Algoritmeanalyse

Identificer essentiel(le) operation(er)

Øvre grænse for algoritme

Find øvre grænse for antallet af gange de(n) essentielle operation(er) udføres.

Øvre grænse for problem

Brug øvre grænse for en god algoritme.

Nedre grænse for problem

- Adverary argument
  Eks: Find max.
- Beslutningstræ
  Eks: sortering

Asymptotisk notation

For en funktion f er $o(f)$, $O(f)$, $\Theta(f)$, $\Omega(f)$ og $\omega(f)$ mængder af funktioner karakteriseret på følgende måde:

$o(f)$: “vokser langsommere end f”

$O(f)$: “vokser højst så hurtigt som f”

$\Theta(f)$: “vokser som f”

$\Omega(f)$: “vokser mindst så hurtigt som f”

$\omega(f)$: “vokser hurtigere end f”

Korrekthed

while-løkker

Partiel korrekthed:

- Bevis, at invarianten er opfyldt
  — første gang vi kommer til while-løkken.
  — hver af de efterfølgende gange.
- Bevis, at invarianten + negeringen af betingelsen i while-løkken medfører post-betingelsen.

Terminering:

Find termineringsfunktion f, så

1. $f(P_{i+1}) < f(P_i)$, for alle $i \geq 1$
2. $f(P_i) \geq 0$, for alle i

P_i: værdierne af variablerne efter i'te gennemløb af løkken.

rekursive algoritmer

Terminering:

Find termineringsfunktion f, så

1. $f(P') < f(P)$, for alle rekursive kald
2. Når f bliver tilstrækkeligt lille, bliver et basistilfælde for algoritmen anvendt.

P: Værdierne af aktuelle parametre
P': Værdierne af parametre til rekursivt kald

Partiel korrekthed:

Bevis korrekthed ved induktion i termineringsfunktionen.

Datatyper

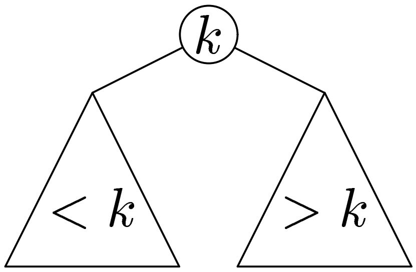
- Hægtet liste
- Binært træ
- FIFO-kø
- Stak
- Prioritetskø: heap
- Dictionary (ordbog): rød-sort søgetræ, hash-tabel
- Disjunkte mængder

Dictionary (ordbog)

`insert`, `delete`, `søgning`

Binære søgetræer

Nøglen i en knude er større end nøglerne i venstre undertræ og mindre end nøglerne i højre undertræ.



`insert`(x): Find vha. `search` den plads i bunden af træet, hvor x passer ind, og indsæt den her.

`delete`(x): Find knuden med x vha. `search`.
Knuden med x har 0, 1 eller 2 børn:

