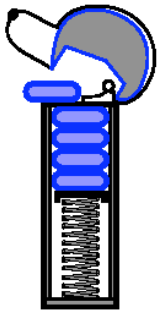
# Elementære datastrukturer

Stakke, køer og lister
Træer
Prioritetskøer
Hashing





# Stak ADT

Den abstrakte datatype (ADT) **stak** indeholder vilkårlige objekter.

Indsættelse og fjernelse følger sidst-ind-først-ud princippet.

Centrale operationer:
- **push**(object): indsætter et objekt
- object **pop**(): fjerner og returnerer det sidst indsatte objekt

Hjælpeoperationer:
- object **top**(), integer **size**(), boolean **isEmpty**()



# Anvendelser af stakke

Direkte anvendelser
- Besøgte sider i en Web-browser
- Fortryd-sekvens i et tekstredigeringsprogram
- Kæde af metodekald i et køretidssystem (f.eks. JVM)

Indirekte anvendelser
- Hjælpedatastruktur i algoritmer
- Komponent i andre datastrukturer



# Array-baseret stak

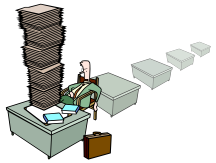
Objekter tilføjes fra venstre mod højre.

En variabel $t$ holder rede på indekset af topelementet (størrelsen er $t + 1$).

**Algorithm** *push*(*o*)
  **if** $t = S.length - 1$ **then**
    **throw** *FullStackException*
  **else**
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$

**Algorithm** *pop*()
  **if** *isEmpty*() **then**
    **throw** *EmptyStackException*
  **else**
    $t \leftarrow t - 1$
    **return** $S[t + 1]$

# Dynamisk array-baseret stak



I **push** kan vi - i stedet for at kaste en undtagelse, hvis arrayet er fyldt op - erstatte arrayet med et større array

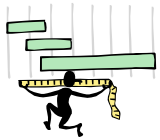
Hvor stort skal det nye array være?

- Inkrementel strategi: øg størrelsen med en konstant, $c$
- Fordoblingsstrategi: fordobl størrelsen

**Algorithm** *push*($o$)
  **if** $t = S.length - 1$ **then**
    $A \leftarrow$ new array of size …
    **for** $i \leftarrow 0$ **to** $t$ **do**
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $t \leftarrow t + 1$
  $S[t] \leftarrow o$



# Sammenligning af strategier



Vi sammenligner de to strategier ved at analysere den samlede køretid $T(n)$, der skal benyttes for at udføre en serie af af $n$ push-operationer

Det antages, at stakken fra starten er tom, og at arrayet har størrelsen 1

Ved den **amortiserede tid** for en push-operation vil vi forstå operationens gennemsnitlige tidsforbrug, d.v.s. $T(n)/n$



# Analyse af den inkrementelle strategi

Vi erstatter arrayet $k = n/c$ gange

Det samlede tidsforbrug $T(n)$ for en serie af $n$ push-operationer er proportionalt med

$$n + c + 2c + 3c + 4c + \ldots + kc =$$
$$n + c(1 + 2 + 3 + \ldots + k) =$$
$$n + ck(k + 1)/2$$

Da $c$ er en konstant, har vi, at $T(n)$ er $O(n + k^2)$, og dermed $O(n^2)$

Den amortiserede tid for en push-operation er $O(n)$



# Direkte analyse af fordoblingsstrategien

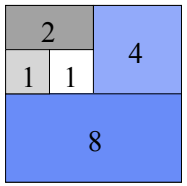
Vi erstatter arrayet $k = \log_2 n$ gange

Det samlede tidsforbrug $T(n)$ for en serie af $n$ push-operationer proportional med

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$n + 2n - 1 =$$
$$3n - 1$$

$T(n)$ er $O(n)$

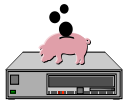
kvotientrække




$1 + 1 + 2^1 + 2^2 + 2^3 = 2^2 2^2 = 2^4$

Den amortiserede tid for en push-operation er $O(1)$

# Bogholdermetoden



**Bogholdermetoden** bestemmer amortiseret køretid ved brug af et system af indtægter og udgifter

Computeren betragtes som en betalingsautomat, der kræver et fast beløb (1 krone) for at udføre en konstant mængde af beregninger

Der opstilles et skema for betalingen. Skemaet, der kaldes for **amortiserings-skemaet**, skal være konstrueret sådan, at der altid er penge nok til at betale for den aktuelle operation

**Amortisering**: tilbagebetaling af lån



# Bogholdermetoden
(fortsat)

Den enkelte operation må gerne prissættes højere end antallet af primitive operationer tilsiger

De enkelte operationer kan således låne til/fra hinanden, dog ikke mere end der er likviditet til

I praksis betyder det, at nogle operationer forudbetaler for andre

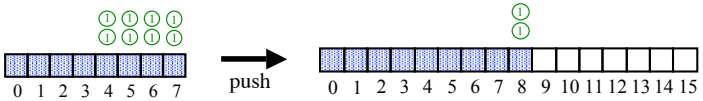
(amortiseret tid) ≤ (**samlet betaling i kroner**) / (# operationer)



