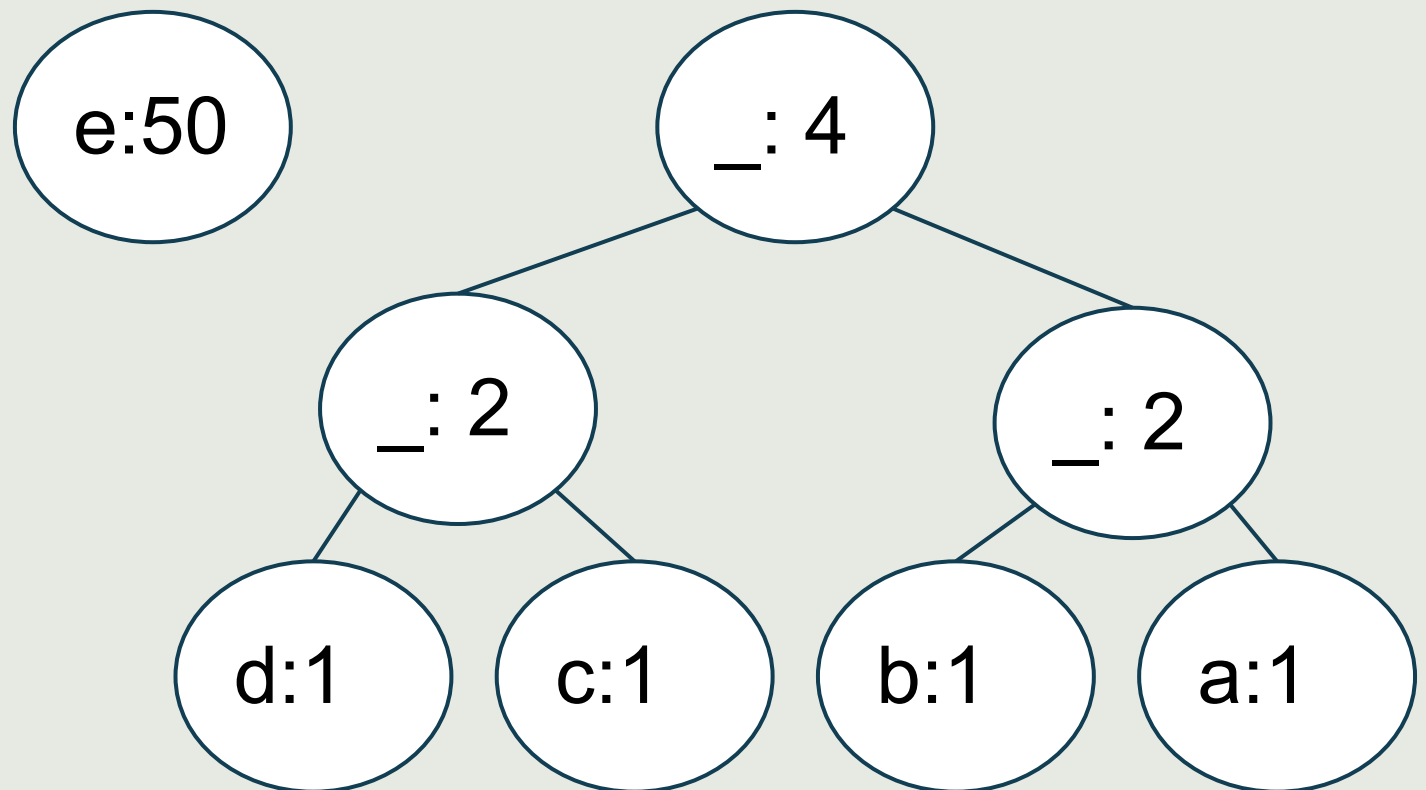
Huffmankoding - Fremgangsmåte



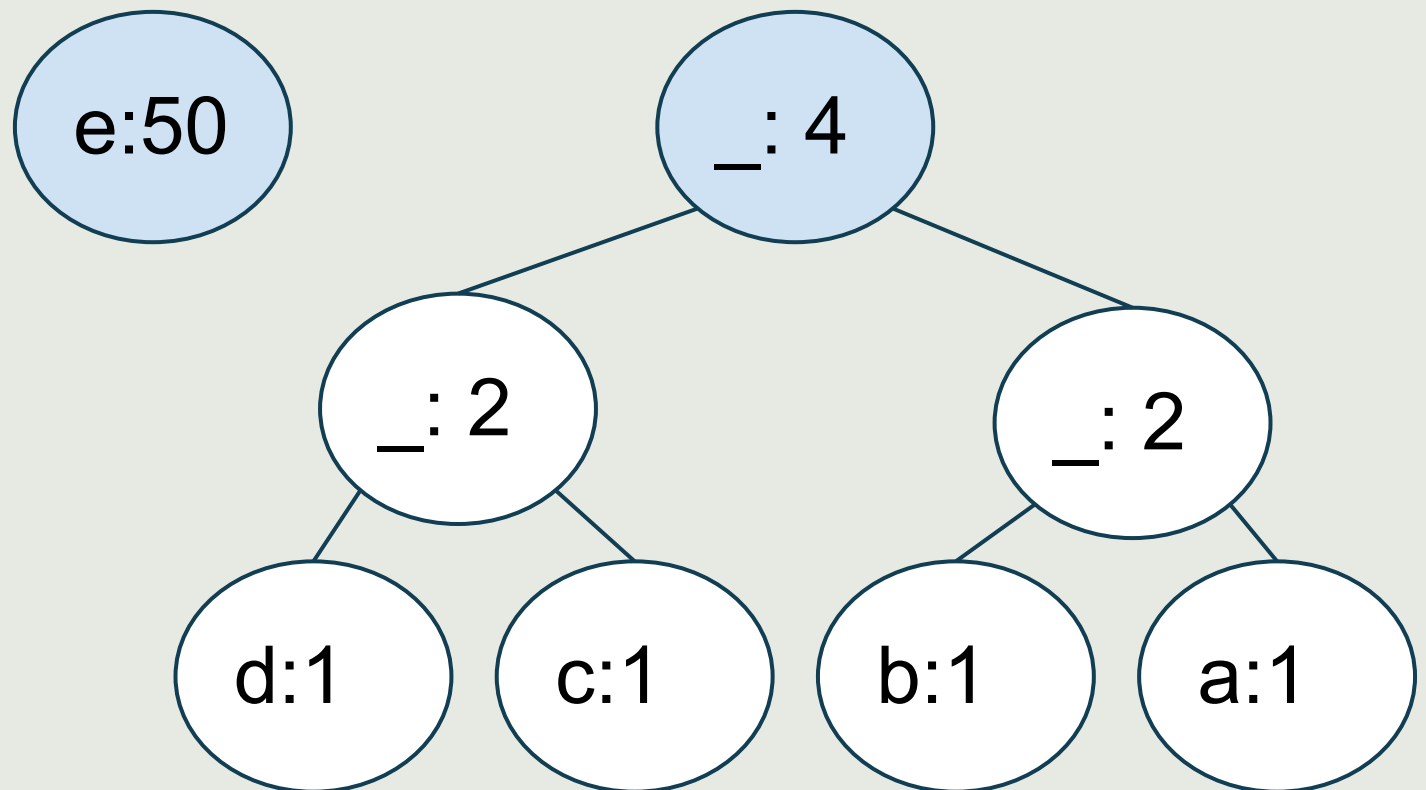
Huffmankoding - Fremgangsmåte



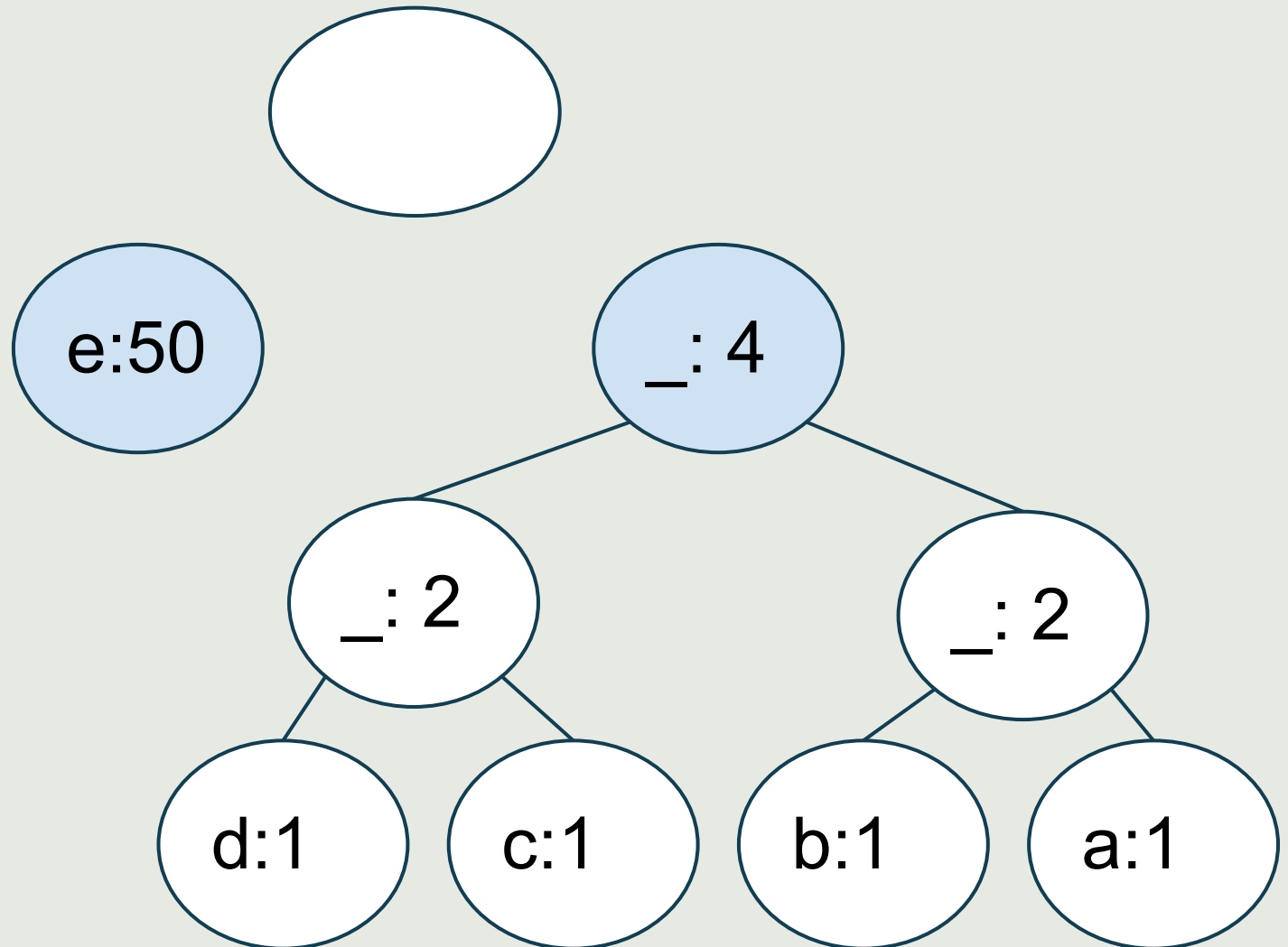
Huffmankoding - Fremgangsmåte



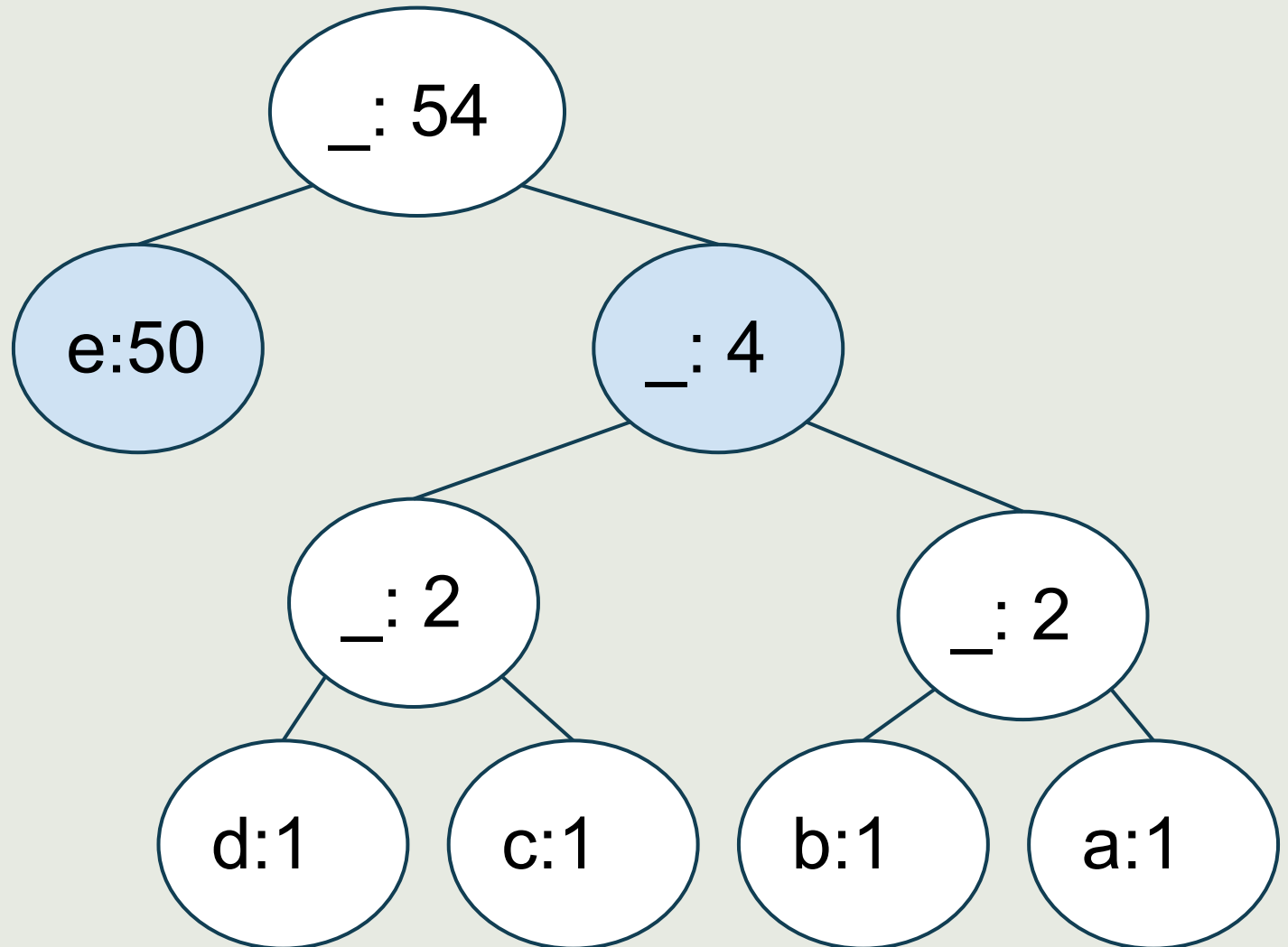
Huffmankoding - Fremgangsmåte



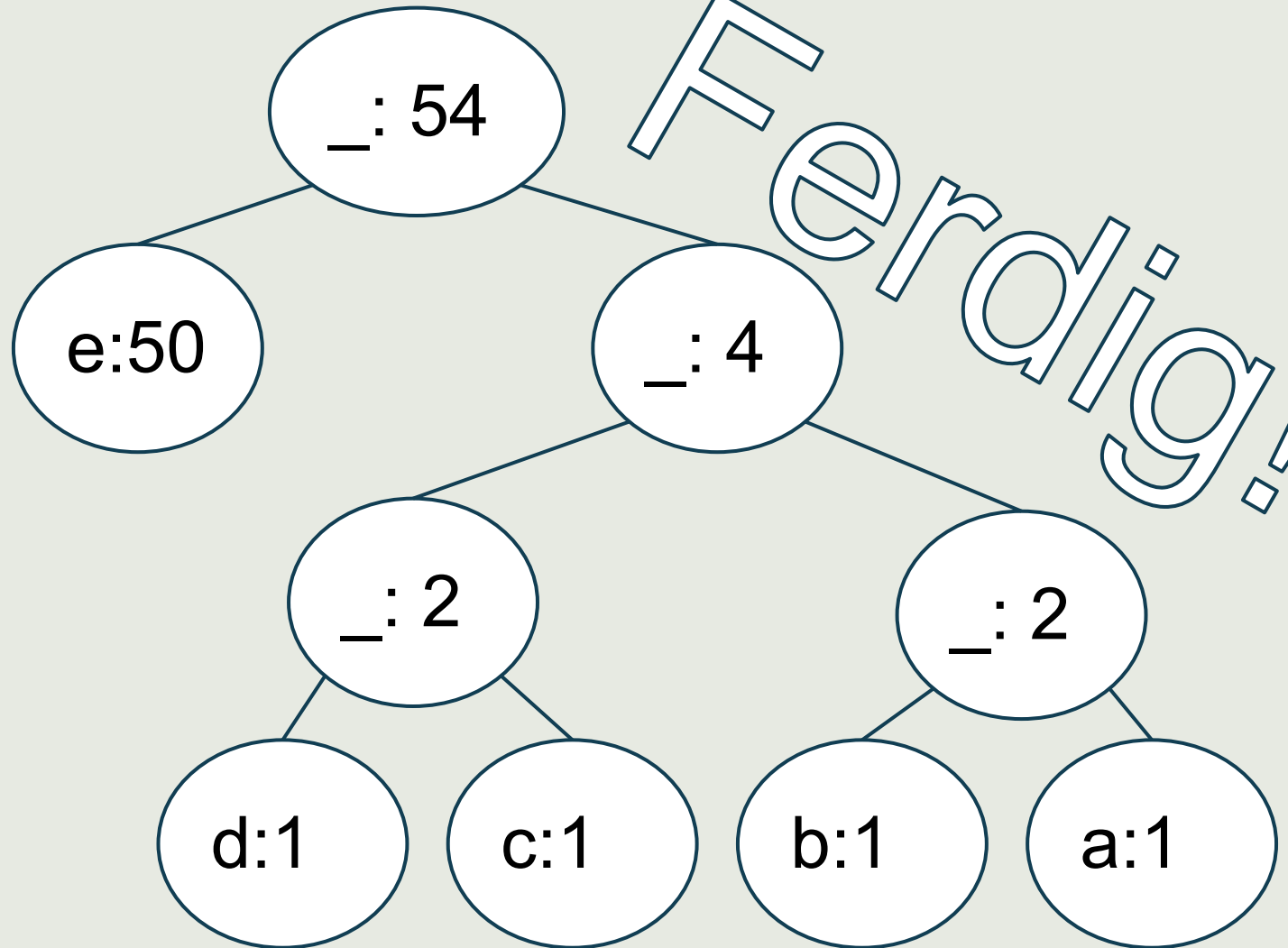
Huffmankoding - Fremgangsmåte



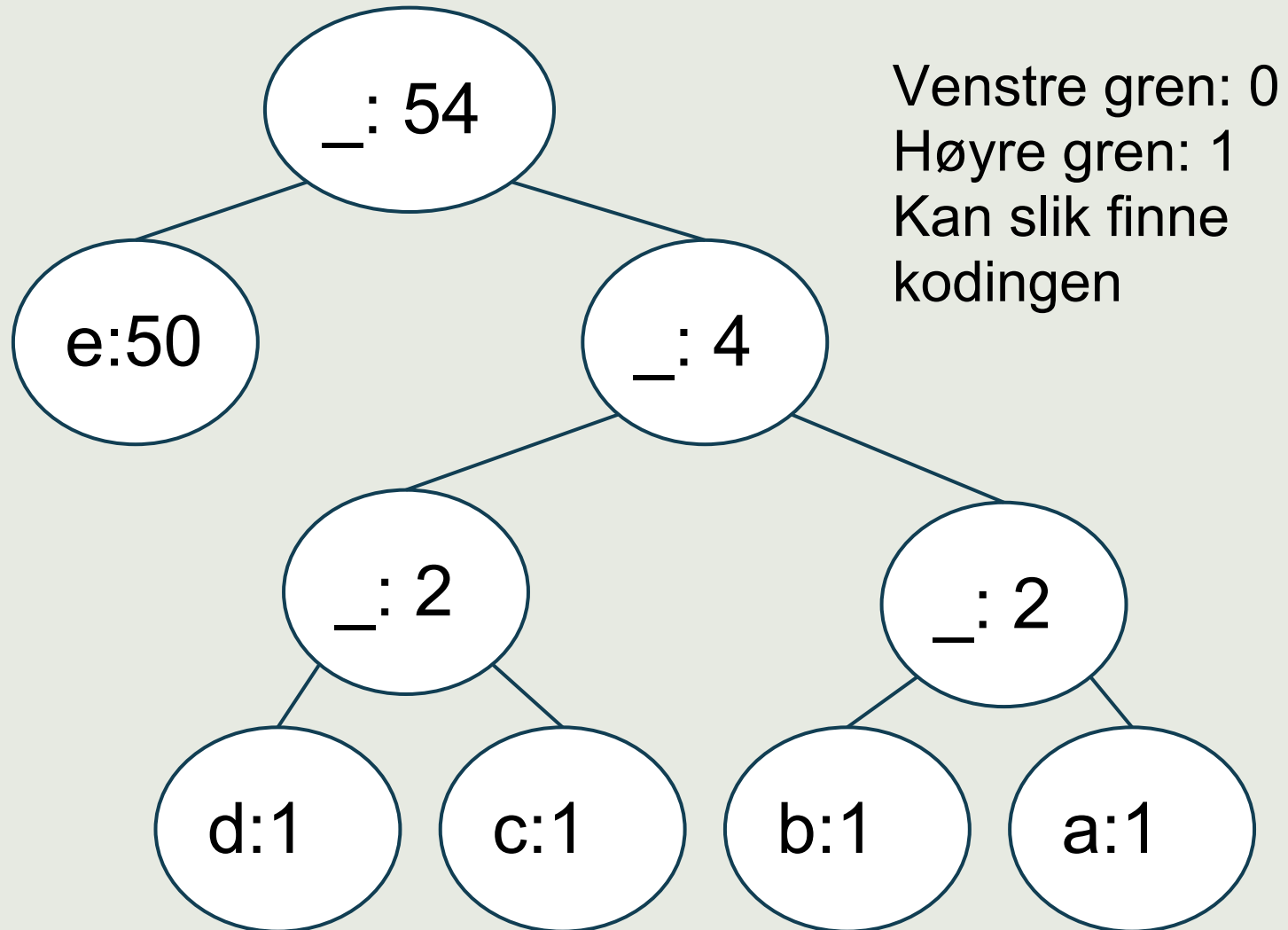
Huffmankoding - Fremgangsmåte



Huffmankoding - Fremgangsmåte



Huffmankoding - Fremgangsmåte



Huffmanalgoritmen

```
def huffman(frequencies):  
    """ frequencies: Dictionary av  
        'bokstav':forekomst """
```

Huffmanalgoritmen

```
def huffman(frequencies):  
    for (k, v) in frequencies.items():  
        # frequencies.items() gir innholdet  
        # som (nøkkel, verdi)-tupler
```

Huffmanalgoritmen

```
def huffman(frequencies):  
    [... for (k, v) in frequencies.items()]  
