# IN2040: Funksjonell programmering

## *Trær, mengder og huffmankoding*

Martin Steffen

Universitetet i Oslo

5. uke (18. 09. 2023)



# Tema

## Forrige uke

- `lambda`, `let` og lokale variabler
- Dataabstraksjon
- Lister av lister: trær
- Rekursjon på trær

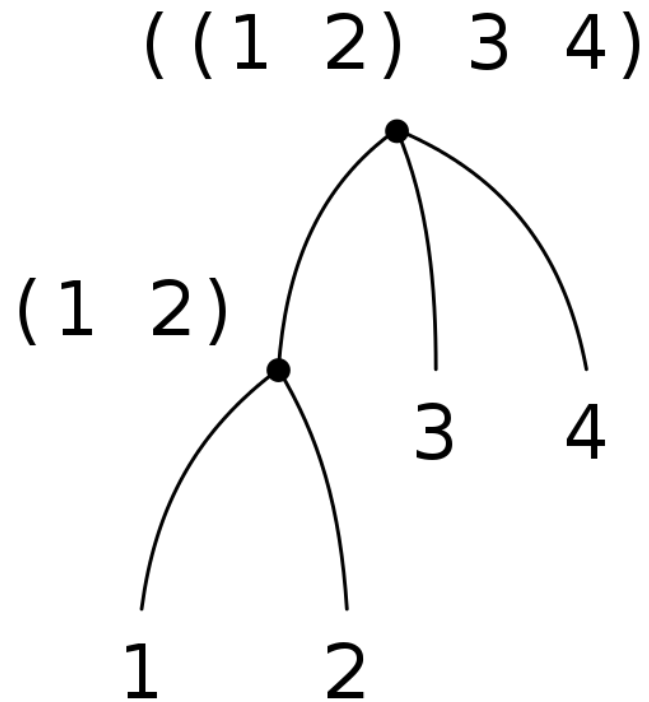
## Denne uka

- Mer om dataabstraksjon og trær
- Eksempler: Mengder og huffmankoding

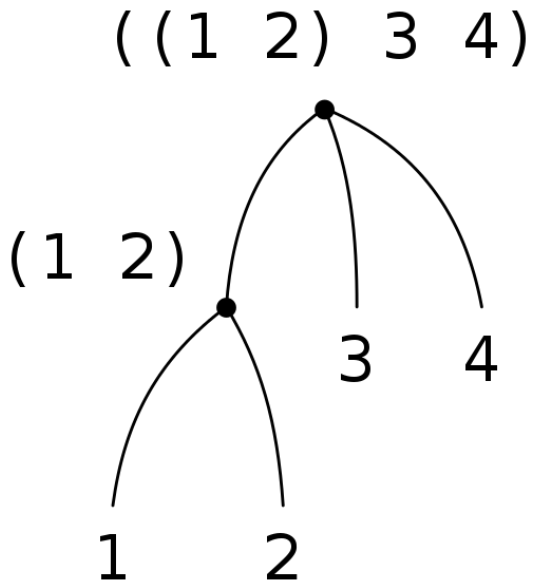


# Lister som trær

- Flate lister: sekvenser.
- Lister av lister: kan sees som trær.
  - Hvert element i en liste er en gren.
  - Elementer som selv er lister er subtrær.
  - Løvnodene i treet er de atomære elementene som ikke er lister.



# Listerekursjon




```
? (count-leaves '((1 2) 3 4)) → 4

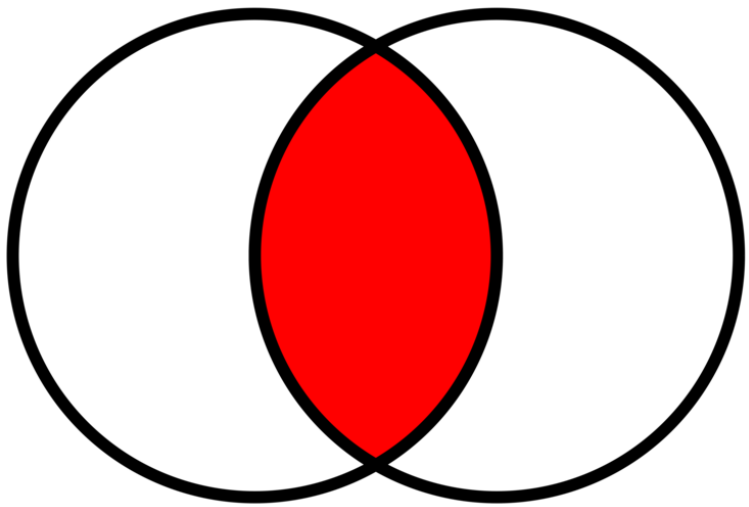
(define (count-leaves tree)
  (cond ((null? tree) 0)
        ((pair? tree)
         (+ (count-leaves (car tree))
            (count-leaves (cdr tree))))
        (else 1)))
```

# Veien videre

- Så langt: Kan behandle nøstede lister som trær,
- men bruker bare `pair?`, `null?`, `cons`, `car` og `cdr` direkte.
- Ingen abstraksjon!
- Vi skal definere abstrakte datatyper for å representere trær.
- Ikke bare *løvnodene*, men også de *interne nodene* kan ha verdier.
- Skal også jobbe med eksempler der rekursjonen velger å kun følge én gren (i stedet for å gå ned i alle subtrær).
- Som praktiske eksempler skal vi se på hvordan både mengder og komprimeringskoder kan implementeres effektivt som trær.