# Amortiseringsskema for fordoblingsstrategien

Betragt igen de *k* faser, hvor hver fase indeholder dobbelt så mange push-operationer som den forrige

Ved afslutningen af hver fase skal vi sørge for at have sparet nok op til, at push-kaldene i næste fase kan betales. Når en fase afsluttes, bør vi have opsparet tilstrækkeligt til, at vi kan betale for array-kopieringen i starten af næste fase




Vi tager 3 kroner for en push-operation. De overskydende 2 kroner gemmes i sidste halvdel af arrayet. Ved afslutningen af en fase, hvor stakken indeholder *i* elementer, vil der være gemt $2(i/2) = i$ kroner. Derfor udføres hver push-operation i $O(1)$ amortiseret tid.



# Kø ADT

En **kø** indeholder vilkårlige objekter.

Indsættelse og fjernelse følger først-ind-først-ud princippet.

Centrale operationer:
- **enqueue**(object): indsætter et objekt bagest
- object **dequeue**(): fjerner og returnerer det forreste objekt

Hjælpeoperationer:
- object **front**(), integer **size**(), boolean **isEmpty**()

Undtagelser:
- Forsøg på at udføre dequeue eller front på en tom kø kaster en **EmptyQueueException**

# Anvendelser af køer

Direkte anvendelser
- Ventelister
- Tilgang til delte ressourcer (f.eks. en printer)
- Multiprogrammering

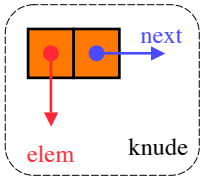
Indirekte anvendelser
- Hjælpe-datastruktur i algoritmer
- Komponent i andre datastrukturer

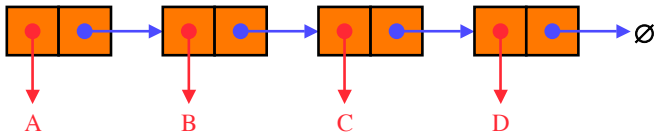


# Enkelthægtet liste

En enkelthægtet liste er en konkret datastruktur, der indeholder en sekvens af knuder

Hver knude lagrer
- et element
- en hægte til den næste knude



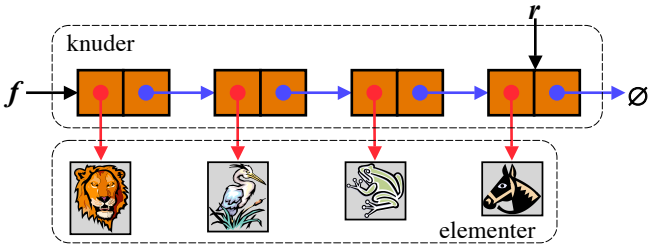






# Kø implementeret med en enkelthægtet liste

Det forreste element lagres i den første knude
Det bageste element lagres i den sidste knude

Pladsforbruget er $O(n)$, og hver operation tager $O(1)$ tid.






# Liste ADT

En liste modellerer en sekvens af **positioner**, der kan lagre vilkårlige objekter

Den tillader indsættelse i og fjernelse fra "midten"

Forespørgelsesmetoder:
- **isFirst**(p), **isLast**(p)

Tilgangsmetoder:
- **first**(), **last**(), **before**(p), **after**(p)

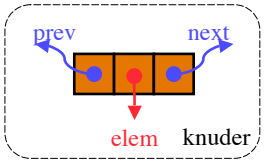
Opdateringsmetoder:
- **replaceElement**(p, o), **swapElements**(p, q)
- **insertBefore**(p, o), **insertAfter**(p, o),
- **insertFirst**(o), **insertLast**(o), **remove**(p)

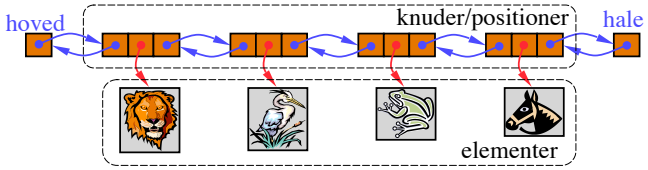
# Dobbelthægtet liste

Knuder implementer positioner og lagrer:

- et element
- en hægte til den foregående knude
- en hægte til den efterfølgende knude

Desuden kan der være specielle hoved- og haleknuder.



