

Disjoint Sets

Partition

En **partition** (dansk: disjunkt opdeling) af en mængde S er en samling ikke-tomme delmængder A_i , $i = 1, \dots, k$, som er disjunkte og tilsammen udgør S :

$$A_i \neq \emptyset \text{ for alle } i$$

$$A_i \cap A_j = \emptyset \text{ for } i \neq j$$

$$A_1 \cup A_2 \cup \dots \cup A_k = S$$

Eksempel:

$\{a, b, e\}$, $\{f\}$, $\{c, d, g, h\}$ er en partition af $\{a, b, c, d, e, f, g, h\}$

Datastrukturen Disjoint Sets

Disjunkte opdeling som datastruktur?

Datastrukturen Disjoint Sets

Disjunkte opdeling som datastruktur? Følgende samling operationer har vist sig relevante i mange anvendelser (herunder nogle senere i kurset):

Datastrukturen Disjoint Sets

Disjunkte opdeling som datastruktur? Følgende samling operationer har vist sig relevante i mange anvendelser (herunder nogle senere i kurset):

FIND-SET(x):

Returner (en ID for) mængden indeholdende x .

UNION(x, y):

Slå $\{a, b, c, \dots, x\}$ og $\{h, i, j, \dots, y\}$ sammen til $\{a, b, c, \dots, x, h, i, j, \dots, y\}$.

MAKE-SET(x):

Opret $\{x\}$ som en ny mængde (x må ikke være element i andre mængder).

Datastrukturen Disjoint Sets

Disjunkte opdeling som datastruktur? Følgende samling operationer har vist sig relevante i mange anvendelser (herunder nogle senere i kurset):

FIND-SET(x):

Returner (en ID for) mængden indeholdende x .

UNION(x, y):

Slå $\{a, b, c, \dots, x\}$ og $\{h, i, j, \dots, y\}$ sammen til $\{a, b, c, \dots, x, h, i, j, \dots, y\}$.

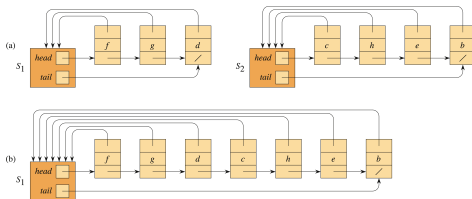
MAKE-SET(x):

Opret $\{x\}$ som en ny mængde (x må ikke være element i andre mængder).

NB: Vi har ingen krav til ID for mængder. Den skal blot være ens for alle x i samme mængde (\Rightarrow vi kan checke om to elementer x og y ligger i samme mængde).

Disjoint Sets implementeret via lænkede lister

Hver mængde er en lænket liste af elementer, ID for mængde er første element i listen:



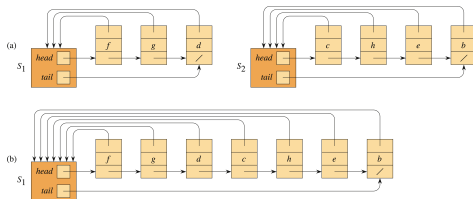
- $FIND-SET(x)$: returner (via header-pointer) første element i listen.
- $MAKE-SET(x)$: opret ny liste.
- $UNION(x, y)$: slå lister sammen, behold én header, ændrer alle header-pointere i den anden liste.

Disjoint Sets implementeret via l  nkede lister

K  retid (n er antal elementer, dvs. antal MAKE-SET udf  rt)?

Disjoint Sets implementeret via lænkede lister

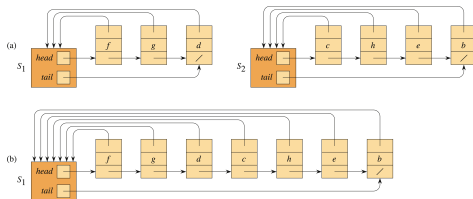
Køretid (n er antal elementer, dvs. antal MAKE-SET udført)?



- FIND-SET(x): returner (via header-pointer) første element i listen: $O(1)$.
- MAKE-SET(x): opret ny liste: $O(1)$.
- UNION(x, y): slå lister sammen, behold én header, ændr alle header-pointere i den anden liste: $O(n)$.

Disjoint Sets implementeret via lænkede lister

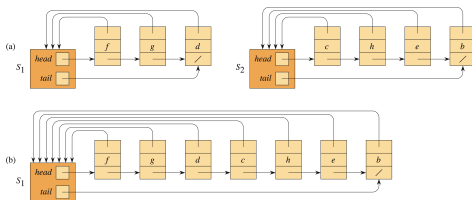
Køretid (n er antal elementer, dvs. antal MAKE-SET udført)?



- ▶ FIND-SET(x): returner (via header-pointer) første element i listen: $O(1)$.
- ▶ MAKE-SET(x): opret ny liste: $O(1)$.
- ▶ UNION(x, y): slå lister sammen, behold én header, ændr alle header-pointere i den anden liste: $O(n)$.

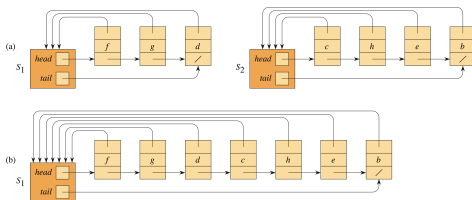
Naiv analyse: n MAKE-SET, op til $n - 1$ UNION, og m FIND-SET koster $O(m + n^2)$.

