W8.L1

Uke 8,
Forelesning 1

HUSK – *Hittil…*

- Forutsetninger for og essensen i faget
- Metodekall, rekursjon, permutasjoner
- Analyse av algoritmer
- Introduksjon til ADT'er
- De første ADT'er: Lister, stabler og køer

Uke 1, Uke 2 og Uke 3

- Flere ADT'er: Generelle trær, binære trær og binære søketrær

Uke 4 og Uke 5

- Flere ADT'er: Hashing, hash-tabeller
- ADT'er for disk-datastrukturer introduseres: Utvidbar hashing, B-Trær

Uke 6

- Prioritetskøer & Heap-implemantasjonen

Uke 7

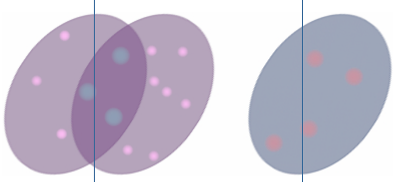
OVERSIKT – *Uke 8, Forelesning 1 (W8.L1)*

Vi introduserer ADT'en for disjunkte mengder.

Spesifikt skal vi se på:

- Relasjoner aRb
- Ekvivalens relasjoner,
  ekvivalens klasser,
  det dynamiske ekvivalensproblemet
- Operasjoner
- Implementasjon

TEMA: DISJUNKTE MENGDER



Introduksjon til Disjunkte Mengder

Disjunkt-mengde ADT'en (MAW kap. 8.1–8.5)

- Løser ekvivalensproblemet
- Lett og rask implementasjon
- Vanskelig tidsforbrukanalyse

- En **relasjon** *R* er definert på en mengde *S* ved at *aRb* enten er sann eller usann for hvert par (*a*, *b*) av elementer i *S*.
- Hvis *aRb* er sann, sier vi at *a* **er relatert til** *b*, eller at *a* **står i forhold til** *b* **via** *R*.

- **Eksempel:** $\leq$-relasjonen på mengden av heltall.

- $3 \leq 5$ er sann. Dermed er 3 relatert til 5 med hensyn på $\leq$-relasjonen. Derimot er $6 \leq 5$ usann, som betyr at 6 er ikke $\leq$-relatert til 5.
- NB! Standard matematiske definisjon av relasjoner er slik: En relasjon *R* er en delmengde av $S \times S$.

  Vi sier at *aRb* holder hvi<u>ss</u> – dvs. "hvis og bare hvis" $(a, b) \in R$.

- En **ekvivalensrelasjon** på en mengde S er en relasjon med følgende egenskaper:

- **Refleksivitet:**
  $a \sim a$ for alle elementer $a \in S$
- **Symmetri:**
  Hvis $a \sim b$, så er $b \sim a$
- **Transitivitet:**
  Dersom $a \sim b$ og $b \sim c$, så er også $a \sim c$

- **Eksempel:**
  En velkjent ekvivalensrelasjon er '='–relasjonen på tall.

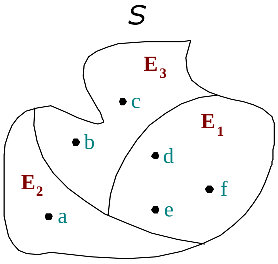
Et annet eksempel er **bilveirelasjonen**:

- *La S være mengden av tettsteder i Norge, og la a og b være to tilfeldige tettsteder. Da er a ~ b hviss (hvis og bare hvis) det finnes en måte å komme fra a til b ved å kjøre bil uten å bruke ferge.*
- Under forutsetning av at ingen veier er enveiskjørte, er dette en ekvivalensrelasjon.

Et tredje eksempel:

- Relasjonen "**går i samme klasse som**" definert over elevene ved en skole er en ekvivalensrelasjon.

- En ekvivalensrelasjon "~" på en mengde S deler elementene i *S* inn i **ekvivalensklasser** slik at alle elementene i en ekvivalensklasse E_i er relaterte til hverandre, men ikke med noen elementer i andre ekvivalensklasser i *S*.
- Vi sier at ekvivalensklassene utgjør **disjunkte** delmengder av *S*.