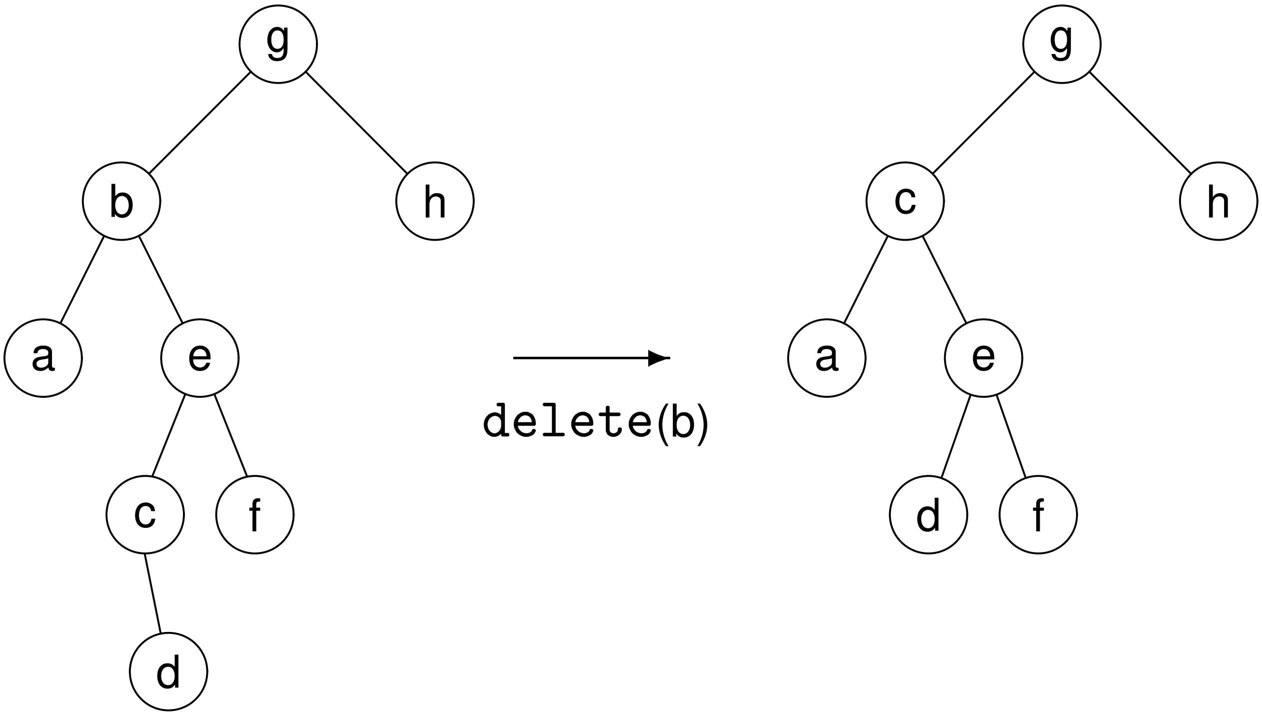
0 : knuden med x kan blot slettes.

1 : x overskrives med elementet i barnet, og barnet slettes.

2 : x overskrives med sin efterfølger y, som er længst til venstre i x's højre undertræ. Derefter slettes knuden, som indeholdt y (den har 0 eller 1 barn).

Logisk slettede knude: Knuden hvis indhold forsvinder.
Strukturelt slettede knude: Knuden som forsvinder.

Eks:



Logisk slettede knude: b

Strukturelt slettede knude: d

Rød-sorte søgetræer

`insert`, `delete`, `search`: $\Theta(\log n)$

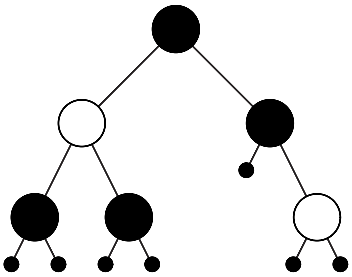
Logaritmisk højde sikres ved at opretholde flg.

- For enhver knude v gælder:

  alle stier fra v til et blad har

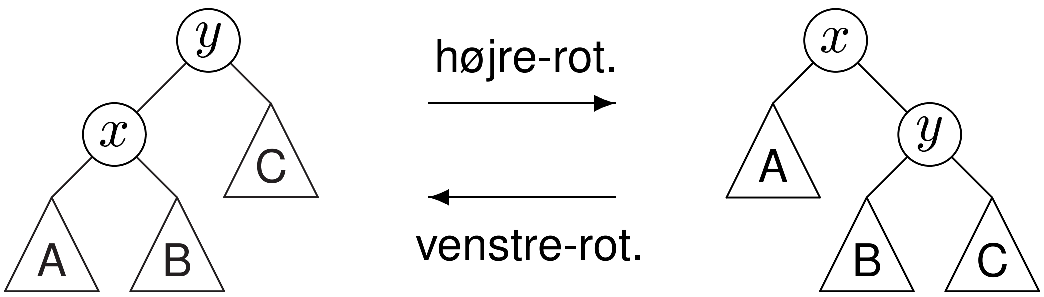
  samme # sorte knuder

- Ingen rød knude har et rødt barn.
- Roden og bladene er sorte.



Til dette formål anvendes rotationer og farveskift.

`insert` og `delete` udføres som i alm. binære søgetræer, efterfulgt af $O(1)$ rotationer og $O(\log n)$ farveskift.



`insert` :

Den indsatte knude farves rød.

Hvis forælderen er

- rød: ryd op!
- sort: færdig!

`delete` :

Hvis den strukturelt slettede knude er

- rød: færdig!
- sort: ryd op!