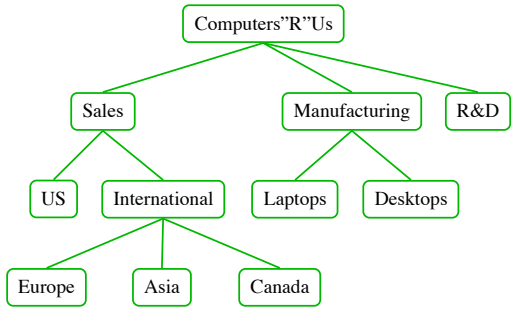



# Træer

I datalogi er et **træ** en abstrakt model af en hierarkisk struktur

Et træ består af knuder med en forældre/barn-relation

Anvendelser:
Organisationsdiagrammer
Filsystemer
Indholdsfortegnelser






# Terminologi for træer

**Rod**: knude uden forælder (A)

**Intern knude**: knude med mindst et barn (A, B, C, F)

**Ekstern knude** (også kaldet et **blad**): knude uden børn (E, I, J, K, G, H, D)

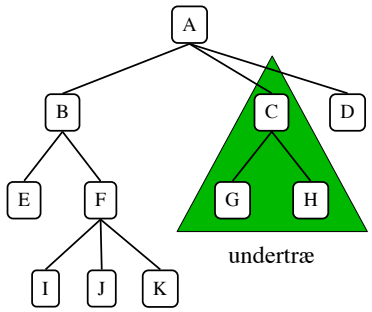
**Forfædre** til en knude: forælder, bedsteforælder, oldeforælder o.s.v.

**Dybden** af en knude: antallet af forfædre

**Højden** af et træ: maksimale dybde for træets knuder (3)

**Efterkommere** af en knude: børn, børnebørn o.s.v.

**Undertræ**: et træ bestående af en knude og dennes efterkommere






# Træ ADT

Positioner benyttes til at abstrahere knuder

Generiske metoder:
- integer **size**()
- boolean **isEmpty**()
- objectIterator **elements**()
- positionIterator **positions**()
- **swapElements**(p, q)
- object **replaceElement**(p, o)

Tilgangsmetoder:
- position **root**()
- position **parent**(p)
- positionIterator **children**(p)

Forespørgselsmetoder:
- boolean **isInternal**(p)
- boolean **isExternal**(p)
- boolean **isRoot**(p)

Opdateringsmetoder:
- defineres ikke her

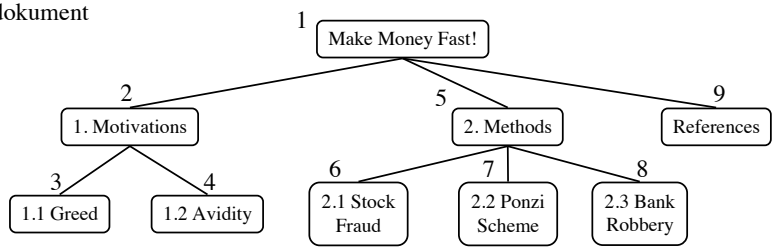
# Preorder-traversering

En traversering besøger knuderne i et træ på en systematisk måde

Ved preorder-traversering besøges en knude før sine efterkommere

*Anvendelse*: udskriv et struktureret dokument

**Algorithm** *preOrder*(*v*)
  *visit*(*v*)
  **for each** child *w* of *v*
    *preOrder*(*w*)



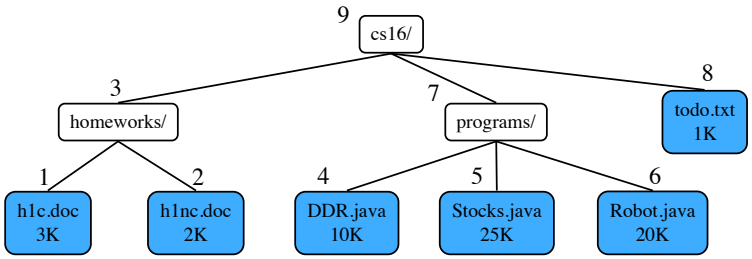



# Postorder-traversering

Ved en postorder-traversering besøges en knude efter sine efterkommere

*Anvendelse*: beregn pladsforbruget for filer i et katalog og dettes underkataloger.

**Algorithm** *postOrder*(*v*)
  **for each** child *w* of *v*
    *postOrder*(*w*)
  *visit*(*v*)






# Amortiseret analyse af træ-traversering

Den tid, der bruges til preorder- og postorder-traversering af et træ med $n$ knuder, er proportional med summen - taget over enhver knude $v$ i træet - af tiden, der anvendes til et rekursivt kald for $v$

Kaldet for $v$ koster $(c_v + 1)$ kroner, hvor $c_v$ er antallet af børn for $v$

Betal 1 krone for $v$, og 1 krone for hver af $v$'s børn. Dermed betales der for enhver knude 2 gange, en gang for dens eget kald, og en gang for dens fars kald

Derfor er traverseringstiden $O(n)$



# Binære Træer

Et **binært træ** er et træ med følgende egenskaber:

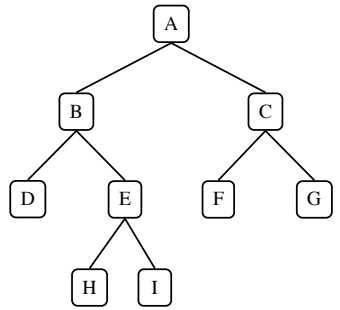
- Hver interne knude har to børn
- Børnene for en knude er et ordnet par
- De to børn kaldes for henholdsvis **venstre** og **højre** barn

Alternativ rekursiv definition:
Et **binært træ** er enten

- et træ bestående af en enkelt knude, eller
- et træ, hvis rod har et ordnet par af børn, som hver er et **binært træ**

