DMA 2016
- Noter til uge 11 -

Vi diskuterer her datastrukturer til diskunkte mængder eller *forén og find* som i CLRS kapital 21, men med en lidt anden tilgang[1].

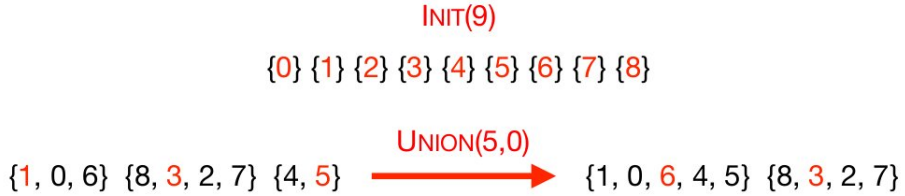
Forén og find

Vi er interesseret i at vedligeholde en dynamisk familie af mængder. Hver mængde i familien har én repræsentant, som identificerer mængden. Vi ønsker at kunne finde den mængde et given element befinder sig i, og vi vil kunne forene to mængder. Som en simplificering kigger vi her på mængder af tal, men datastrukturen kan ligeledes fungere med mængder af objekter. Vores datastruktur skal understøtte følgende operationer.

- Init(n): Opret n nye mængder $\{0\}, \{1\}, \cdots, \{n-1\}$, med ét element i hver.
- Union(i,j): Forén de to mængder som indeholder i og j. Hvis i og j er i samme mængde, der skal ingenting ske.
- Find(i): Returnér repræsentanten for den mængde som inderholder i.

Input til Init(n) er antallet af nye mængder som skal konstrueres. Et kald til Init(n) svarer til n kald til Make-Set i CLRS. Til Find og Union er input elementer i og j, som befinder sig i en henholdsvis to af de mængder vi vedligeholder. Bemærk at hver mængde har præcis én repræsentant, og et hvilken som helst element i en mængde kan være repræsentant. Dermed har vi Find(i) == Find(j) præcis når i og j er i samme mængde.

Hvis vi kalder Init(9) får vi ni mængder som illustreret herunder. Hver mængde indeholder ét element, der repræsenterer mængden. Som eksempel på brug af Union kalder vi Union(5,0) i en situation hvor vi har mængderne $\{1, 0, 6\}$, $\{8, 3, 2, 7\}$ og $\{4, 5\}$.



[1]Disse noter er stærkt inspireret af noter af Philip Bille og Inge Li Gørtz til kurset Algoritmer og Datastrukturer, på DTU, http://www2.compute.dtu.dk/courses/02105+02326/2015/#generelinfo

Hurtig forening Hvordan kan vi konkret implementere denne abstrakte datastruktur, på en effektiv måde? En mulighed er repræsentere hver mængde som et rodfæstet træ, hvor roden er repræsentant for mængden. Dette kan vi konkret gøre ved at benytte en tabel p[0..$n-1$], hvor hver indgang, p[i], indeholder en *forældrepeger*. Roden i et træ er sin egen forælder.

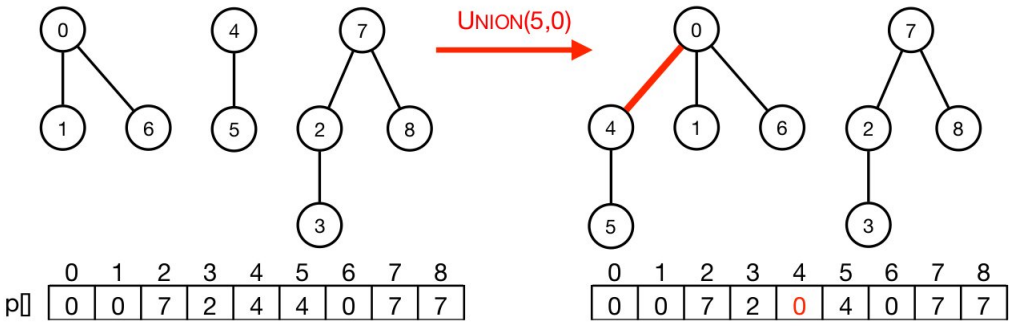
- Init(n): Opret ny tabel p[0..$n-1$], hvor der er n mængder med ét element. Hver indgang er sin egen forældre, så vi sætter p[i] $= i$ for alle $i = 0, \cdots, n-1$.
- Union(i,j): Hvis Find(i) $\neq$ Find(j), gør roden af det ene træ, til roden af det andet.
- Find(i): Følg forældrepegere til roden, og returnér roden.

```
INIT(n):
   for k = 0 to n-1
      p[k] = k
```

```
FIND(i):
   while (i != p[i])
      i = p[i]
   return i
```

```
UNION(i,j):
   rᵢ = FIND(i)
   rⱼ = FIND(j)
   if (rᵢ ≠ rⱼ)
      p[rᵢ] = rⱼ
```