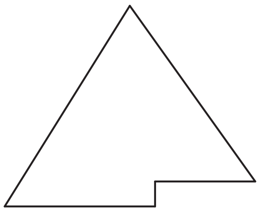
Prioritetskøer

Heap

`getMax`	$\Theta(1)$
`deleteMax`	$\Theta(\log n)$
`insert`	$\Theta(\log n)$

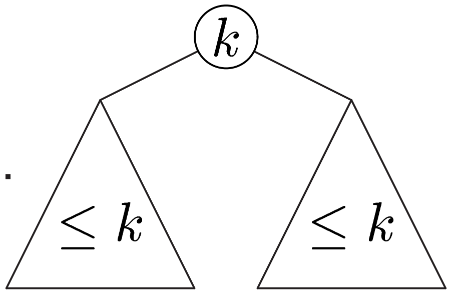
Struktur-invariant:

- De øverste $h-1$ lag er helt fyldt op.
- Det nederste lag fyldes op fra venstre.



Ordnings-invariant:

- Nøglen i en knude er mindst lige så stor som nøglerne i dens børn.



`getMax`: returnerer elementet i roden.

`deleteMax`: sletter elementet i roden, og bobler den tomme knude ned, til det sidste element i nederste lag kan indsættes uden at ødelægge den partielle ordning.

`insert`: opretter ny, tom knude på første ledige plads i nederste lag, og bobler knuden op, til det nye element kan indsættes uden at ødelægge den partielle ordning.

Disjunkte Mængder

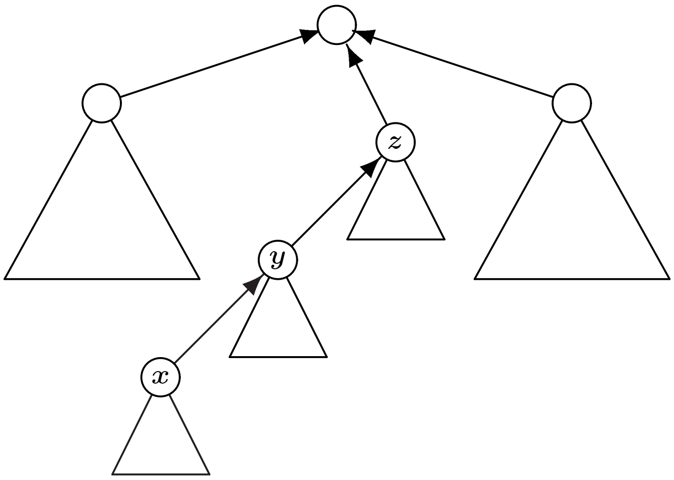
Implementering vha. træer:

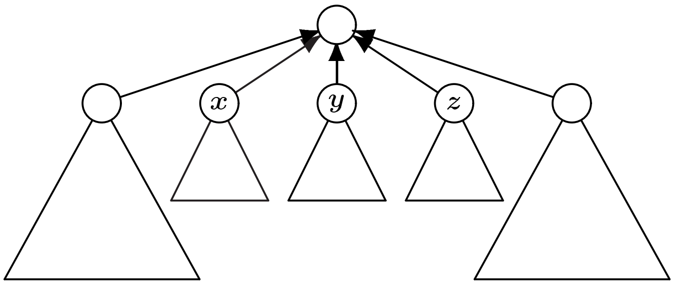
Hvert træ repræsenterer en mængde.

`find`(x) returnerer roden i træet, som indeholder x.

`find` m. path compression:

alle knuder på stien fra x til roden gøres til børn af roden.

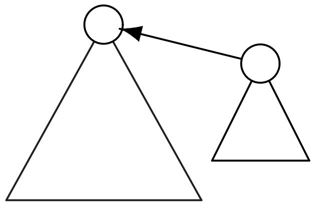




union(x,y): x og y skal være rødder.

x gøres til barn af y.

Vægtet union: roden i træet med færrest knuder gøres til barn af roden i træet med flest knuder.



Med vægtet union og `find` med path compression bliver worst-case kompleksiteten

$$O\big((n+m)\log^* n\big)$$

n: samlet # elementer

m: # `union`/`find` operationer.

$\log^*$: "# gange man skal tage $\log$ for at komme ned på 1".

$\log^* n \leq 5$ for alle praktiske formål.

Design-teknikker

Dynamisk programmering

Husk løsninger til delproblemer, så hvert delproblem kun skal løses én gang.

Rekursiv løsning:

- Identificer hukommelsesparametre.
- Lav tabel til løsninger af delproblemer.
- Inden et delproblem løses, undersøges om den tilsvarende tabelplads er tom.
- Når et delproblem løses, skrives løsningen i tabellen.