        # list-comprehension  
        # ett element i lista for hvert  
        # element i items
```

Huffmanalgoritmen

```
def huffman(frequencies):  
    [Node(f=v, v=k) for (k, v) \  
        in frequencies.items()]  
    # Konstruer en ny node per element  
    # f: frekvens, v: bokstav  
    # '\' betyr 'fortsetter på neste linje'
```

Huffmanalgoritmen

```
def huffman(frequencies):  
    alphabet = [Node(f=v, v=k) for (k, v) \  
                in frequencies.items()]  
    # lagre listen i 'alphabet'
```

Huffmanalgoritmen

```
def huffman(frequencies):  
    alphabet = [Node(f=v, v=k) for (k, v) \  
                in frequencies.items()]  
    Q = Heap(alphabet)  
    # Lag en heap av alle nodene
```

Huffmanalgoritmen

```
def huffman(frequencies):  
    alphabet = [Node(f=v, v=k) for (k, v) \  
                in frequencies.items()]  
    Q = Heap(alphabet)  
    while len(Q) > 1:  
        # Vi har minst to noder igjen
```

Huffmanalgoritmen

```
def huffman(frequencies):
    alphabet = [Node(f=v, v=k) for (k, v) \
                in frequencies.items()]
    Q = Heap(alphabet)
    while len(Q) > 1:
        a, b = Q.extract_min(), \
              Q.extract_min()
        # Hent de to sjeldneste nodene
        # (i python: x, y = 1, 2
        #           setter x = 1, y = 2)
```

Huffmanalgoritmen

```
def huffman(frequencies):  
    alphabet = [Node(f=v, v=k) for (k, v) \  
                in frequencies.items()]  
    Q = Heap(alphabet)  
    while len(Q) > 1:  
        a, b = Q.extract_min(), \  
              Q.extract_min()  
        z = Node(left=a, right=b)  
        # Konstruer en ny node  
        # Node()-konstruktør setter opp  
        # frekvens til å være sum av a og b
```

Huffmanalgoritmen

```
def huffman(frequencies):  
    alphabet = [Node(f=v, v=k) for (k, v) \  
