Uke 5

Disjunkte mengder

MAW, kap. 8

- Forutsetninger for og essensen i faget
- Metodekall, rekursjon, permutasjoner
- Analyse av algoritmer
- Introduksjon til ADT'er
- Den første ADT: trær

Uke 1

- Binære trær, binære søketrær

Uke 2

- Hashing, hash-tabeller

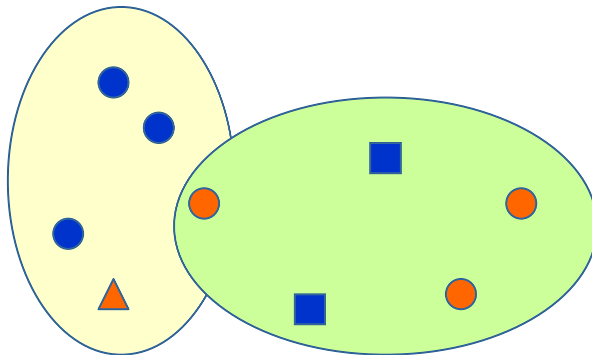
Uke 3

- B-TrærB+ trær. ADTer og stakker. Rød-svarte trær

Uke 4

Oppsummering av dagens forelesning:

- Disjunkte sett
- Relasjoner
- Ekvivalensrelasjoner
- Disjunkte sett ADT og operasjoner
- Problemer og løsninger
- Fun-time analyse



http://no.wikipedia.org/wiki/Mengde

Russells paradoks

Disjunkte mengder er mengder som ikke har felles elementer

Relasjonener

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 holder (er sann) så sier vi at a **er relatert til** b, eller at a **står i forhold til** b via R.

Eksempel:
≤ relasjonen på mengden av heltall.
$3 \leq 5$ er sann, så 3 er relatert til 5 (med hensyn på mindre-eller-lik-relasjonen). Derimot er $6 \leq 5$ usann, så 6 er ikke relatert til 5.

Parentes: Standard matematisk definisjon av relasjoner er slik: en relasjon R er en delmengde av SXS. Vi sier at aRb holder hviss $(a,b) \in R$.

Ekvivalensrelasjoner

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

1. $a \sim a$ for alle elementer *a* i S (**refleksivitet**)

2. Hvis $a \sim b$, så er $b \sim a$ (**symmetri**)

3. Dersom $a \sim b$ og $b \sim c$, så har vi også $a \sim c$ (**transitivitet**)

Eksempel:

En velkjent ekvivalensrelasjon er = relasjonen på tall.

Flere eksempler:

bilveirelasjonen:

La S være mengden av tettsteder i Norge, og la a og b være to tilfeldige tettsteder. Da er a b hvis og bare hvis det finnes en måte å komme fra a til b ved å kjøre bil uten å bruke ferge.

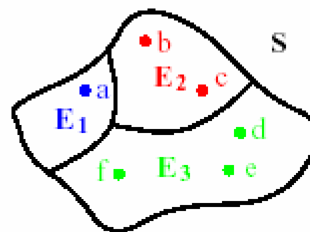
Under forutsetning av at ingen veier er enveiskjørte, er dette en ekvivalensrelasjon.

går i samme klasse som definert over elevene ved en skole er en ekvivalensrelasjon.

Ekvivalensklasser

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}

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.

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 $a \sim d$.

Viktig observasjon:

For å bestemme om $a \sim d$, er det nok å sjekke om de er i samme ekvivalensklasse!

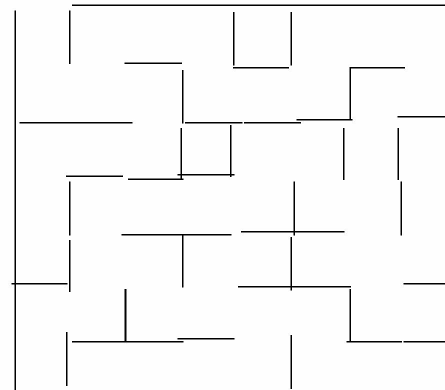
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
48	49	50	51	52	53	54	55

Eksempel: labyrint!

Fjern vegger av cellene tilfeldig, slik at de finns en vei mellom cell 0 og cell 55

```
Initialize set S of all 56 cells
as union/find ADT

while ( S.find(0) != S.find(55) ) {
   choose randomly a pair (i,j)
              of adjacent cells;
   if (find(i) != find(j)) {
       knock down the wall
             between cells i and j;
       S.union(i,j);
      }
}
```



Disjunkte mengder ADT

Weiss kap. 8.1–8.5

Løser ekvivalensproblemet

Lett og rask implementasjon

Vanskelig tidsforbrukanalyse

Operasjonene til disjunkte mengder

Disjunkt mengde ADT tilbyr to operasjoner:

Find(*a*) returnerer en representant for ekvivalensklassen til element *a*. Representanten er altid 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 $a \sim b$ 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*)**.

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.

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 disjunkt sett ADT:

- **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).

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

Implementasjon som gir rask finn

Hvis vi ønsker $O(1)$ tid for **Find**, kan vi bruke et 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.