Iterativ løsning:

- Identificer hukommelsesparametre.
- Lav tabel til løsninger af delproblemer.
- Udfyld alle tabelpladser i fornuftig rækkefølge.

Eks:

Find lgd. af længste fælles delfølge af to strenge x og y.

Kald nedenstående algoritme med `llcs`$(|x|, |y|, 0, 0)$

```
llcs(m,n,i,j)
Hvis i=m eller j=n
  Returner 0
Ellers hvis x[i]=y[i]
  Returner 1 + llcs(i+1,j+1)
Ellers
  Returner max{llcs(i,j+1),llcs(i+1,j)}
```

Rekursiv løsning:

Kald nedenstående algoritme med `llcsWrap`$(|x|, |y|)$.

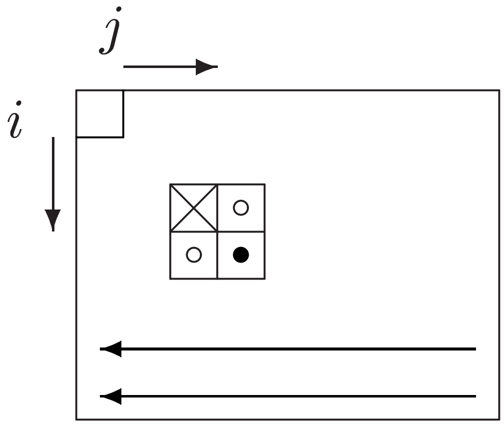
```
llcsWrap(m,n)
For (i=0; i≤m; i++)
  For (j=0; j≤n; j++)
    tabel[i][j] = −1
Returner llcs(m,n,0,0)

llcs(m,n,i,j)
Hvis tabel[i][j] = −1
  Hvis i=m eller j=n
    tabel[i][j] = 0
  Ellers hvis x[i]=y[i]
    tabel[i][j] = 1 + llcs(i+1,j+1)
  Ellers
    tabel[i][j] =
              max{llcs(i,j+1),llcs(i+1,j)}
Returner tabel[i][j]
```

Iterativ løsning:

Kald nedenstående algoritme med `llcs`$(|x|, |y|)$.

```
llcs(m,n)
for (i=0; i≤m; i++)
  tabel[i][n] = 0
for (j=0; j≤n−1; j++)
  tabel[m][j] = 0
for (i=m−1; i≥0; i--)
  for (j=n−1; j≥0; j--)
    Hvis x[i]=y[j]
      tabel[i][j] = 1 + tabel[i+1][j+1]
    Ellers
      tabel[i][j] = max{tabel[i][j+1],
                        tabel[i+1][j]}
Returner tabel[0][0]
```

Rækkefølge:



$\boxtimes$ afhænger kun af $\boxed{\circ}$ eller $\boxed{\bullet}$.

Resultatet kan aflæses i $\square$.

Del og Hersk

Eks: Mergesort, binær søgning.

- Opdel i mindre delproblemer, som løses rekursivt.
  Helt små problemer løses direkte.

- Kombiner løsninger til delproblemer.

Kompleksitet:

$$T(n) = \begin{cases} B(n), & \text{hvis } n \text{ er helt lille} \\ D(n) + \sum_{i=1}^{k} T(n_i) + C(n), & \text{ellers} \end{cases}$$

$B(n)$: tiden for at løse problemet direkte.

$D(n)$: tiden for at opdele problemet.

$C(n)$: tiden for at kombinere delløsningerne.

n_i: størrelsen af i'te delproblem.

NB: denne type rekursionsligninger kan ofte løses vha.
Master Theorem.

Rekursionsligninger

Substitution:

Gæt en løsning, og bevis vha. induktion, at den er rigtig.

Udfoldning: "Fold ud", indtil du kan se mønsteret.

Master Theorem: Kan bruges til rek.ligninger på formen

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n),$$

som opfylder betingelsen i 1., 2., eller 3.