                in frequencies.items()]  
    Q = Heap(alphabet)  
    while len(Q) > 1:  
        a, b = Q.extract_min(), \  
              Q.extract_min()  
        z = Node(left=a, right=b)  
        Q.insert(z)  
    # Legg den nye noden inn i heapen
```

Huffmanalgoritmen

```
def huffman(frequencies):
    alphabet = [Node(f=v, v=k) for (k, v) \
                in frequencies.items()]
    Q = Heap(alphabet)
    while len(Q) > 1:
        a, b = Q.extract_min(), \
              Q.extract_min()
        z = Node(left=a, right=b)
        Q.insert(z)
    return Q.extract_min()
# Kun rotnoden igjen i heapen
# Returner rotnoden
```

Kjøretid

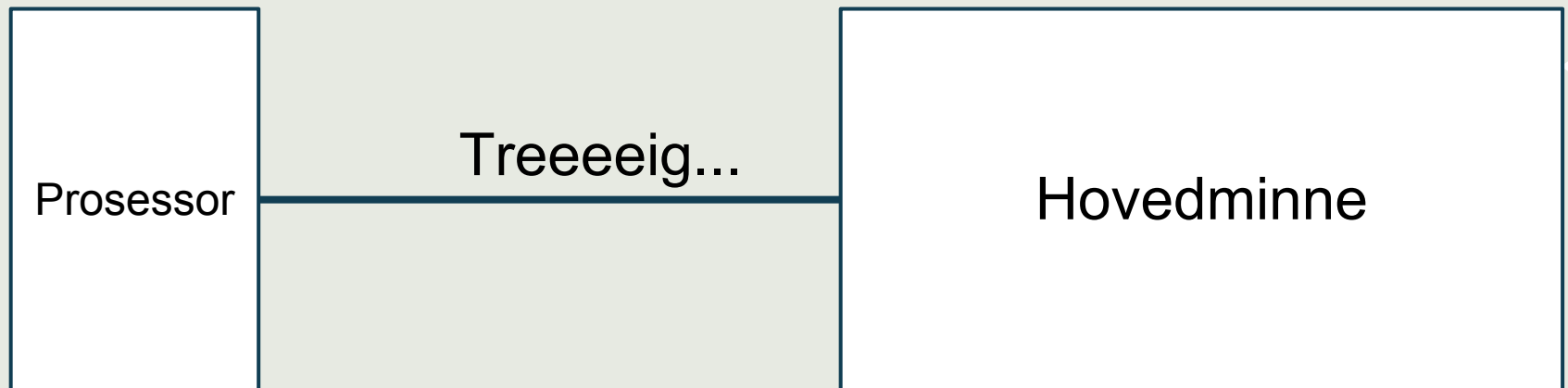
- Kjører hovedløkka n ganger
 - n - antall bokstaver
- Inne i løkka:
 - 2 extract-min - $2 * O(\lg(n)) = O(\lg(n))$
 - Konstruerer node - $O(1)$
 - heap-insert - $O(\lg(n))$
- Totalt: $O(n \lg(n))$

Huffman - bevis?