Eksempel:

- Ekvivalensklasser som kan tenkes indusert av bilveirelasjonen definert over mengden av tettsteder i Norge:
- *E*1 = {Longyearbyen}
- *E*2 = {Svolvær, Kabelvåg, Stamsund, Leknes, Reine, Sørvågen}
- *E*3 = {alle tettstedene på fastlandet}

DISJUNKTE MENGDER – *Det dynamiske ekvivalensproblemet #1*

Det dynamiske ekvivalensproblemet

- **Ekvivalensproblemet** består i å avgjøre om to elementer *a* og *b* i *S* står i relasjon til hverandre, dvs. om $a \sim b$ for en gitt ekvivalensrelasjon .
- Hvis alle relasjonene mellom elementene er gitt eksplisitt ved en 2-dimensjonal boolsk matrise, behøver vi bare et enkelt oppslag i matrisen for å avgjøre om $a \sim b$.

	a	b	c	d	e
a	1	0	1	1	1
b	0	1	0	1	0
c	1	0	1	1	0
d	1	1	1	1	1
e	1	0	0	1	1

- Problemet er at vi ikke alltid får eksplisitt oppgitt alle verdiene i matrisen, dvs. hvilke elementer som er relatert til hverandre.

DISJUNKTE MENGDER – *Det dynamiske ekvivalensproblemet #2*

- Grunnen til det er at antall relasjonsforhold vokser kvadratisk (n^2) i forhold til antall elementer i mengden.
- I stedet er det vanlig å få oppgitt noen relasjonsforhold, f.eks. at $a \sim b$, $c \sim d$, $e \sim a$ og $d \sim b$. Så må vi raskt kunne svare på om det fra de tre ekvivalensegenskapene følger at $a \sim d$ ("deduktivt").
- Eksempel: labyrint!
- **Viktig observasjon:** For å bestemme om $a \sim d$, er det nok å sjekke om de er i samme ekvivalensklasse!

Disjunkt sett ADT tilbyr to operasjoner:

- **Find(*a*)** returnerer en representant for ekvivalensklassen til element *a*

  Representanten er alltid den samme, uavhengig av hvilken *a* i ekvivalensklassen man angir.

- **Union(*a, b*)** legger inn opplysningen om at $a \sim b$.

  Operasjonen heter 'Union' fordi effekten av å legge inn at *a* *b* er at ekvivalensklassene til *a* og *b* blir slått sammen (fordi de nå må tilhøre samme ekvivalensklasse).

- Vi kan løse ekvivalensproblemet på en mengde *S* ved følgende algoritme:

- Ved starten av algoritmen er hvert element i sin egen ekvivalensklasse.
- Når vi får vite at $a \sim b$, bruker vi operasjonen **Union(*a, b*)**.
- Når vi skal avgjøre om $c \sim d$, sjekker vi om **Find(*c*)** == **Find(*d*)**.

DISJUNKTE MENGDER – *Operasjoner #3*

- Legg merke til at algoritmen er dynamisk på den måten at ekvivalensklassene vil forandre seg etter hvert som vi utfører union-operasjonene.

Observasjon 1:

- Vi sammenligner ikke navnene (verdiene) til elementene direkte. Alt vi er interessert i er hvilken (ekvivalens)klasse de er i.
- Dermed kan vi for enkelhets skyld anta at at vi jobber med elementer fra 1 til N og at $E_i = \{ i \}$ når algoritmen starter.

DISJUNKTE MENGDER – *Operasjoner #4*

Observasjon 2:

- Vi bryr oss egentlig ikke så mye om navnet på klassen som **Find** returnerer.

  Det som er viktig er at **Find(a)**==**Find(b)** hvis og bare hvis $a \sim b$ (dvs. at a og b er i samme klasse).

Observasjon 3:

- Man kan velge to strategier når man implementerer disjunkte mengder:
- **Find** kan være rask, O (1), men da blir **Union** relativt treg, O (log n)
- **Union** kan være rask, O (1), men da blir **Find** relativt treg, O (log n).

- **OBS!** Det er bevist at man ikke kan få både **Find** og **Union** O (1) samtidig.

DISJUNKTE MENGDER – *Implementasjon #1*

En implementasjon som gir en rask finn-metode

- Hvis vi ønsker *O* (1) tid for **Find**, kan vi bruke en array der vi for hvert element lagrer navnet på ekvivalensklassen:

	1	2	3		N
S	E_1	E_1	E_2	…	E_2

- Da blir **Find** bare et enkelt oppslag i tabellen.
- Men **Union** er kostbar fordi vi må gå gjennom hele arrayet og bytte klassenavnet til alle elementer i klassene som skal slås sammen. Det tar *O* (*n*) tid.

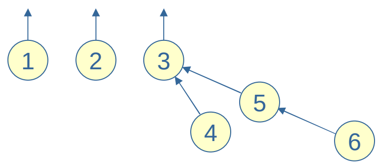
DISJUNKTE MENGDER – *Implementasjon #2*

- Ved å ta vare på størrelsen til hver ekvivalensklasse og alltid la klassen med færrest elementer bytte navn til klassen med flest elementer, kan man garantere at $N - 1$ unioner (som er det meste man kan ha før alle *N* elementene er i samme klasse) ikke tar mer enn *O* (*N* log *N*) tid.