Disjoint Sets implementeret via l nkede lister, version 2



- FIND-SET(x): returner (via header-pointer) f rste element i listen: $O(1)$.
- MAKE-SET(x): opret ny liste: $O(1)$.
- UNION(x, y): sl  lister sammen, behold header af l ngste liste,  ndr alle header-pointere i korteste liste: $O(n)$.

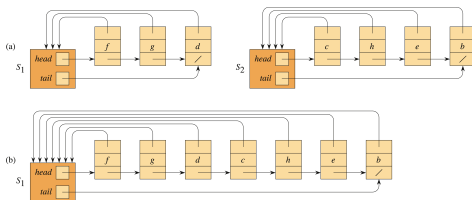
Disjoint Sets implementeret via l nkede lister, version 2



- FIND-SET(x): returner (via header-pointer) f rste element i listen: $O(1)$.
- MAKE-SET(x): opret ny liste: $O(1)$.
- UNION(x, y): sl  lister sammen, behold header af l ngste liste,  ndr alle header-pointere i korteste liste: $O(n)$.

Observ r at nu g lder: en knude kan kun  ndre sin header-pointer log n gange, da st rrelsen af dens m ngde hver gang vokser mindst en faktor to ($1 \cdot 2^k \leq n \Leftrightarrow k \leq \log n$).

Disjoint Sets implementeret via lænkede lister, version 2



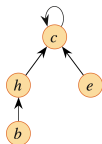
- ▶ FIND-SET(x): returner (via header-pointers) første element i listen: $O(1)$.
- ▶ MAKE-SET(x): opret ny liste: $O(1)$.
- ▶ UNION(x, y): slå lister sammen, behold header af **længste liste**, ændr alle header-pointere i **korteste liste**: $O(n)$.

Observér at nu gælder: en knude kan kun ændre sin header-pointer $\log n$ gange, da størrelsen af dens mængde hver gang vokser mindst en faktor to ($1 \cdot 2^k \leq n \Leftrightarrow k \leq \log n$).

Så bedre analyse: n MAKE-SET, op til $n - 1$ UNION, og m FIND-SET koster $O(m + n \log n)$.

Disjoint Sets implementeret via træer

Hver mængde er et træ med elementer i knuder, rod er ID for mængde:



(a)



(b)

- ▶ $\text{FIND-SET}(x)$: gå til rod.
- ▶ $\text{MAKE-SET}(x)$: opret nyt træ.
- ▶ $\text{UNION}(x, y)$: gør rod af ét træ til barn af andet træ.

Disjoint Sets implementeret via træer, version 2

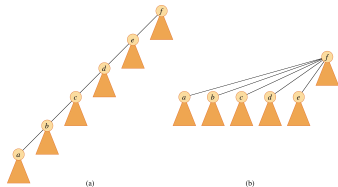
Disjoint Sets implementeret via træer, version 2

Union by rank: Tilføj et heltal (kaldet rank) til alle rødder. Sættes til 0 ved nye træer (under $\text{MAKE-SET}(x)$). Rank kontrollerer forældre-barn beslutningen ved $\text{UNION}(x, y)$: rod med størst rank får den anden rod som barn. Ved ens rank, lad y få x som barn og øg y 's rank med 1.

Disjoint Sets implementeret via træer, version 2

Union by rank: Tilføj et heltal (kaldet rank) til alle rødder. Sættes til 0 ved nye træer (under $\text{MAKE-SET}(x)$). Rank kontrollerer forældre-barn beslutningen ved $\text{UNION}(x, y)$: rod med størst rank får den anden rod som barn. Ved ens rank, lad y få x som barn og øg y 's rank med 1.

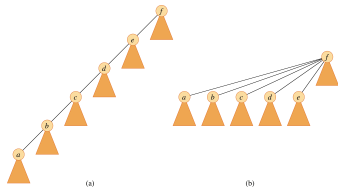
Path compression: Under $\text{FIND-SET}(x)$, sæt alle passerede knuder op som børn af roden. Eksempel med $\text{FIND-SET}(a)$:



Disjoint Sets implementeret via træer, version 2

Union by rank: Tilføj et heltal (kaldet rank) til alle rødder. Sættes til 0 ved nye træer (under $\text{MAKE-SET}(x)$). Rank kontrollerer forældre-barn beslutningen ved $\text{UNION}(x, y)$: rod med størst rank får den anden rod som barn. Ved ens rank, lad y få x som barn og øg y 's rank med 1.

Path compression: Under $\text{FIND-SET}(x)$, sæt alle passerende knuder op som børn af roden. Eksempel med $\text{FIND-SET}(a)$:



Union by rank og path compression \Rightarrow meget tæt på $O(m + n)$ tid. Mere præcist $O(m \cdot \alpha(n) + n)$, hvor $\alpha(n)$ er en meget langsomt voksende funktion.

Funktionen $\alpha(n)$ samt beviset for køretiden findes i afsnit 19.4, som ikke er pensum (gennemgås i et senere kursus på datalogistudiet).

Disjoint Sets implementeret via træer, version 2

Pseudokode (med union by rank og path compression) er forbavsende simpel:

MAKE-SET(x)

$x.p = x$

$x.rank = 0$

UNION(x, y)

LINK(FIND-SET(x), FIND-SET(y))

FIND-SET(x)

if $x \neq x.p$

$x.p = \text{FIND-SET}(x.p)$

return $x.p$

LINK(x, y)

if $x.rank > y.rank$

$y.p = x$

else $x.p = y$

// If equal ranks, choose y as parent and increment its rank.

if $x.rank == y.rank$

$y.rank = y.rank + 1$