# Først

- Skal se 3 implementasjoner av en abstrakt datatype for mengder (hvorav den tredje er basert på trær).



# Mengder (som matematisk konsept)



- En uordnet samling elementer.
- Uordnet (usortert): $\{1, 2, 3\} = \{2, 3, 1\}$
- Distinkte (unike) elementer: $\{1, 2, 3, 3, 1\} = \{1, 2, 3\}$
- Vi skal se på ulike måter å representere mengder.
- Som dataabstraksjon er eneste krav at vi kan utføre følgende:
- `element-of-set?`
  - spør om et element er medlem av en mengde.
- `adjoin-set`
  - returnerer en mengde med et gitt element lagt til som medlem.
- `intersection-set`
  - tar to mengder og returnerer en mengde der kun elementer som er medlem av dem begge er med (*snitt*).
- `union-set`
  - tar to mengder, returnerer en mengde med alle elementer fra begge.

# Representasjon av mengder #1: Uordnet liste

```
(define (element-of-set? x set)
  (cond ((null? set) #f)
        ((= x (car set)) #t)
        (else (element-of-set? x (cdr set)))))


(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))


(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-set? (car set1) set2)
         (cons (car set1)
               (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set2))))
```

- Mengder som uordnede lister.
- $\{5, 1, 3, 9, 7, 11\}$ = `'(5 1 3 9 7 11)`
- Ingen elementer forekommer mer enn én gang.
- `element-of-set?` er sentral.
- $\{\}$ = `'()`

(men: hva med abstraktion?)

# Kompleksitet

- Hva er kompleksiteten til `element-of-set?` ?
- Predikatet søker sekvensielt gjennom lista helt til
  - vi finner elementet vi ser etter ($\rightarrow$ `#t`),
  - eller det ikke er flere elementer igjen ($\rightarrow$ `#f`).
- I værste fall $O(n)$ gitt en mengde med $n$ elementer.
- Det samme gjelder dermed for `adjoin-set`.
- `intersection-set` involverer sjekk av medlemskap for hvert element i en mengde mot en annen: $O(n^2)$ for to lister med $n$ elementer.
- Det samme vil gjelde `union-set`.

# Representasjon av mengder #2: Ordnet liste



- En enkel måte å gjøre operasjonene mer effektive på er å endre representasjonen til å være ordnet. F.eks:
  - numerisk eller leksikografisk rekkefølge.
  - gi hvert element et numerisk id og sorter på dette.
- Trenger uansett en måte å sammenlikne objekter på.
- For strenger kunne vi brukt `string>?`, `string<?` og `string=?`. (PS: Det finnes også en prosedyre `symbol->string`.)
- Fortsetter vi å anta at elementene er tall kan vi bruke <, > og =.
- $\{5, 1, 3, 9, 7, 11\}$ = `(1 3 5 7 9 11)`

# Sjekk av medlemskap mot ordnet liste

- Ny versjon av `element-of-set?`
- (Antar at mengden er gitt som en sortert liste av tall.)

```
(define (element-of-set? x set)
  (cond ((null? set) #f)
        ((= x (car set)) #t)
        ((< x (car set)) #f)
        (else (element-of-set? x (cdr set)))))
```

- Trenger ikke lenger nødvendigvis sjekke hvert element: kan avbryte hvis vi finner et større element.
- Fortsatt teoretisk $O(n)$ vekst (f.eks. i tilfellet elementet er det siste).
- Men i praksis kan vi regne med at antall trinn i snitt vil være $n/2$.

# Snitt for ordnede lister (1:2)

```
(define (intersection-set set1 set2)
  (if (or (null? set1) (null? set2))
      '()
      (let ((x1 (car set1))
            (x2 (car set2)))
        (cond ((= x1 x2) (cons x1
                               (intersection-set (cdr set1)
                                                 (cdr set2))))
              ((< x1 x2) (intersection-set (cdr set1) set2))
              ((< x2 x1) (intersection-set set1 (cdr set2)))))))
```

- Enda mer å spare på implementasjonen av snitt:
- Sammenlikner første element i hver liste:
  - Hvis de er like inkluderer vi det, og ser videre på `cdr` av listene.
  - Hvis det ene er mindre enn det andre vet vi at det ikke kan være medlem og vi ser på `cdr` av lista med det minste elementet.

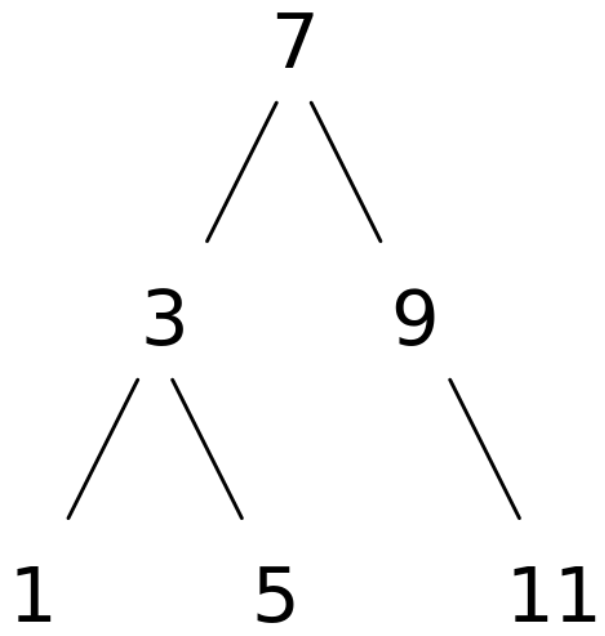
# Snitt for ordnede lister (2:2)

```
(define (intersection-set set1 set2)
  (if (or (null? set1) (null? set2))
      '()
      (let ((x1 (car set1))
            (x2 (car set2)))
        (cond ((= x1 x2) (cons x1
                               (intersection-set (cdr set1)
                                                 (cdr set2))))
              ((< x1 x2) (intersection-set (cdr set1) set2))
              ((< x2 x1) (intersection-set set1 (cdr set2)))))))
```