Anvendelser:

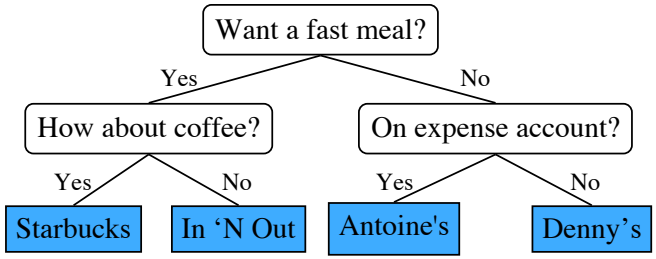
| | |
|---|---|
| Søgning | Beslutningsprocesser |
| | Repræsentation af aritmetiske udtryk |



# Beslutningstræ

Interne knuder: spørgsmål med ja/nej-svar
Eksterne knuder: beslutninger

Eksempel: Valg af spisested

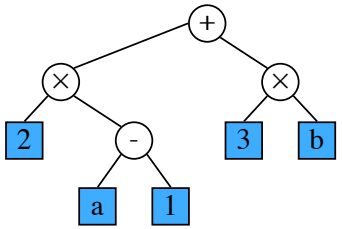





# Aritmetisk udtryktrykstræ

Interne knuder: operatorer
Eksterne knuder: operander

Eksempel: Aritmetisk udtrykstræ for udtrykket $(2 \times (a - 1) + (3 \times b))$

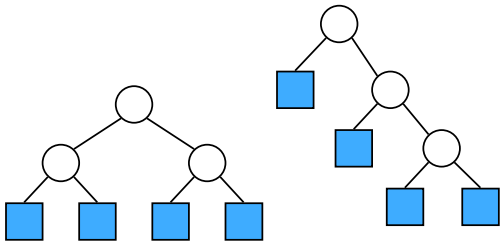





# Egenskaber ved binære træer

Notation

- $n$ antal knuder
- $e$ antal eksterne knuder
- $i$ antal interne knuder
- $h$ højde



Egenskaber:

- $e = i + 1$
- $n = 2e - 1$
- $h \leq i$
- $h \leq (n - 1)/2$
- $e \geq h + 1$
- $e \leq 2^h$
- $h \geq \log_2 e$
- $h \geq \log_2 (n + 1) - 1$

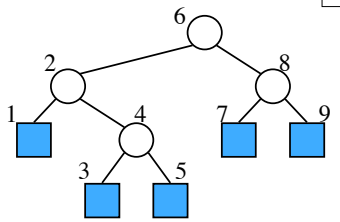


# Inorder-traversering

Ved en inorder-traversering besøges en knude efter sit venstre undertræ, men før sit højre undertræ

*Anvendelse*:
Tegning af et binært træ
x($v$) = inorder-rangen af $v$
y($v$) = dybden af $v$

```
Algorithm inOrder(v)
    if isInternal(v) then
        inOrder(leftChild(v))
    visit(v)
    if isInternal(v) then
        inOrder(rightChild(v))
```



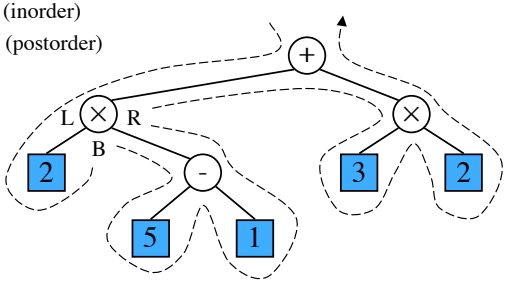
# Euler-tur-traversering

Generisk traversering af et binært træ

Indeholder preorder-, postorder- og inorder-traversering som specialtilfælde

Gå rundt i træet og besøg hver knude 3 gange:

- første besøg, L (preorder)
- andet besøg, B (inorder)
- tredje besøg, R (postorder)

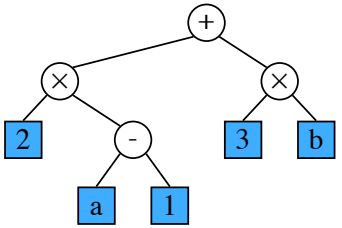





# Udskrivning af aritmetiske udtryk

Specialisering af inorder-traversering.