- Se lærebok
- Se forelesningsfoiler

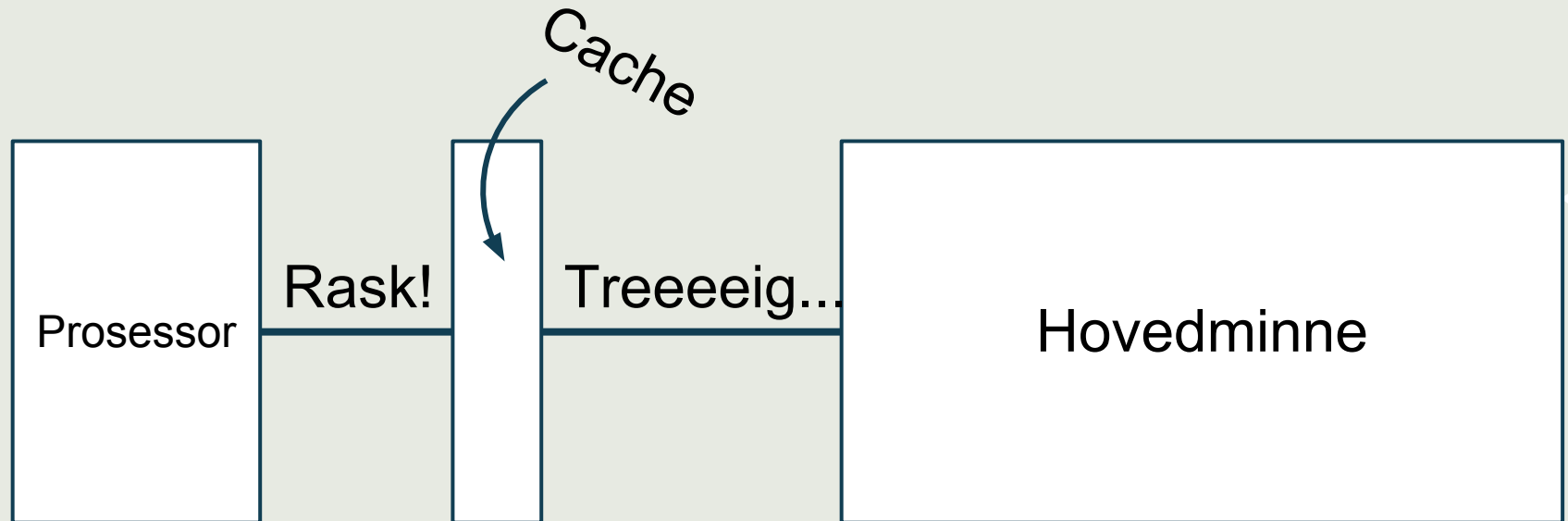
Eksempel: Cache-håndtering

- Cache: Mellomlager



Eksempel: Cache-håndtering

- Cache: Mellomlager

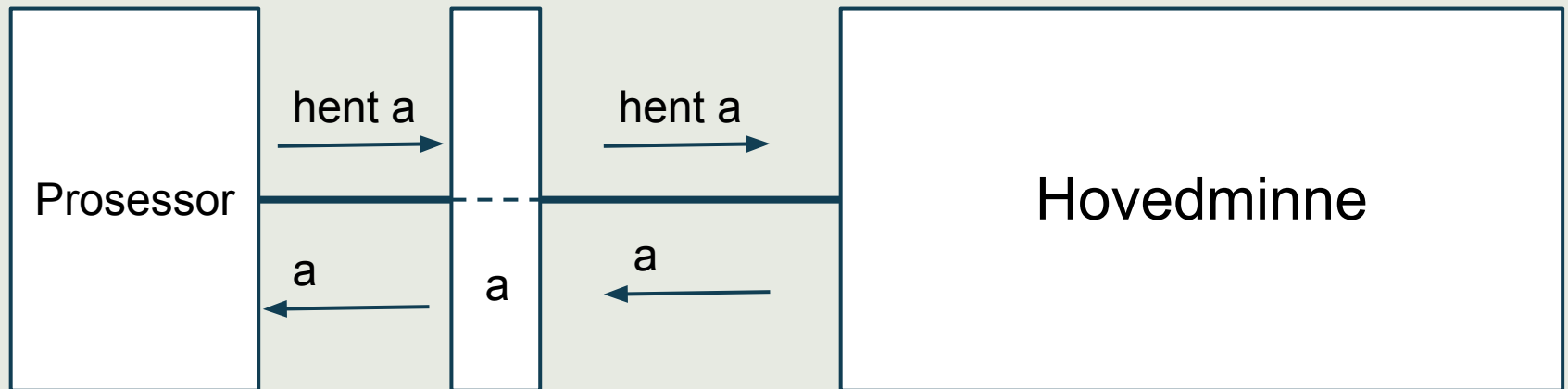


Eksempel: Cachehåndtering

- Cache er til gjengjeld liten
 - Ikke plass til så mye
 - Må bestemme oss for hva vi vil ha i cache
- Jo oftere vi har det vi skal i cache, jo bedre!
- Høy ytelse krever dermed god utkastelsesstrategi
 - Hva skal vi kvitte oss med når cache er full?

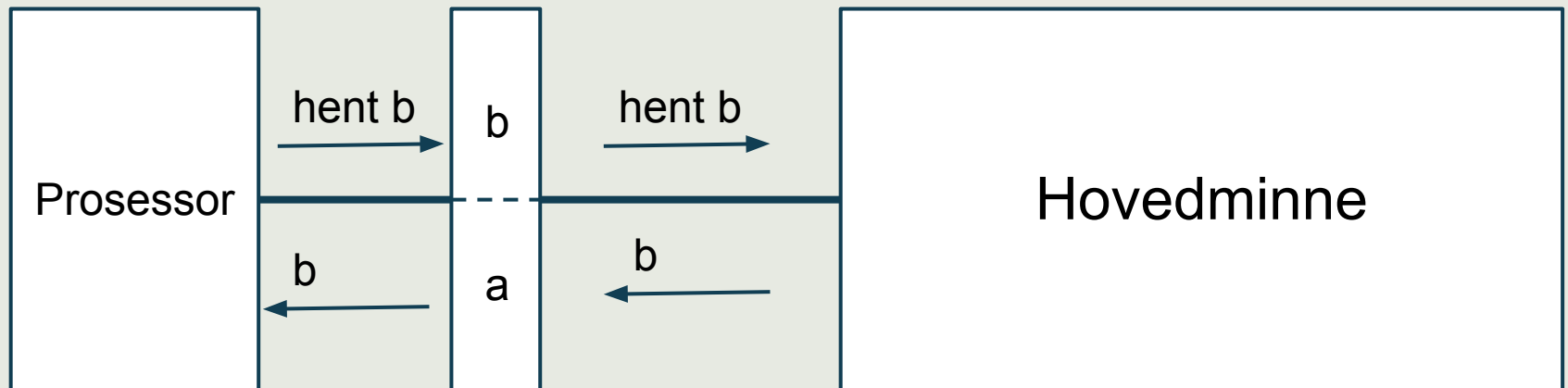
Eksempel: Cachehåndtering

- Antar at vi kjenner fremtidige minneforespørsler
 - Liste $\langle r_1, r_2, \dots, r_n \rangle$ med forespørsler
 - Eksempelvis $\langle a, b, a, d, d, c, b, d, a \rangle$