- I hvert trinn reduseres problemet til snittet av mindre mengder.
- Utnytter den ordnede representasjonen direkte, kaller ikke `element-of-set?`.
- Antallet trinn er i værste fall proporsjonalt med summen av lengdene på listene (i stedet for produktet)
- O($n$) vekst i stedet for O($n^2$).

# Representasjon av mengder #3: Binærtrær

- Enda mer effektivt: mengde som binærtre med ordnede noder.
- AKA: binære søketrær
- Hver node lagrer
  - et element
  - en *venstregren* med *mindre* elementer
  - en *høyregren* med *større* elementer
- Ved søk etter et gitt element x:
  - Hvis x er mindre enn nodeverdien, søk til venstre.
  - Hvis x er større, søk til høyre.
- Kan halvere søkerommet i hvert trinn:
- Logaritmisk vekst: $\mathrm{O}(\log n)$.



# Abstraksjonsbarriere for binærtrær

- Vi representerer mengder som trær, og trær som lister.
- Har allerede abstraksjonsbarriere for mengder, men mangler for trær.
- Konstruktor(er) og selektorer:

```
(define (make-tree entry left right)
  (list entry left right))

(define (entry tree) (car tree))

(define (left-branch tree) (cadr tree))

(define (right-branch tree) (caddr tree))
```

# Mengdeoperasjoner for binærtrær

```
(define (element-of-set? x set)                    ;; log(n) vekst
  (cond ((null? set) #f)
        ((= x (entry set)) #t)
        ((< x (entry set))
         (element-of-set? x (left-branch set)))
        ((> x (entry set))
         (element-of-set? x (right-branch set)))))


(define (adjoin-set x set)                         ;; samme strategi
  (cond ((null? set) (make-tree x '() '()))
        ((= x (entry set)) set)
        ((< x (entry set))
         (make-tree (entry set)
                    (adjoin-set x (left-branch set))
                    (right-branch set)))
        ((> x (entry set))
         (make-tree (entry set)
                    (left-branch set)
                    (adjoin-set x (right-branch set))))))
```

# Abstraksjonsbarrierer: lag på lag

`en-prosedyre-som-bruker-mengder`

---

`element-of-set?`, `adjoin-set`, `intersection-set`, `union-set`

---

`make-tree`, `entry`, `left-branch`, `right-branch`

---

`'()`, `cons`, `car`, `cdr`, `null?`, `pair?`

---

`lambda`, eller en annen par-implementasjon?

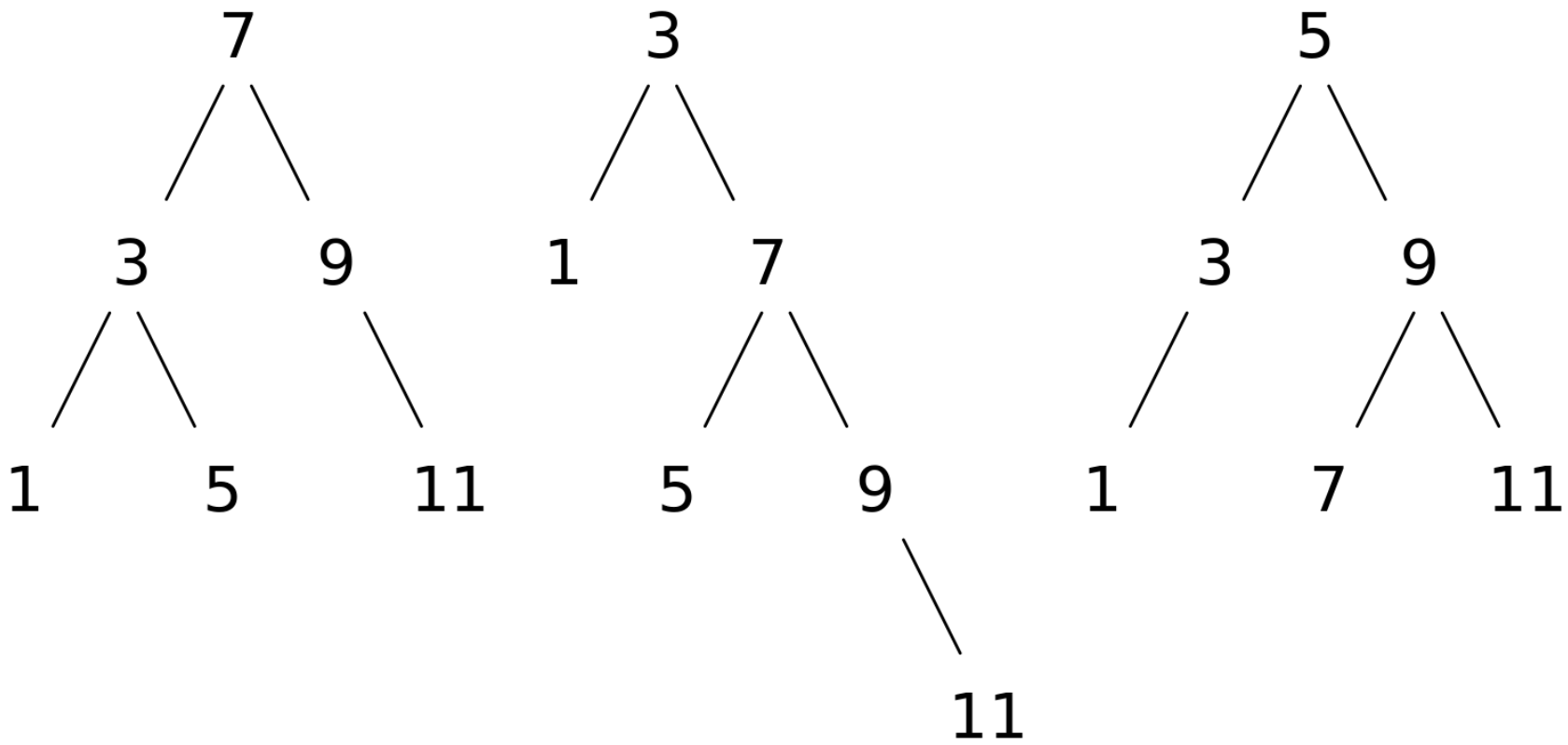
# Abstraksjonsbarrieren for trær lekker litt

```
(define (adjoin-set x set)
  (cond ((null? set) (make-tree x '() '()))
        ((= x (entry set)) set)
        ((< x (entry set))
         (make-tree (entry set)
                    (adjoin-set x (left-branch set))
                    (right-branch set)))
        ((> x (entry set))
         (make-tree (entry set)
                    (left-branch set)
                    (adjoin-set x (right-branch set))))))
```