- Udskriv "(" inden traversering af venstre undertræ
- Udskriv operand eller operator, når en knude besøges
- Udskriv ")" efter traversering af højre undertræ




**Algorithm** *printExpression*(*v*)
  **if** *isInternal*(*v*) **then**
    *print*("(")
    *printExpression*(*leftChild*(*v*))
  *print*(*v.element* ())
  **if** *isInternal*(*v*) **then**
    *printExpression*(*rightChild*(*v*))
    *print*(")")

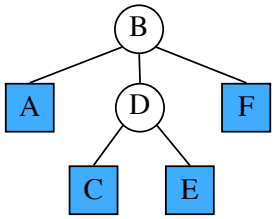
((2 × (a - 1)) + (3 × b))

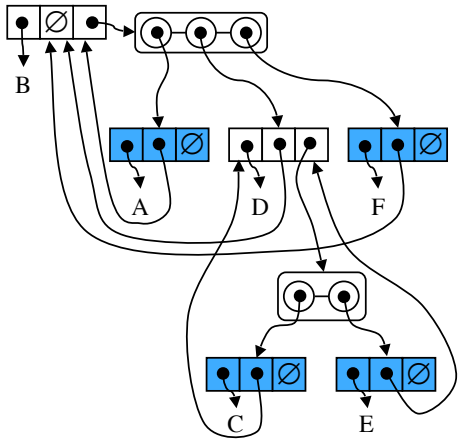


# Hægtet datastruktur til repræsentation af træer

En knude repræsenteres ved et objekt, der indeholder

- Element
- Hægte til forældreknude
- Sekvens af børneknuder



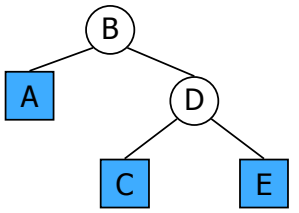



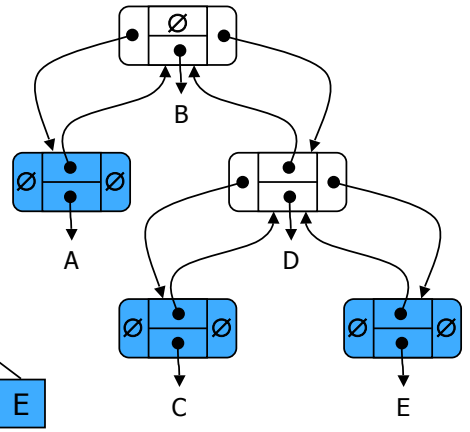



# Hægtet datastruktur til repræsentation af binære træer

En knude repræsenteres ved et objekt, der indeholder

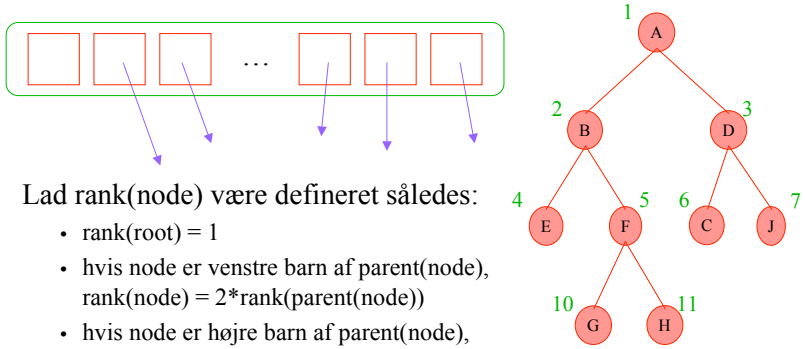
- Element
- Hægte til forældreknude
- Hægte til venstre barn
- Hægte til højre barn



# Array-baseret repræsentation af binære træer

Knuder lagres i et array




Lad rank(node) være defineret således:

- rank(root) = 1
- hvis node er venstre barn af parent(node), rank(node) = 2*rank(parent(node))
- hvis node er højre barn af parent(node), rank(node) = 2*rank(parent(node)) + 1



# Prioritetskø ADT

En **prioritetskø** lagrer en samling af emner
Hvert emne er et par
(nøgle, element)

Centrale metoder:
**insertItem**(k, o)
indsætter et emne med nøglen k og elementet o
**removeMin**()
fjerner det emne, der har den mindste nøgle og returnerer det tilhørende element

Yderligere metoder:
**minKey**()
returnerer, men fjerner ikke, den mindste nøgle
**minElement**()
returnerer, men fjerner ikke, det element, der har mindst nøgle
**size**(), **isEmpty**()

Anvendelser:
Opgaveplanlægning
Grafalgoritmer



# Total ordningsrelation

Nøglerne i en prioritetskø kan være vilkårlige objekter, for hvilke der er defineret en total ordningsrelation

To forskellige emner i en prioritetskø kan have samme nøgle

**Total ordningsrelation** ≤
Refleksivitet:
$x \leq x$
Antisymmetri:
$x \leq y \wedge y \leq x \Rightarrow x = y$
Transitivitet:
$x \leq y \wedge y \leq z \Rightarrow x \leq z$



# Sortering med en prioritetskø

Vi kan benytte en prioritetskø til at sortere en mængde af sammenlignelige objekter

Indsæt elementerne et efter et med en serie af insertItem(e, e)-operationer

Udtag elementerne i sorteret orden med en serie af removeMin()-operationer

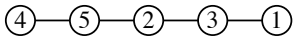
Køretiden for denne sorteringsmetode afhænger af implementeringen af prioritetskøen

**Algorithm** *PQ-Sort*(*S*, *C*)
**Input** sequence *S*, comparator *C* for the elements of *S*
**Output** sequence *S* sorted in increasing order according to *C*
*P* ← empty priority queue with comparator *C*
**while** ¬*S.isEmpty*() **do**
*e* ← *S.remove*(*S. first*())
*P.insertItem*(*e*, *e*)
**while** ¬*P.isEmpty*() **do**
*e* ← *P.removeMin*()
*S.insertLast*(*e*)