1. $$\exists \varepsilon > 0\colon\ f(n) \in O(n^{\log_b a - \varepsilon})$$
$$\Downarrow$$
$$T(n) \in \Theta(n^{\log_b a})$$

2. $$f(n) \in \Theta(n^{\log_b a})$$
$$\Downarrow$$
$$T(n) \in \Theta(n^{\log_b a} \log n)$$

3. $$\begin{cases} \exists \varepsilon > 0\colon\ f(n) \in \Omega(n^{\log_b a + \varepsilon}) \\ \exists c < 1, n_0 \in \mathbb{N}\colon\ \forall n \geq n_0 : af(\frac{n}{b}) \leq cf(n) \end{cases}$$
$$\Downarrow$$
$$T(n) \in \Theta(f(n))$$

Sortering

Algoritme	worst-case	average-case	best-case	in-place
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\times$
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	
Heapsort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\times$
Radix sort	$\Theta(n \log_k M)$	$\Theta(n \log_k M)$	$\Theta(n \log_k M)$	

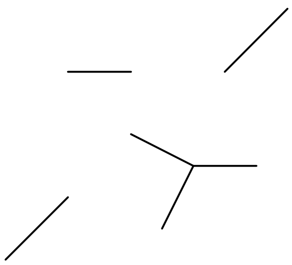
M: største tal i input

k: # spande

$\log_k M$: # “cifre”

Letteste udspændende træ

Kruskals algoritme $\Theta(m \log n)$

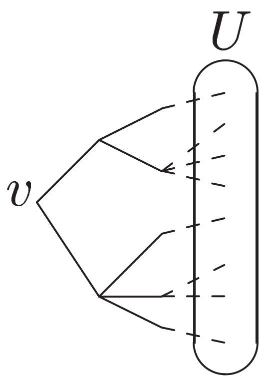


Vælg letteste kant,
som ikke danner en kreds.

Prims algoritme

— med prioritetskøen impl. vha.

- array: $\Theta(n^2)$
- heap: $\Theta(m \log n)$
- fibonacci heap: $\Theta(m + n \log n)$



Vælg letteste kant
mellem træet og U.

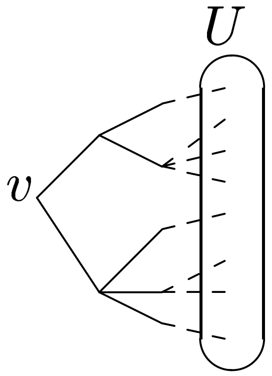
Korteste veje

<u>Bredde-først-søgning</u> $\Theta(m + n)$

Hvis alle kanter har samme vægt.

<u>Dijkstras algoritme</u> kompl. som Prims algoritme

Hvis alle kanter har ikke-negative vægte.



Vælg knude i U med

min. afstandsestimat til v.

Topologisk sortering

Dybde-først postorder-nummerering giver omvendt topologisk orden i tid $\Theta(m+n)$.

<u>Kritisk sti</u> (længste sti)

Brug dybde-først-søgning: $\Theta(m+n)$

(Algoritme i bogen s. 357)

Stærke sammenhængskomponenter

(Orienterede grafer)

(1) Lav en dybde-først postorder-nummerering.

(2) Inverter grafen (behold numrene).

(3) Lav et dybde-først-gennemløb af den inverterede graf (hver gang man går i stå, startes med højst nummererede ikke-besøgte knude).

(4) Aflæs komponenterne fra de konstruerede dybde-først-træer.

2-sammenhængende komponenter

(Ikke-orienterede grafer)

k-sammenhængende:

Man skal fjerne mindst k knuder for at gøre grafen usammenhængende.

Eks:

Træer er 1-sammenhængende

Kredse er 2-sammenhængende

Artikulationspunkt:

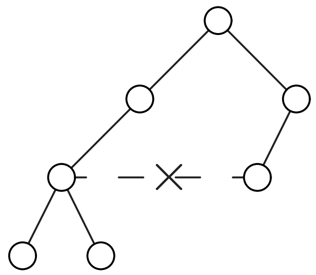
Hvis knuden fjernes, er grafen ikke længere sammenhængende.

Se på et dybde-først-træ.

En knude x er et artikulationspunkt, hvis og kun hvis

- x er roden og har mindst 2 børn
  eller
- x er *ikke* roden, og x har et barn y, så ingen af y's efterkommere (inkl. y) har en baglænskant til en af x's ægte forfædre (ekskl. x).

(Bevises let ved at huske på, at der ikke er tværkanter i et dybde-først-træ i en ikke-orienteret graf.)



Derfor kan artikulationspunkter findes vha. ét dybde-først-gennemløb. (Algoritme i bogen s. 372)