- Koden slik den er gitt i SICP har noen lekkasjer i abstraksjonsbarrieren:
- Skjuler ikke alle designvalgene våre for binærtrær og for mengder.
- Burde hatt et predikat `empty-tree?` (og `empty-set?`)
- Og kanskje en konstant `empty-tree` (og `empty-set`)
- Eller `(make-leaf x)` i stedet for `(make-tree x '() '())`.

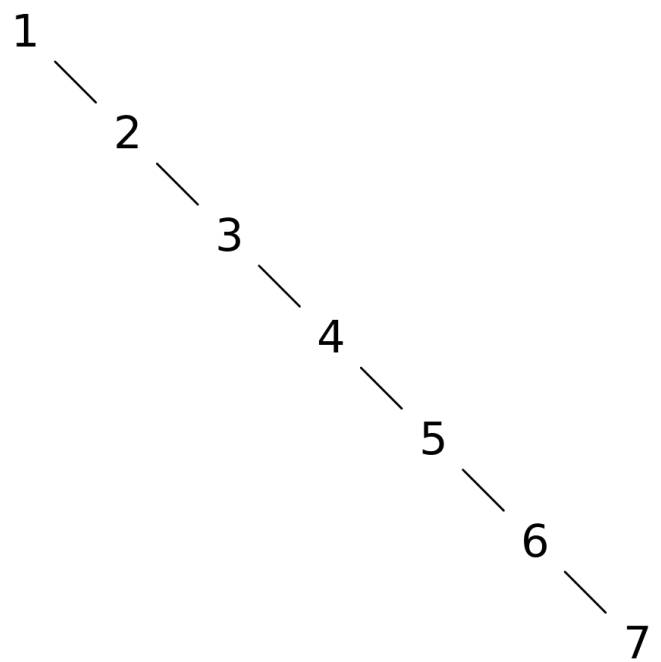
# Flere trær for samme mengde

- Kan finnes flere mulige trær for én og samme mengde.
- Eksempel på noen mulige trær for $\{5, 1, 3, 9, 7, 11\}$:



# Balanserte trær

- `adjoin-set` bruker samme strategi som `element-of-set?` og får $O(\log n)$ vekst.
- Forutsetter at treet er relativt balansert.
- Eksempel på tre laget med `adjoin-set` anvendt i sekvens på 1 til 7.
- Ingen besparelser i forhold til en ordnet liste.
- Med "tilfeldig" distribuerte elementer kan vi forvente at trærne i snitt er balanserte.
- Men vi har ingen garanti.
- Kan løses ved å ha en prosedyre som sørger for å balansere treet jevnlig (eller bruke mer sofistikerte trestrukturer).



# Hvor vi er på vei: Huffman-koding



- Ved komprimering er målet å kode informasjon med færrest mulig bits (og færre enn den opprinnelige representasjonen).
- Huffman-koding: En algoritme for å generere en optimal komprimeringskode automatisk fra data.
- Oppdaget i 1951.
- Brukes for å komprimere forskjellige typer data som lyd (f.eks. *mp3*), bilder (*jpeg*), video (*mpeg-1*), og tekst (*gzip* / *bzip2*).
- Men aller først trenger vi litt terminologi...