Her er et eksempel på hvilken familie af træer, som tabellen p[0, 0, 7, 2, 4, 4, 0, 7, 7] repræsenterer, og hvilken effekt Union har.

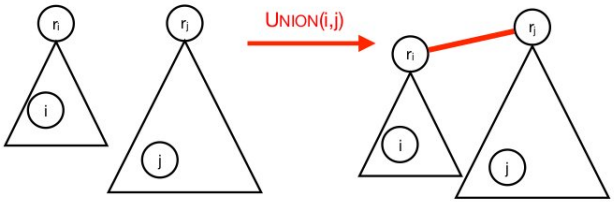


Køretiden Det er klart at køretiden af Init(n) bliver $\Theta(n)$ og køretiden af Find(j) afhænger af dybden af vores træ, fordi vi følger en direkte vej gennem træet til roden. Vi definerer dybden af et træ til at være den maksimale længde af en sti, fra rod til blad. Hvis vi kalder dybden for d, har vi at Find(i) er $O(d)$. Da union blot kalder Find et fast antal gange, vil køretiden for Union også blive $O(d)$. Effektiviteten afhænger dermed meget af hvor dybe vores træer kan blive.

Desværre er det muligt at lave en sekvens af operationer, som laver et træ hvis dybde er $n-1$. Tænk f.eks. på tabellen p[0, 0, 1, 2, 3, 4], som repræsenterer et træ med dybde 5. Heldigvis kan vi forene træerne på en snedig måde, som begrænser dybden betydeligt.

Vægtet forening Idéen er, at når vi forener to træer tager vi roden til det mindste træ, og får den til et pege på roden i det større træ. Derved vil dybden af det nye træ, være lig med dybden af det største af de oprindelige træer. Størrelsen af en knude, er antallet af kunder i deltræet. For at gemme denne ekstra data bruger et en tabel sz$[0..n-1]$

- Init(n): Ligesom før, bortset fra vi sætter sz$[i] = 1$ for alle $i = 0, \cdots, n-1$.
- Find(i): Ligesom før.
- Union(i,j): Hvis Find(i) $\neq$ Find(j), gør roden af det mindste træ til barn af roden af det største træ.




```
UNION(i,j):
    rᵢ = FIND(i)
    rⱼ = FIND(j)
    if (rᵢ ≠ rⱼ)
        if (sz[rᵢ] < sz[rⱼ])
            p[rᵢ] = rⱼ
            sz[rⱼ] = sz[rᵢ] + sz[rⱼ]
        else
            p[rⱼ] = rᵢ
            sz[rᵢ] = sz[rᵢ] + sz[rⱼ]
```

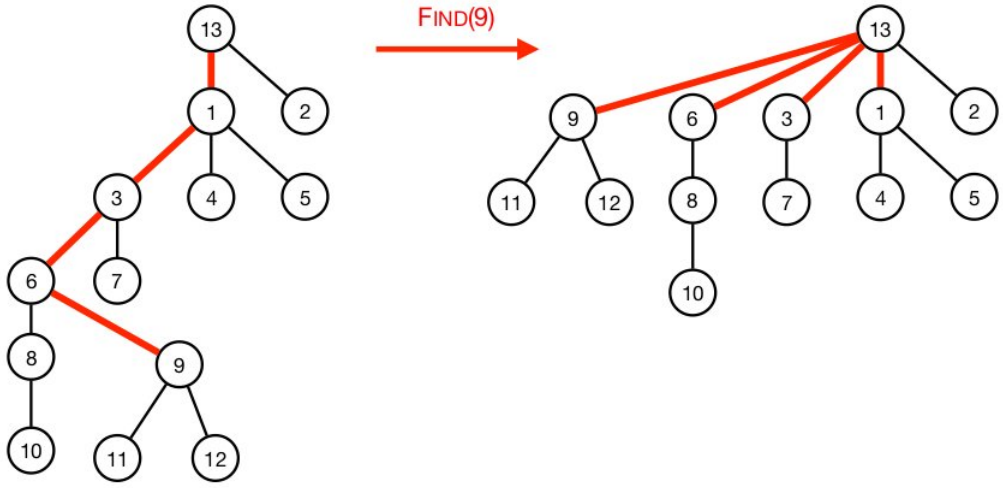
Lemma 1. *Med vægtet forening er dybden d af en knude $\leq \log n$.*