- Det kommer av at et element *a* bare kan skifte klassetilhørighet log *N* ganger fordi antall elementer i klassen til *a* vil bli minst fordoblet ved hver eneste union.

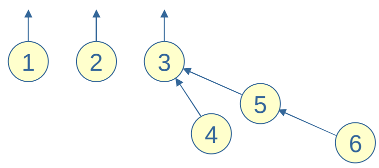
En implementasjon som gir en hurtig union-metode

- Vi implementerer operasjonene til disjunkte sett ved hjelp av en **skog**, dvs. en mengde trær.
- Ideen er å plassere alle elementer i en ekvivalensklasse i samme tre, og la roten i treet identifisere ekvivalensklassen.
- Trærne er ikke binære, men allikevel veldig enkle fordi vi bare behøver å lagre forelderpekeren for å finne roten, altså ingen barnepekere.



- Legg merke til at trærne kan representeres ved et enkelt array S fra 1 til N:
  - $S[i] == y$ betyr at node nr. i har y som forelder.
  - $S[i] == -1$ betyr at noden er en rotnode.

	1	2	3	4	5	6
S	-1	-1	-1	3	3	5



- **Union** gjøres ved å sette den ene rotpekeren til å peke på den andre roten. Det tar konstant tid hvis vi allerede kjenner røttene til klassene som skal slås sammen.
- **Find(*a*)** tilsvarer å traversere forelderpekeren opp til roten. Tidsforbruket er proporsjonalt med dybden til node *a*, og den er i verste fall $O(N)$ (hvis alle nodene er i samme ekvivalensklasse).
- Gjennomsnittsanalysen til operasjonene er veldig vanskelige.
- **Eksempel:** se side 272 i MAW.

Union-by-size

- Vi kan redusere tidsforbruket til finn-operasjonen ved å bruke en smartere union-strategi:
- Vi lar alltid det minste treet (færrest elementer) bli et subtre i det største (flest elementer).
- Med denne strategien, kalt **union-by-size**, blir dybden til et tre maks $\log N$.
- Det kommer av at når dybden til en gitt node *a* øker (med 1), så skjer det ved at treet det er med i blir slått sammen med et tre som er større enn seg selv.

- Dermed fordobles (minst) antall noder i treet hver gang dybden til *a* øker med 1.

  Det kan bare skje log *N* ganger.

- Finn-operasjonen blir altså *O* (log *n*).

- Vi må lagre størrelsen til hvert tre. Det kan typisk gjøres ved å lagre den **negative** størrelsen i tabellcellen til roten.

- **Eksempel:** se MAW side 275.

Union-by-height

- En annen union-strategi er å lagre høyden til hvert tre (den lengste veien fra roten til en bladnode), og alltid la treet med minst høyde bli subtre av treet med størst høyde.

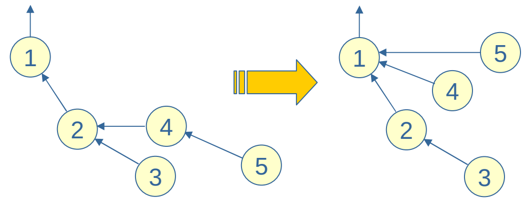
- Høyden til det nye treet vil bare øke (med 1) når trærne som slås sammen har samme høyde!

- Også denne strategien gir *O* (log *n*) tidsforbruk **worst case**.

- **Eksempel:** se MAW side 276.

Stikomprimering

- Det er sannsynligvis ikke mulig å gjøre union på en smartere måte, så vi kan i stedet prøve en lurere finn-strategi:
- Når vi skal svare på **Find(*a*)**, kan vi minske dybden til nodene i den grenen som *a* ligger i ved å forandre på forelderpekerne slik at de peker direkte på roten.



Før **Find(5)** Etter **Find(5)**

- Anta at *a* har *b* som forelder. Da kan denne strategien enkelt implementeres rekursivt ved å sette $S[a] = Find(b)$, dvs. returverdien av det rekursive kallet til forelderen.
- Denne strategien kalles **stikomprimering**.
- Man kan vise at dersom man kombinerer stikomprimering med union-by-size eller union-by-height, så vil tidsforbruket til M union/finn-operasjoner bli *nesten* $O(M)$.

Vi introduserer ADT'en for GRAFER (Graphs)

- Definisjon av en graf (MAW kapittel 9.1)
- Varianter av grafer
- Intern representasjon av grafer (MAW kapittel 9.1.1)
- Topologisk sortering (MAW kapittel 9.2)