# Litt kompresjons- og kryptografi-sjargong



- Koding for kompresjon vs kryptering. Forskjellig mål: effektiv lagring/overføring vs. effektiv skjuling av informasjon.
- Vi har et alfabet av symboler.
  - F.eks. $\{\texttt{E}, \texttt{T}, \texttt{A}, \texttt{Z}\}$
- Vi har også et kodealfabet.
  - F.eks. $\{0, 1\}$ om vi antar binærkode
- En kodebok tilskriver et kodeord til hvert symbol.
  - F.eks. $\texttt{E} = 00$
- Vi ønsker å kunne gjøre to ting:
- å kode / komprimere data (ofte kalt melding) til en kompakt kode.
- å dekode / dekomprimere en kode tilbake til lesbare data.
- Tapsfri (*lossless*): Opprinnelig melding kan rekonstrueres eksakt.
- Ikke-tapsfri (*lossy*): Opprinnelig melding kan ikke rekonstrueres eksakt.

# Koder med fast lengde

- Eksempel på en binærkode: ASCII
  - Sekvens av 7 bits for hvert tegn.
  - Gir mulighet for å representere $2^7 = 128$ ulike tegn.
- Eksempel på såkalt *fixed-length code*
- For å skille mellom $n$ symboler trenger vi $\log_2 n$ bits per symbol.
  - Enda mer generelt: må bruke $\log_b n$ kodeposisjoner gitt et kodealfabet av størrelse $b$ (hvor $b = 2$ for bits / binærkode).
- Gitt alfabetet {E, T, A, Z } trenger vi $\log_2 4 = 2$ bits.
- En mulig kodebok:

| **symbol** | E | T | A | Z |
|---|---|---|---|---|
| **kode** | 00 | 01 | 10 | 11 |

- 0001 = ET, 1011 = AZ
- Svakheter?

# Frekvens

IN2110 – Methods in Language Technology
Course content:
This course gives an in-depth study in basic methods and practical tools for basal language technology (methods for automatic analyzation of language-based data). It covers both rule-based techniques, such as phrase structure grammar, and approximations with a starting point in machine learning, such as vector space semantics and classification. The course will take a look at some applications of methods for issues within language technology such as tagging, parsing and text classification (such as sentiment analysis). The course has a strong practical component, with use of relevant tools and projects with written rapports, among other things, which is needed to qualify for the exam.

- 'e': 51.
- 'z': 1.

# Med fast lengde ignoreres frekvens

- Svakhet: koden over tar ikke hensyn til symbolenes relative frekvens.
- Koden er kun optimal dersom alle symbolene er like sannsynlige.
- Vi kan få en mer kompakt kode dersom symboler som forekommer ofte gis en kortere representasjon.
- På Engelsk er f.eks. bokstaven 'E' mye vanligere enn 'Z'.
- Gjenspeiles f.eks. i morsekode:
  - E = •
  - Z = − − • •
- Morsekode er eksempel på såkalt *variable-length code* (også ikke en binærkode[1])

[1] Faktisk 5, ikke 2 "symboler": •, -, og 3 ulike pauser.

# Kode med variabel lengde

- La oss lage en ny kodebok på bakgrunn av frekvensen for hvert symbol (som vi tenker oss at vi har observert fra data):

| **symbol** | E | T | A | Z |
|---|---|---|---|---|
| **frekvens** | 55 | 30 | 10 | 5 |
| **kode** | 0 | 1 | 00 | 01 |

- 01 = ET eller Z?
- 0001 = EEET, AZ, EAT, AET, eller EEZ?
- Kodeordene har variabel lengde, men er ikke unikt dekodbare.
- Mulig løsning: bruk en dedikert utvetydig kode som separator.
- F.eks. kort pause i morsekode.
- Bedre: sørg for at ingen kode er prefiks for en annen kode!

# Prefikskode

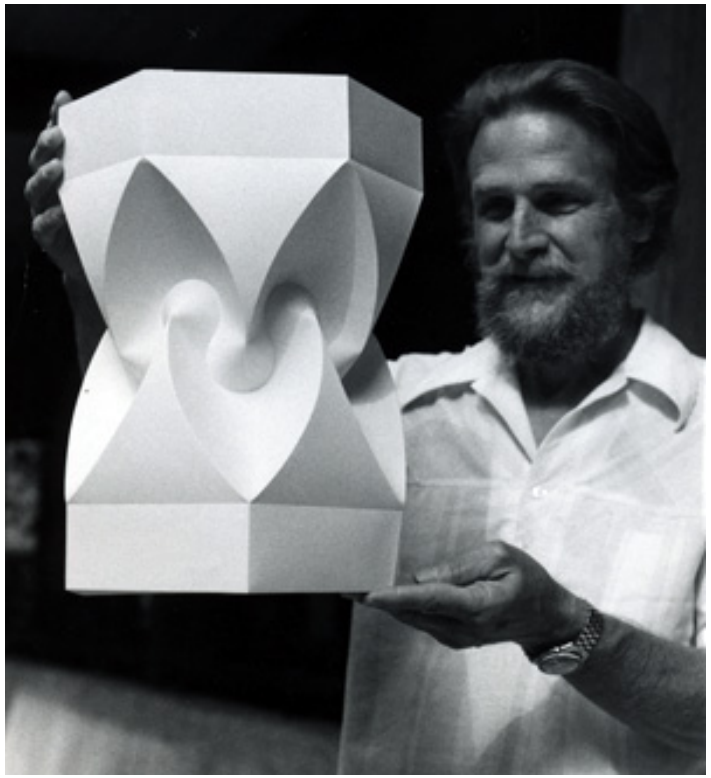
- Nytt forslag til kodebok:

| **symbol** | E | T | A | Z |
|---|---|---|---|---|
| **frekvens** | 55 | 30 | 10 | 5 |
| **kode** | 0 | 10 | 110 | 111 |

- Her har vi en prefikskode med variabel lengde som er unikt dekodbar.
- Kodeordet som representerer et gitt symbol er aldri et prefiks for kodeordet for et annet symbol.
- 010 = ET (utvetydig)
- Oppnår kompresjon kun dersom symbolene er ikke-uniformt distribuert.
  - Forekommer alle like ofte kunne vi like gjerne brukt fast lengde.