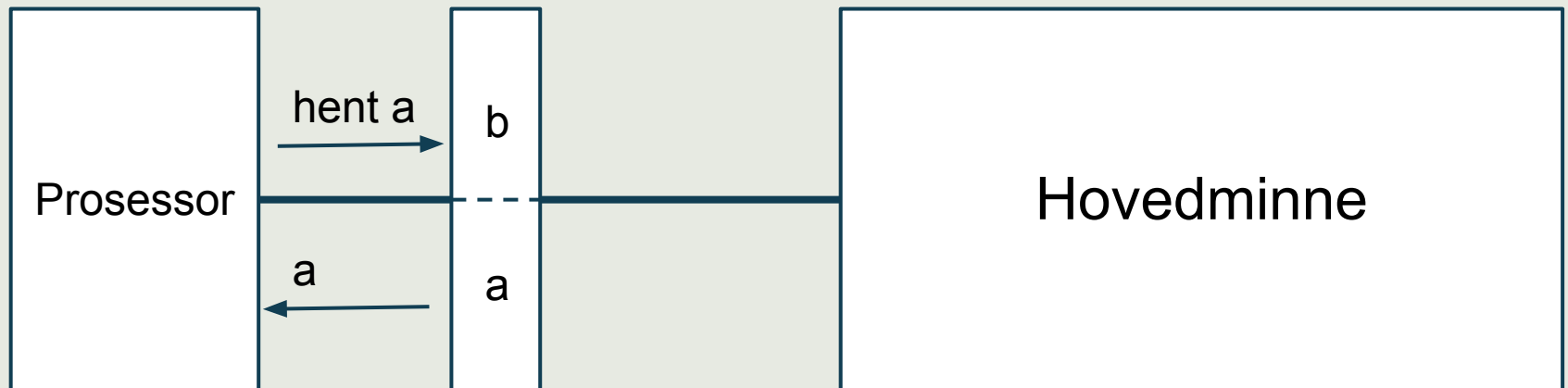
Eksempel: Cachehåndtering

- Antar at vi kjenner fremtidige minneforespørsler
 - Liste $\langle r_1, r_2, \dots, r_n \rangle$ med forespørsler
 - Eksempelvis $\langle a, b, a, d, d, c, b, d, a \rangle$



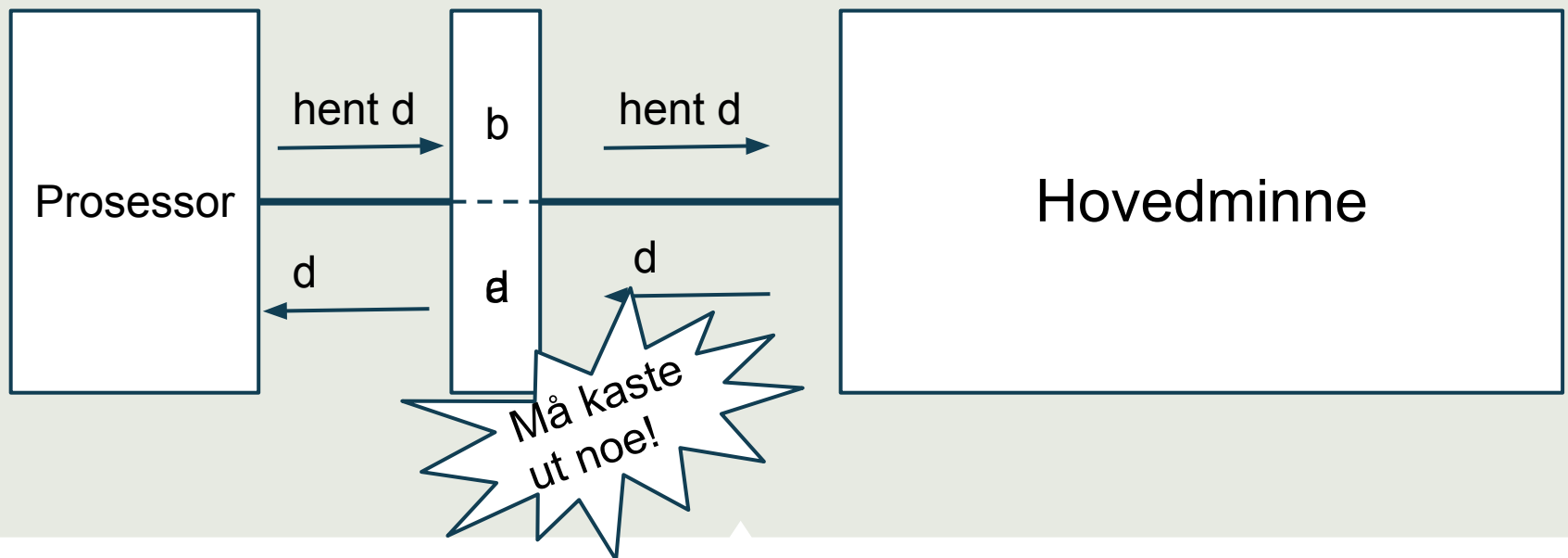
Eksempel: Cachehåndtering

- Antar at vi kjenner fremtidige minneforespørsler
 - Liste $\langle r_1, r_2, \dots, r_n \rangle$ med forespørsler
 - Eksempelvis $\langle a, b, a, d, d, c, b, d, a \rangle$



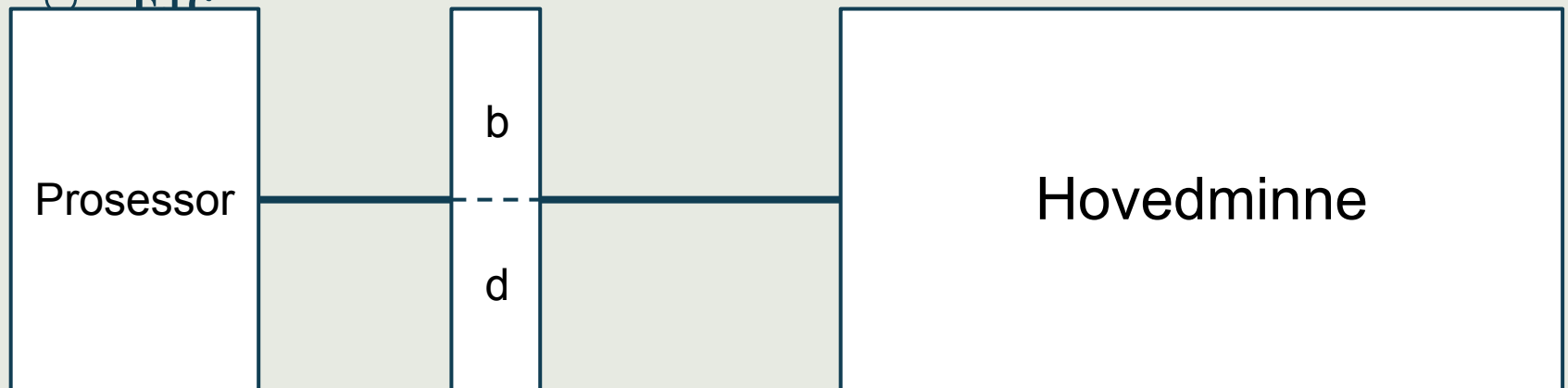
Eksempel: Cachehåndtering

- Antar at vi kjenner fremtidige minneforespørsler
 - Liste $\langle r_1, r_2, \dots, r_n \rangle$ med forespørsler
 - Eksempelvis $\langle a, b, a, d, d, c, b, d, a \rangle$



Eksempel: Cachehåndtering

- Antar at vi kjenner fremtidige minneforespørsler
 - Liste $\langle r_1, r_2, \dots, r_n \rangle$ med forespørsler
 - Eksempelvis $\langle a, b, a, d, d, c, b, d, a \rangle$
 - Etc



Eksempel: Cachehåndtering

- Hva skal vi kaste ut?
- Intuisjon: Virker smart å kaste ut det det er lengst tid til vi skal bruke
 - Aller helst et element vi aldri kommer til å bruke
- Er dette optimalt?