# Sekvens-baseret prioritetskø

Implementering med en **usorteret** sekvens:

Gem emnerne i en liste-baseret sekvens i vilkårlig rækkefølge



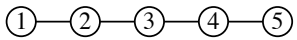

Effektivitet:

**insertItem** tager $O(1)$ tid, da vi kan indsætte emnet i starten eller slutningen af listen

**removeMin**, **minKey** og **minElement** tager $O(n)$ tid, da vi skal gennemløbe hele sekvensen for at finde den mindste nøgle

Implementering med en **sorteret** sekvens:

Gem emnerne i en liste-baseret sekvens i sorteret rækkefølge




Effektivitet:

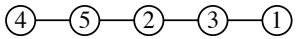
**insertItem** tager $O(n)$ tid, da vi skal finde den plads, hvor emnet skal indsættes

**removeMin**, **minKey** og **minElement** tager $O(1)$ tid, da den mindste nøgle er forrest i listen



# Sortering ved udvælgelse

Sortering ved **udvælgelse** er den udgave af PQ-sort, hvor prioritetskøen er implementeret ved brug af en usorteret sekvens.




Køretid:

Indsættelse af elementerne i prioritetskøen med $n$ **insertItem**-operationer tager $O(n)$ tid

Fjernelse af elementerne i sorteret rækkefølge fra prioritetskøen med $n$ **removeMin**-operationer tager tid proportional med

$$1 + 2 + \ldots + n$$

Metoden kører i $O(n^2)$ tid



# Sortering ved indsættelse

Sortering ved **indsættelse** er den udgave af PQ-sort, hvor prioritetskøen er implementeret ved brug af en sorteret sekvens


Køretid:

Indsættelse af elementerne i prioritetskøen med $n$ **insertItem**-operationer tager tid proportional med

$$1 + 2 + \ldots + n$$

Fjernelse af elementerne i sorteret rækkefølge fra prioritetskøen med $n$ **removeMin**-operationer tager $O(n)$ tid

Metoden kører i $O(n^2)$ tid

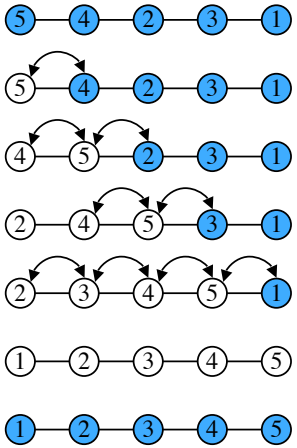


# Sortering ved indsættelse på stedet (in-place)

I stedet for at bruge en ekstern datastruktur, kan vi implementere både sortering ved udvælgelse og sortering ved indsættelse "på stedet"

En del af input-sekvensen tjener som prioritetskø

For sortering ved indsættelse holdes den første del af sekvensen sorteret, og **swapElements**-operationen benyttes til at udvide denne del af sekvensen



# Hvad er en hob?

En **hob** er et binært træ, der lagrer nøglerne i de interne knuder, og som opfylder følgende 2 betingelser:

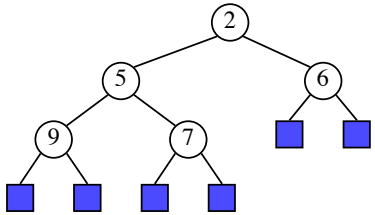
**Hob-orden**: For enhver intern knude, $v$, med undtagelse af roden, gælder

$$key(v) \geq key(parent(v))$$

**Komplet binært træ**: Lad $h$ være højden af hoben

(1) for $i = 0, \ldots, h-1$ er der $2^i$ knuder med dybde $i$

(2) på dybden $h-1$ er de interne knuder til venstre for de eksterne knuder






# Højden af en hob

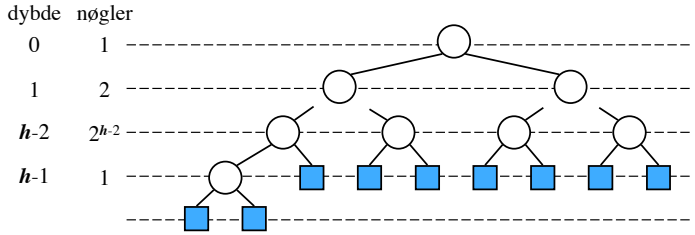
**Sætning**: En hob, der indeholder $n$ nøgler har højden $O(\log n)$

**Bevis**: (vi anvender egenskaben for et komplet binært træ)

Lad $h$ være højden af en hob, der indeholder $n$ nøgler

Da der er $2^i$ nøgler på dybden $i = 0, \ldots, h-2$, og mindst en nøgle på dybden $h-1$, har vi, at $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1$