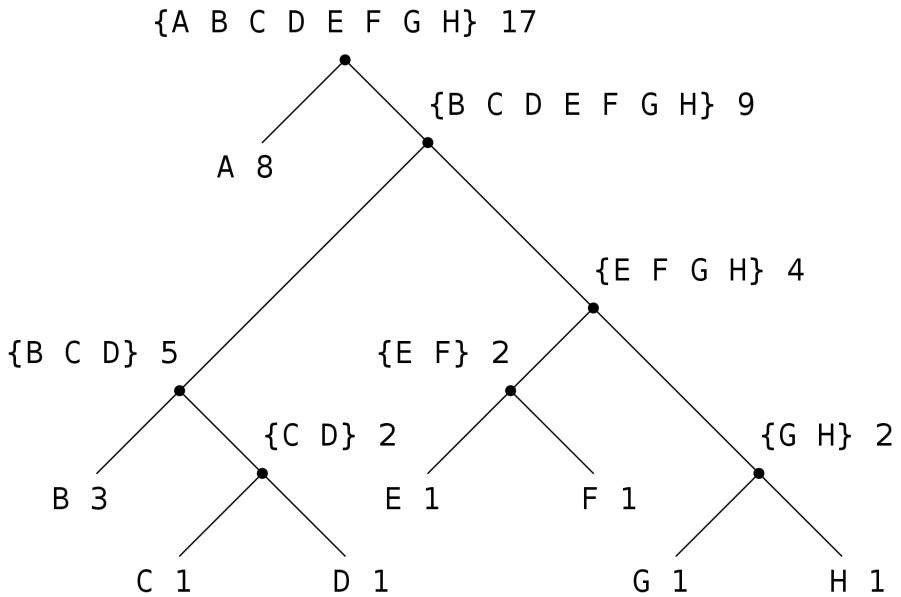
# David A. Huffman (1925–99)

- Huffman var en pioner i informasjonsteori og matematisk origami (!)
- Semesteroppgave som student i 1951:
- Definerte en algoritme for å automatisk generere optimale prefikskoder fra data.
- Basert på at vi kjenner de relative frekvensene til symbolene.
- Såkalt Huffman-koding.
- Kodeboken representeres ved et Huffman-tre.



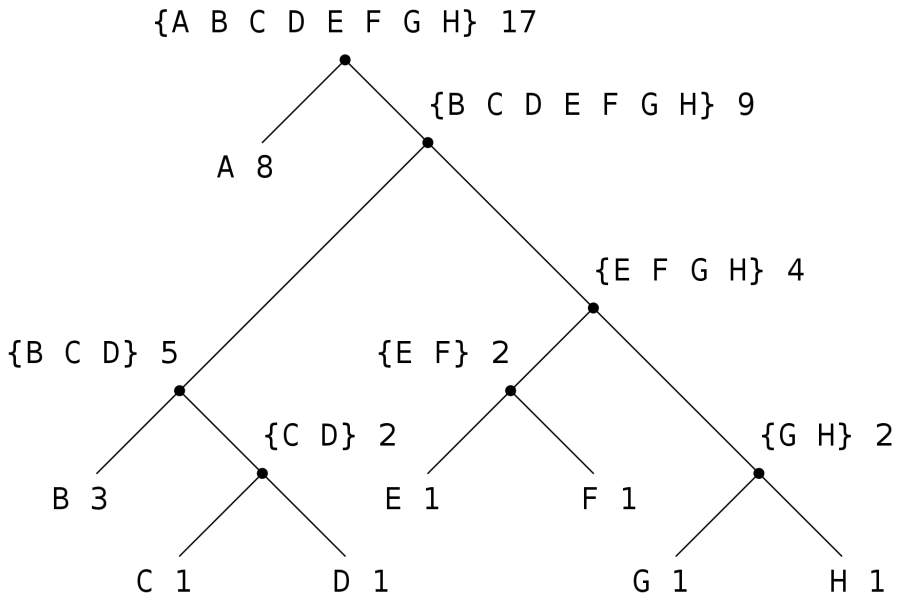
# Huffman-trær: kodeboken

- Kodeboken gitt ved et binærtre:
  - en venstregren representerer 0,
  - en høyregren representerer 1.
- Løvnoder, lagrer:
  - et symbol,
  - vekt (frekvens).
- Interne noder, lagrer:
  - mengden av symbolene i løvnodene under det i treet,
  - summen av vektene,
  - to barn.
- Gitt et Huffman-tre kan vi
  - kode en gitt datasekvens,
  - dekode en gitt bitsekvens.