Terminologi

- $\langle r_1, r_2, \dots, r_n \rangle$ - Minneforespørsler
- k - Cachestørrelse
- $m[i]$ - Cachebom etter $\langle r_1, \dots, r_i \rangle$
- $e[i]$ - Element som kastes ut ved r_i
- C_i - Cache ved r_i

Steg 1 - Beskriv valg

- Antar i utgangspunktet tom cache
- For $\langle r_1, \dots, r_k \rangle$ har vi plass i cache
 - Trenger ikke kaste ut noe
 - Intuitivt optimalt å ikke gjøre det heller - kunne bare gjort det når det var nødvendig i stedet. (mer senere)
- For $r_{i > k}$, kan to ting skje
 1. r_i er i cache
 2. r_i er ikke i cache
- Må velge om vi skal kaste ut et element, og i så fall hvilket

Steg 2 - Vis at det grådige valget er trygt

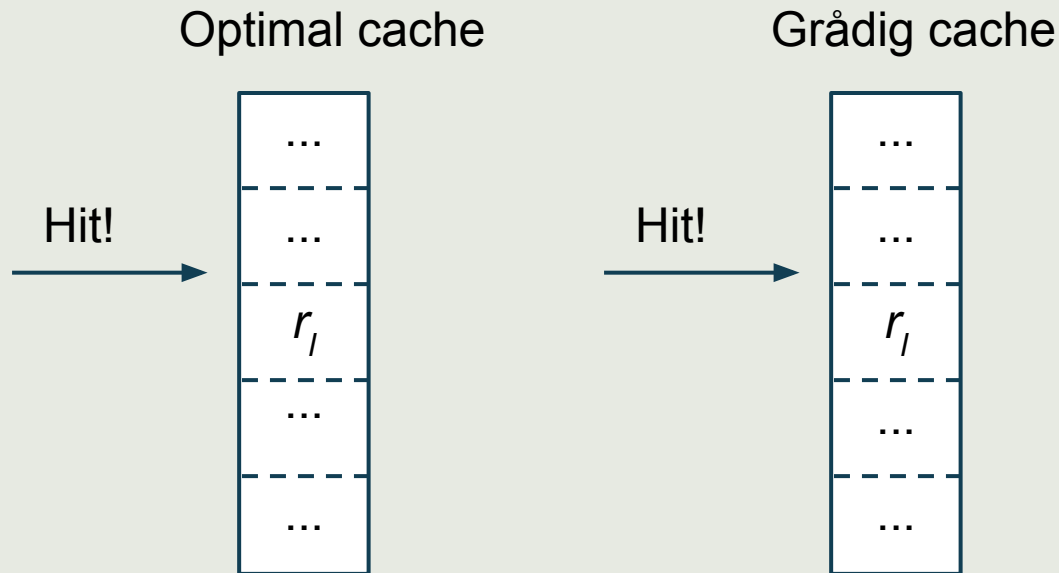
- Det grådige valget er, i de respektive situasjoner:
 1. Ikke kast ut noe fra cache.
 2. Kast ut det elementet det er lengst tid før etterspørres

Bevis for situasjon #1

- Anta at vi har en optimal utkastssekvens hvor et element kastes ut
- Vi kan lage en ny optimal utkastssekvens ved å kaste samme element neste gang denne utkastssekvens fyller cache

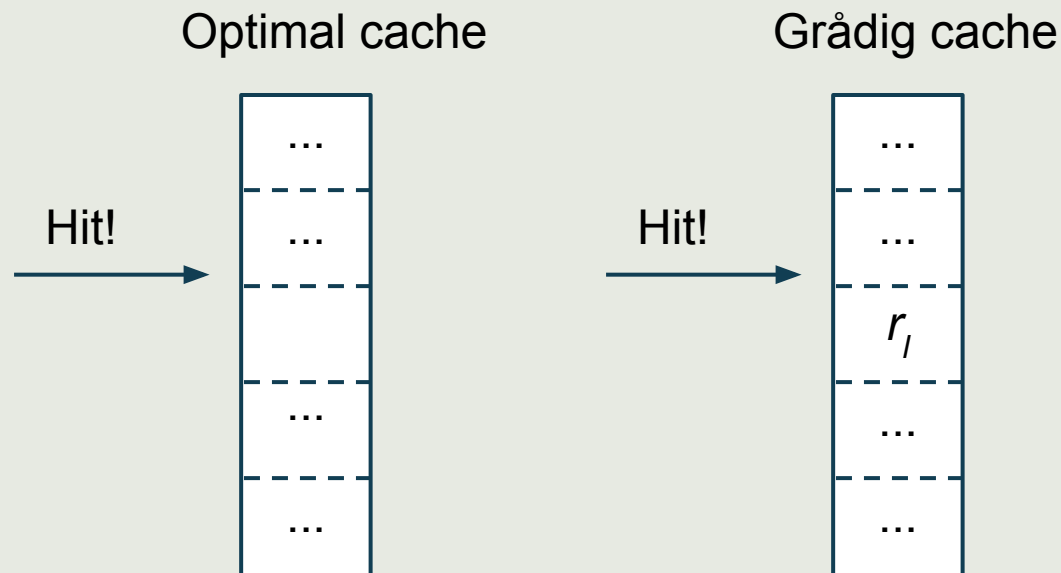
Bevis for situasjon #1

- Anta at vi har en optimal utkastssekvens hvor et element kastes ut
- Vi kan lage en ny optimal utkastssekvens ved å kaste samme element neste gang denne utkastssekvens fyller cache