Og dermed $n \geq 2^{h-1}$, d.v.s. $h \leq \log n + 1$



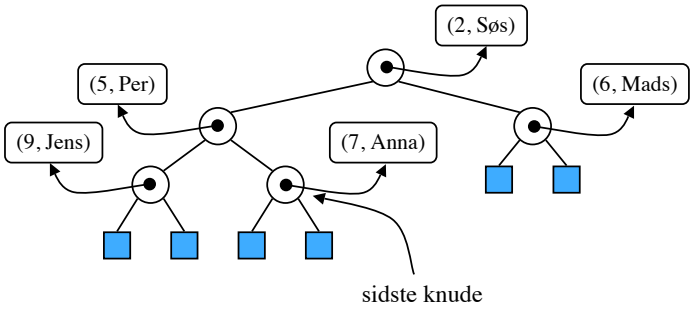



# Hobe og prioritetskøer

Vi kan bruge en hob til at implementere en prioritetskø

I hver intern knude gemmes et (nøgle, element)-par

Der holdes rede på den sidste knude





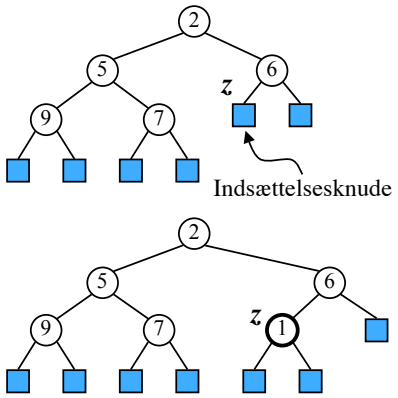

# Indsættelse i en hob

Indsættelses-algoritmen består at af 3 trin:

(1) Find indsættelsesknuden $z$ (den nye sidste knude)

(2) Gem $k$ i $z$ og omdan $z$ til en intern knude

(3) Genskab hob-ordens-egenskaben

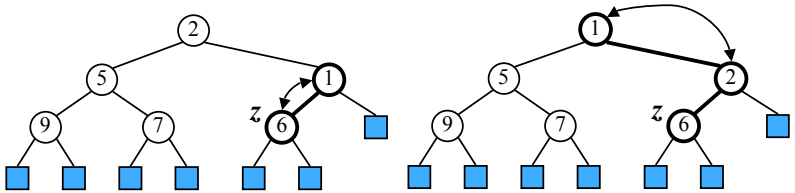


# upheap

Efter indsættelse af en nøgle ***k*** kan hob-ordens-egenskaben blive ødelagt

Algoritmen **upheap** genskaber egenskaben ved at ombytte ***k*** på en opadgående vej fra indsættelsesknuden. Algoritmen terminerer, når ***k*** når op til roden eller til en knude, hvis forældre har en nøgle, der er mindre end eller lig med ***k***

Da en hob har højden $O(\log n)$, kører **upheap** i $O(\log n)$ tid





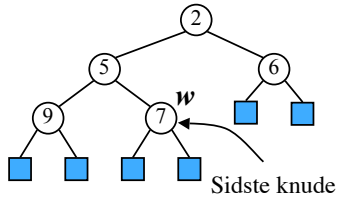

# Fjernelse fra en hob

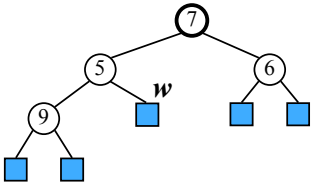
Fjernelses-algoritmen består at af 3 trin:

(1) Erstat roden med nøglen i den sidste knude, ***w***

(2) Omdan ***w*** og dens børn til en ekstern knude

(3) Genskab hob-ordens-egenskaben



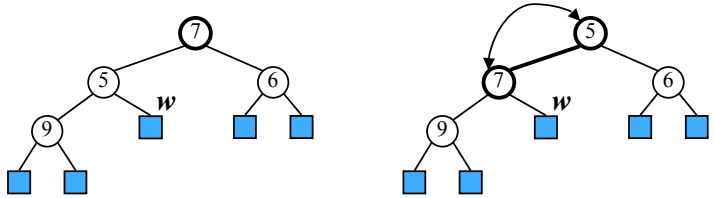



# downheap

Efter erstatning af rodens nøgle med ***k*** for den sidste knude, kan hob-ordens-egenskaben være ødelagt.

Algoritmen **downheap** genskaber egenskaben ved at ombytte ***k*** på en neadadgående vej fra roden. Algoritmen terminerer, når ***k*** når ned til en ekstern knude eller til en knude, hvis børn har nøgler, der er større end eller lig med ***k***.

Da en hob har højden $O(\log n)$, kører **downheap** i $O(\log n)$ tid.