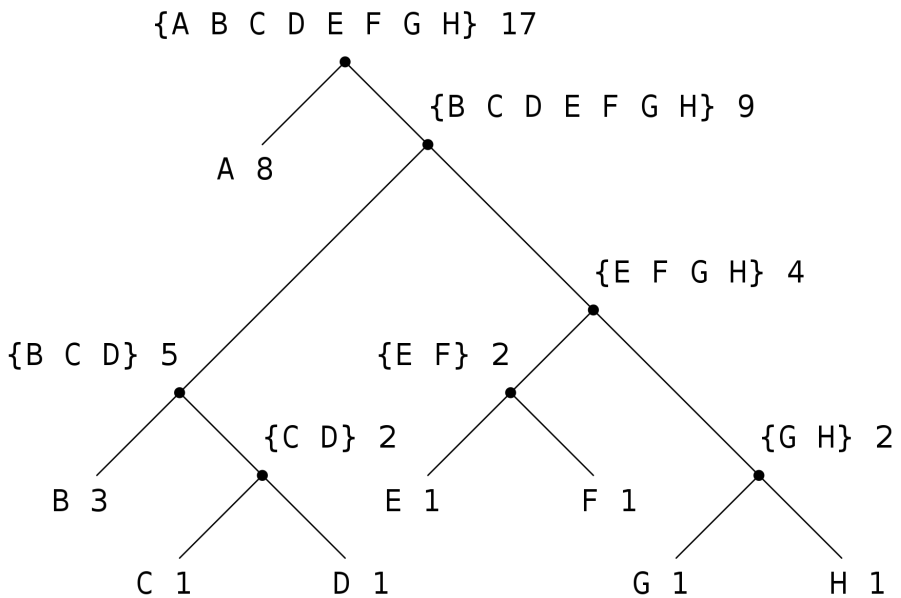
# Huffman-trær: koding

- For å kode en melding:
- For hvert symbol:
- Vi begynner ved roten og klatrer nedover treet til løvnoden som representerer symbolet.
- Velger gren ved å sjekke hvilken node symbolet tilhører.
- Legger til 0 for hver venstregren.
- Legger til 1 for hver høyregren.
- F.eks. `D` = 1011, `DB` = 1011100



# Huffman-trær: dekoding

- For å dekode en bitsekvens:
- Begyn ved roten og les av bits:
- Ta til venstre for hver 0.
- Ta til høyre for hver 1.
- Ved en løvnode: les av symbolet og begynn fra roten igjen.
- F.eks. `10001010` = `BAC`



# Å generere et Huffman-tre

- Gitt et alfabet og de relative frekevensene til symbolene, hvordan kan vi generere kodeboken som vil kode meldinger med færrest mulig bits?

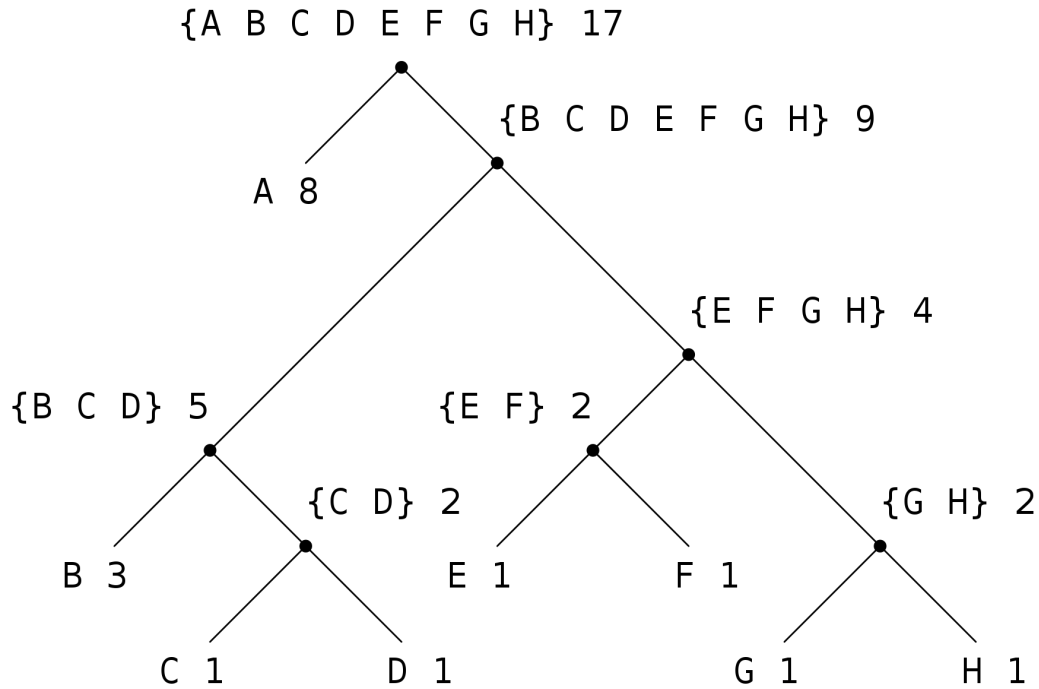
1: Start med en liste av løvnodene.

2: Slå sammen de to nodene med lavest frekvens til en ny node (med summen av frekvensene og unionen av symbolene).
   - La den ene noden tilsvare venstregrenen og den andre høyre.

3: Oppdater lista av noder og gjenta fra 2.
   - Legg til den nye noden og fjern nodene vi slo sammen.

4: Stopp når vi står igjen med kun én node (roten).