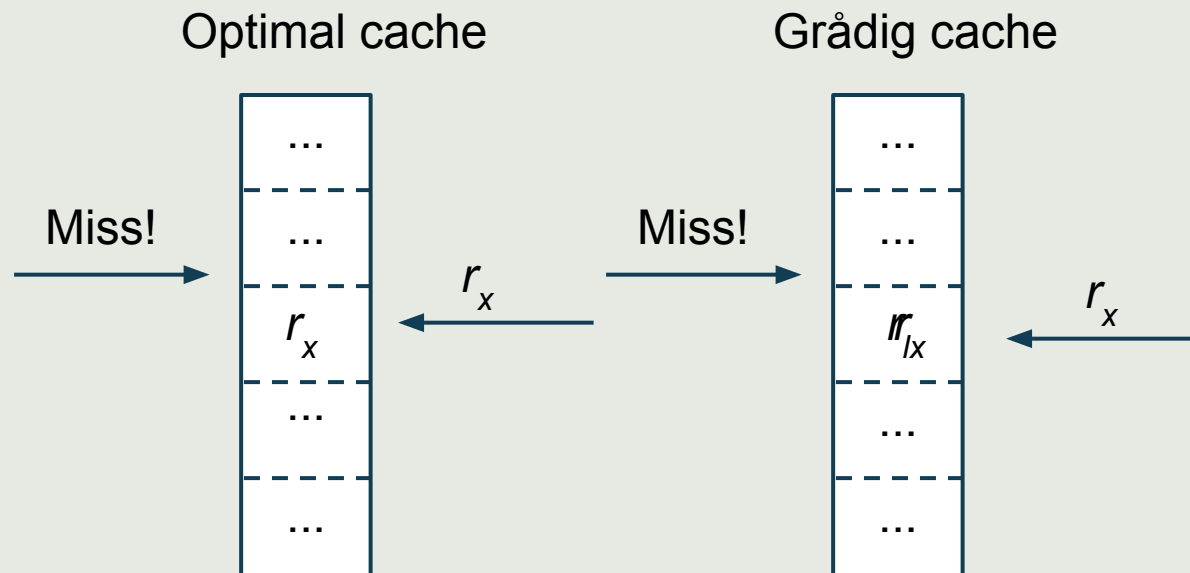
Bevis for situasjon #1

- Anta at vi har en optimal utkastssekvens hvor et element kastes ut
- Vi kan lage en ny optimal utkastssekvens ved å kaste samme element neste gang denne utkastssekvens fyller cache



Bevis for situasjon #1

- Anta at vi har en optimal utkastssekvens hvor et element kastes ut
- Vi kan lage en ny optimal utkastssekvens ved å kaste samme element neste gang denne utkastssekvens fyller cache



Bevis for situasjon #1

- Anta at vi har en optimal utkastssekvens hvor et element kastes ut
- Vi kan lage en ny optimal utkastssekvens ved å kaste samme element neste gang denne utkastssekvens fyller cache
- Ergo vil det alltid finnes en optimal løsning der cache alltid er full for $r_{i > k}$

Bevis for situasjon #2

- Anta at
 - Optimal utkastsekvens kaster ut r_j
 - r_m er elementet det er lengst til etterspørres
 - $r_j \neq r_m$

Etterspørsler



Bevis for situasjon #2

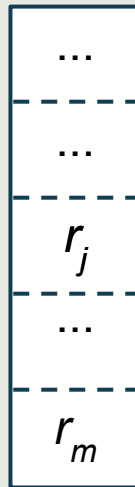
- Anta at
 - Optimal utkastsekvens kaster ut r_j
 - r_m er elementet det er lengst til etterspørres
 - $r_j \neq r_m$
- Vil vise at vi kan kaste ut r_m i stedet

Etterspørsler

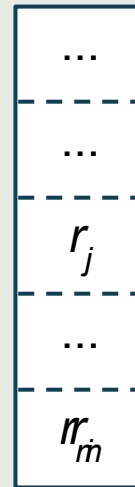


Bevis for situasjon #2

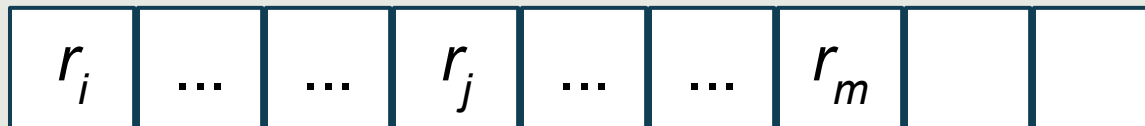
Optimal cache



Grådig cache



Etterspørsler



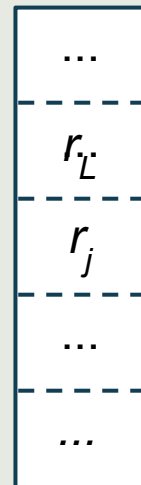
Bevis for situasjon #2

- Frem til r_j , hold cache lik
- Ved r_j :
 - Optimal cache får miss
 - Grådig cache får hit
- Optimal cache må kaste noe
 - r_L

Optimal cache



Grådig cache



Etterspørsler



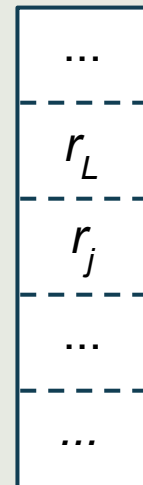
Bevis for situasjon #2

- Optimal cache skilles fra grådig cache kun i $r_m \neq r_L$
- For $\langle r_{j+1}, \dots, r_{m-1} \rangle$:
 - $r = r_L$ - Optimal miss, grådig hit
 - $r \neq r_L$ - Likt

Optimal cache



Grådig cache



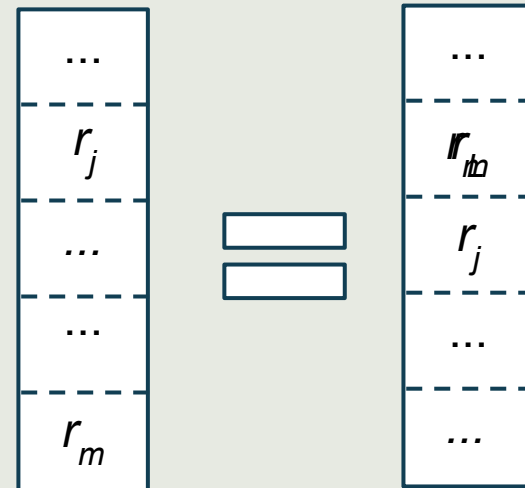
Etterspørsler



Bevis for situasjon #2

- For r_m

- Grådige cache miss - Kast ut r_L
- Optimal cache hit



Etterspørsler



Bevis for situasjon #2

- Herfra og ut holdes cache lik
 - Samme antall cache misses fra r_m til r_n
- Frem til r_j var det også likt antall cache misses
- For $\langle r_j, \dots, r_m \rangle$:
 - r_j - Optimal cache miss
 - r_m - Grådig cache miss
 - Eventuelt ekstra bom for optimal cache i resten

Bevis for situasjon #2

- Cache miss for grådig løsning \leq Cache miss for optimal løsning
- \Rightarrow Det finnes en optimal løsning med det grådige valget
 - Det grådige valget forhindrer ikke optimal løsning

Steg 3: Optimal substruktur

- Optimal løsning består av
 - Valg for $\langle r_{i+1}, \dots, r_n \rangle$
 - Det grådige valget
- Må velge optimalt for $\langle r_{i+1}, \dots, r_n \rangle$
 - Klipp-og-lim bevis

Steg 4: Løsning fra rekursjon

$$m[i] = \begin{cases} m[i-1] & ; r_i \text{ er i cache} \\ m[i-1] + 1 & ; r_i \text{ er ikke i cache} \end{cases}$$

$$e[i] = \begin{cases} \text{NULL} & ; r_i \text{ er i cache} \\ \text{FurthestInFuture}(C_{i-1}) & ; r_i \text{ er ikke i cache} \end{cases}$$

$$C_i = \begin{cases} C_{i-1} & ; r_i \text{ er i cache} \\ C_{i-1} \cup \{r_i\} \\ - \{ \text{FurthestInFuture}(C_{i-1}) \} & ; r_i \text{ er ikke i cache} \end{cases}$$

Øving 9 - Teori

Øving 9 - Praksis