- 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.

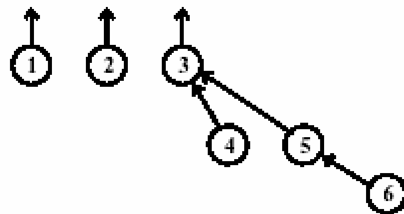
- Det kommer av at et element *a* bare kan skifte klassetilhørighet log N ganger fordi antall elementer i klassen til *a* vil bli minst fordoblet ved hver eneste union.

Implementasjon som gir hurtig union

- 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.

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 vanskelig.

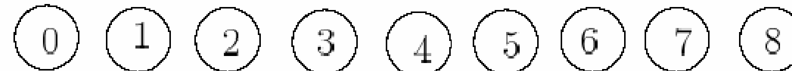
```
Initialization:
   for (i=0; i<n; i++)
      parent[i] = -1;

find(i):
   // traverse to the root
   for (j=i;
        parent[j]>=0;
        j=parent[j]);
   return j

union(i,j):
   root1 = find(i);
   root2 = find(j);
   if (root1 != root2)
      parent[root2] = root1;
```

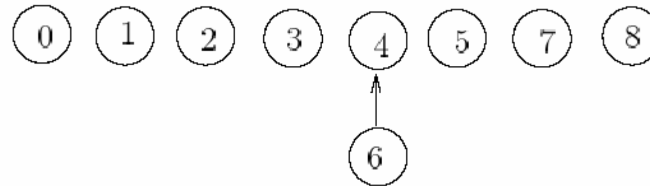
Eksampel: n=9

0	1	2	3	4	5	6	7	8
-1	-1	-1	-1	-1	-1	-1	-1	-1



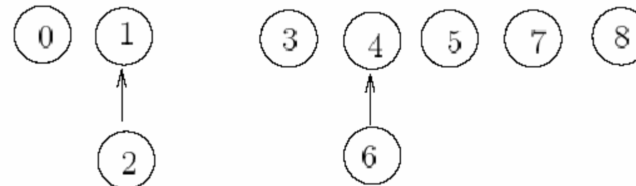
Etter Union(4,6)

0	1	2	3	4	5	6	7	8
-1	-1	-1	-1	-1	-1	4	-1	-1



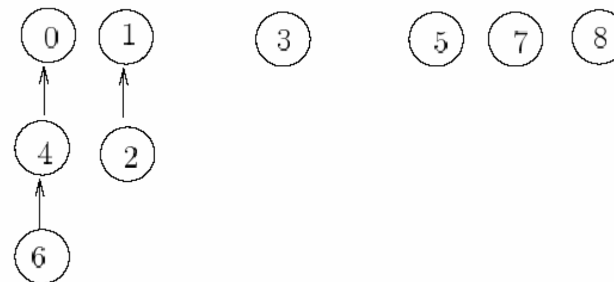
Etter Union(1,2)

0	1	2	3	4	5	6	7	8
-1	-1	1	-1	-1	-1	4	-1	-1



Etter Union(0,4)

0	1	2	3	4	5	6	7	8
-1	-1	1	-1	0	-1	4	-1	-1



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.

TEMA: Disjunkte mengder

0	1	2	3	4	5	6	7	8
-1	-1	-1	-1	-1	-1	-1	-1	-1

Union (4, 6)

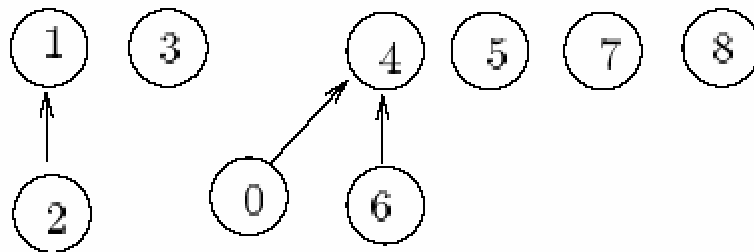
0	1	2	3	4	5	6	7	8
-1	-1	-1	-1	-2	-1	4	-1	-1

Union (1, 2)

0	1	2	3	4	5	6	7	8
-1	-2	1	-1	-2	-1	4	-1	-1

Union (0,4)

0	1	2	3	4	5	6	7	8
4	-2	1	-1	-3	-1	4	-1	-1



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.

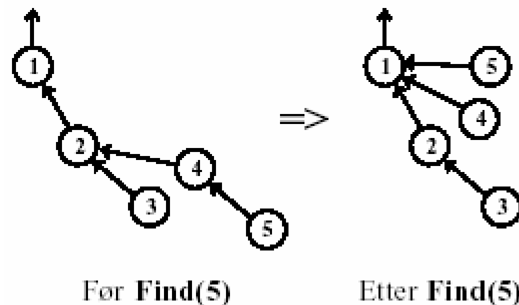
Høyden til det nye treet vil bare øke (med 1) når trærne som slås sammen har samme høyde!

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

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.



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* $O(M)$.

Neste gang:

Prioritetskøer

MAW, kap. 6