# Å generere et Huffman-tre (forts.)




| | |
|---|---|
| Start: | {(A 8) (B 3) (C 1) (D 1) (E 1) (F 1) (G 1) (H 1)} |
| Sammenslåing: | {(A 8) (B 3) ({C D} 2) (E 1) (F 1) (G 1) (H 1)} |
| Sammenslåing: | {(A 8) (B 3) ({C D} 2) ({E F} 2) (G 1) (H 1)} |
| Sammenslåing: | {(A 8) (B 3) ({C D} 2) ({E F} 2) ({G H} 2)} |
| Sammenslåing: | {(A 8) (B 3) ({C D} 2) ({E F G H} 4)} |
| Sammenslåing: | {(A 8) ({B C D} 5) ({E F G H} 4)} |
| Sammenslåing: | {(A 8) ({B C D E F G H} 9)} |
| Stopp: | {({A B C D E F G H} 17)} |

# Noen kommentarer

- Definerer ikke alltid et unikt tre
  - Avhenger av hvordan vi velger hvilken node som blir venstre-/høyregren,
  - og hvilke noder som slås sammen når flere enn to har samme verdi.
  - Det kan altså finnes flere optimale koder.
- Effektivisering
  - Hold nodelista frekvenssortert: Kan da alltid slå sammen de to første.

# Enda flere kommentarer

- En Huffman-kode er optimal i den forstand at den gir minst gjennomsnittlig kodestørrelse.
- Premisser:
  - Kodeboken må også overføres eller allerede være kjent.
  - Faktiske symbolfrekvenser i meldingen vi skal kode stemmer overens med frekvensene som ble brukt for å generere kodeboka.
  - Vi koder ett symbol av gangen.



- Det siste punktet kan i praksis omgås ved at et separat trinn først definerer symbolene i alfabetet.
  - Symbolene kan f.eks. være lengre sekvenser av tegn eller ord.
  - Bruk av to trinn er også grunnen til at Huffman-koding, som gir en tapsfri kode, kan brukes i ikke-tapsfrie formater:
    - F.eks. ved *kvantisering* mappes flere opprinnelige symboler til et mindre sett av symboler (*ikke-tapsfritt*), som så Huffman-kodes (*tapsfritt*).

# Huffman-trær: Abstraksjonsbarriere (1:2)

- Løvnoder: representert av en liste med tre elementer:
  - `'leaf` (typetagg)
  - symbol
  - vekt (frekvens)
- Konstruktor:
  - `make-leaf`
- Predikat
  - `leaf?`
- Selektorer:
  - `symbol-leaf`
  - `weight-leaf`

```
(define (make-leaf symbol weight)
  (list 'leaf symbol weight))

(define (leaf? object)
  (eq? (car object) 'leaf))

(define (symbol-leaf x) (cadr x))

(define (weight-leaf x) (caddr x))
```

# Huffman-trær: Abstraksjonsbarriere (2:2)



- Interne noder: liste med fire elementer:
  - venstregren
  - høyregren
  - liste av symboler
  - vekt
- Konstruktor:
  - `make-code-tree`
- Selektorer:
  - `left-branch`
  - `right-branch`
  - `symbols`
  - `weight`

```
(define (make-code-tree left right)
  (list left
        right
        (append (symbols left)
                (symbols right))
        (+ (weight left) (weight right))))

(define (left-branch tree) (car tree))

(define (right-branch tree) (cadr tree))

(define (symbols tree)
  (if (leaf? tree)
      (list (symbol-leaf tree))
      (caddr tree)))

(define (weight tree)
  (if (leaf? tree)
      (weight-leaf tree)
      (cadddr tree)))
```

# Dekoding

```
(define (decode bits tree)
  (define (decode-1 bits current-branch)
    (if (null? bits)
        '()
        (let ((next-branch
               (choose-branch (car bits) current-branch)))
          (if (leaf? next-branch)
              (cons (symbol-leaf next-branch)
                    (decode-1 (cdr bits) tree))
              (decode-1 (cdr bits) next-branch)))))
  (decode-1 bits tree))

(define (choose-branch bit branch)
  (cond ((= bit 0) (left-branch branch))
        ((= bit 1) (right-branch branch))))
```

- Hvorfor bruker vi `decode-1`? Kalles jo med de samme argumentene, hvorfor ikke bare kalle `decode`?
- Kan vi skrive en halerekursiv versjon av `decode`?

# Andre ting som gjenstår

- Flere prosedyrer dere kommer til å skrive i neste innlevering:
- `encode`
  - **Input**: et Huffman-tre og en melding (liste av symboler).
  - **Output**: en bitsekvens (liste av `0` og `1`).
- `grow-huffman-tree`
  - **Input**: en liste av symboler og frekvenser.
  - **Output**: et Huffman-tre.



# Parprogrammering

- Husk at oblig 2 og 3 (a+b) anbefales løst i grupper av to eller tre studenter sammen.
- Ok om du vil levere individuelt.
- Trenger du samarbeidspartner? Spør på gruppetimen eller forumet (GitHub eller astro-discourse).
- Ok å bytte grupper mellom oblig 2 og 3, men ikke mellom *a* og *b*.
- Bruk gruppefunksjonen i Devilry ved levering.





# Neste uke

- Vi bryter med ren funksjonell programmering.
- Vi skal snakke om:
- Tilstand
- Destruktive operasjoner
- Omgivelsesmodellen