Bevis. Tag en knude j. Hver gang j's repræsentant ændres, som følge af et kald til union, forøges antallet af elementer i j's mængde med (mindst) en faktor af 2. Dette skyldes at j's repræsentant kun ændres, når j's mængde er den mindste af de to mængder som forenes.

Så hvis j's repræsentant ændres k gange, indeholder j's mængde mindst 2^k elementer. Hvis der er n elementer i alt, har vi $k \leq \log n$. Da dybden af j øges med 1 præcis når j forenes med en større mængde har vi $d = k \leq \log n$. □

Køretid Med vægtet forening bliver køretiden dermed $O(n)$ for Init(n), og $O(\log n)$ for Find og Union. Spørgsmålet er om dette kan forbedres, og hvad er den bedst mulige køretid vi kan opnå?

Sti kompression Køretiden for Find dominerer i Forén og find. Når vi kalder Union, bruges tiden på at finde repræsentater for mængderne. Selve det at opdatere forældrepegere tager konstant tid. Så vi fokuserer på at forbedre Find. Når vi kalder Find(i) følger vi en sti fra i op til roden. På denne sti kommer vi forbi nogle andre knuder i træet. Idéen er at opdatere forældrepegeren for hver knude vi kommer forbi, til roden af træet. Så vi "presser"stien sammen. Efterfølgende kald til Find(j), hvor j er en af de knuder vi mødte på stien, er derved hurtige fordi deres forælder nu er selve roden. Dette ændrer ikke på køretiden af Find.



Køretid For at bestemme køretiden skal vi nu bruge amortiseret analyse. Vi ser altså ikke længere på køretiden af et enkelt kald til Find, Union og Init, men på køretiden af en sekvens af f.eks. m Find og Union operationer på en mængde med n elementer.

Tarjan's oprindelige bevis for køretiden er en anelse indviklet, men i 2005 fandt Raimund Seidel og Micha Sharir et mere simpelt bevis. Hvis I er interesseret i læse dette, kan I finde en udlægning i disse noter. Disse noter inderholder også andre interessante måder at implementere forén-find strukturen.

Theorem 2 (Tarjan 1975)**.** *Med stikompression og vægtet forening tager en sekvens af m Find og Union operationer, på en mængde med n elementer, $O(n + m\alpha(m, n))$ tid.*

Funktionen $\alpha(m, n)$ er den inverse til Ackermans funktion. Den vokser meget langsomt. For alle tal vi realistisk kan forestille os, vil $\alpha(m, n) \leq 5$. Den er dog ikke $O(1)$: For tilpas store tal vil den blive større en enhver given konstant, men i praktisk siger sætnigen ovenfor, at vi næsten har en køretid på $O(m + n)$. Køretiden er altså næsten lineær i antallet af operationer. Bemærk at hvis Find og Union tager $O(\log n)$ tid vil køretiden for en sekvens af operationer blive $O(n + m \log n)$. Med stikompression og vægtet forening har vi altså opnået en mærkbar forbedring. Spørgsmåler nu, kan vi gøre det endnu bedre? Svaret er nej. I 1985 blev følgende bevist.

Theorem 3 (Fredman-Saks 1985)**.** *Det er ikke muligt at understøtte m Find og Union operationer i $O(m + n)$ tid.*

Med stikompression og vægtetforening har vi en optimal løsning på foren-find datastrukturen. Her er en kort opsummering af vores fund.

Datastruktur	m UNION og FIND
hurtig forening	O(mn)
vægtet forening	O(n + mlog n)
vægtet forening + stikompression	O(n + m α(m,n))
umuligt	O(n + m)