# Heap-Sort

Betragt en prioritetskø med ***n*** emner, der er implementeret ved hælp af en hob

Pladsforbruget er $O(n)$

Metoderne **insertItem** og **removeMin** tager $O(\log n)$ tid

Metoderne **size**, **isEmpty**, **minKey** og **minElement** tager $O(1)$ tid

Ved at bruge en hob-baseret prioritetskø kan vi sortere en sekvens i $O(n \log n)$ tid

Denne algoritme, der kaldes Heap-sort, er meget hurtigere end kvadratiske sorteringsalgoritmer, som f.eks. sortering ved udvælgelse og sortering ved indsættelse

# Array-baseret hob

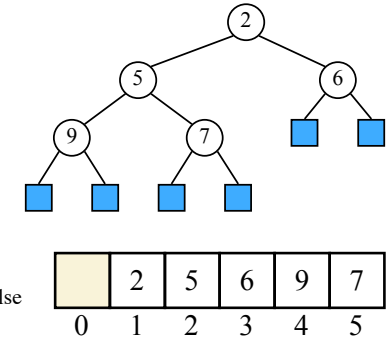
Vi kan repræsentere en hob med $n$ nøgler ved hjælp af et array af længden $n + 1$

For knuden på indeks $i$ er
det venstre barn på indeks $2i$
det højre barn på indeks $2i + 1$

Hægterne imellem knuderne lagres ikke
De eksterne knuder er ikke repræsenteret
Cellen på indeks 0 bruges ikke

Operationen **insertItem** svarer til indsættelse på indeks $n + 1$

Operationen **removeMin** svarer til fjernelse på indeks 1



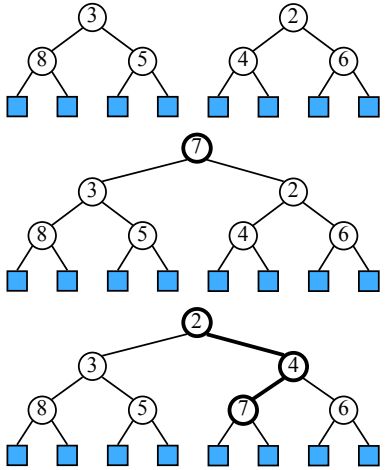



# Fletning af to hobe

Lad der være givet to hobe og en nøgle $k$

Vi kan skabe et nyt træ, hvor roden indeholder $k$ og har de to hobe som undertræer

Hob-ordenen retableres ved at udføre **downheap**

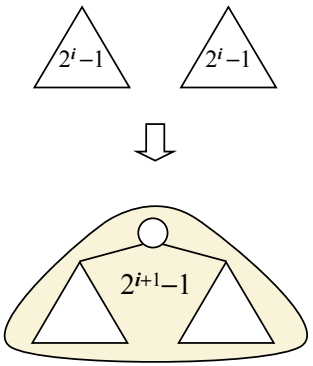





# Bund-til-top hob-konstruktion

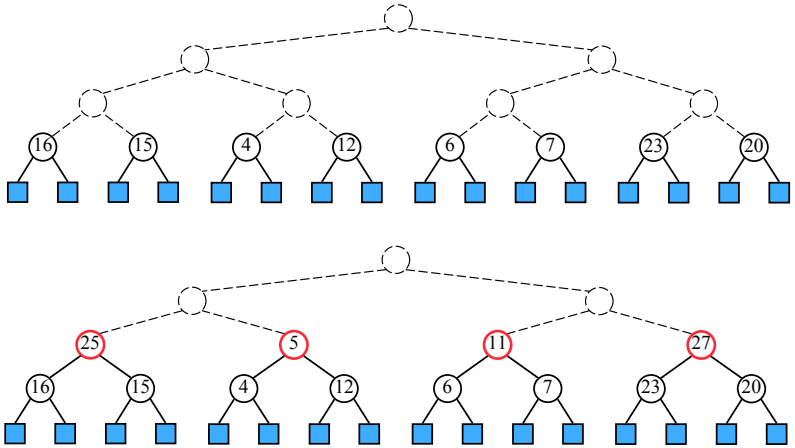
Vi kan konstruere en hob med $n$ givne nøgler i log $n$ faser ved at benytte bund-til-top hob-konstruktion

I fase $i$ flettes hobe med $2^i - 1$ nøgler til hob med $2^{i+1} - 1$ nøgler

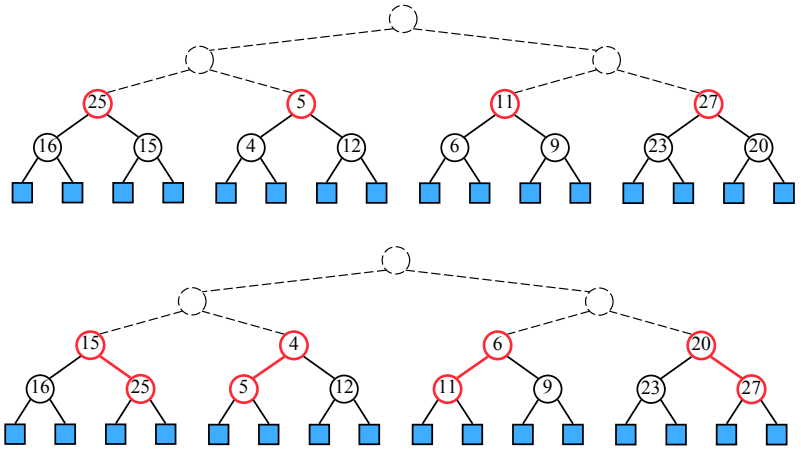





# Eksempel

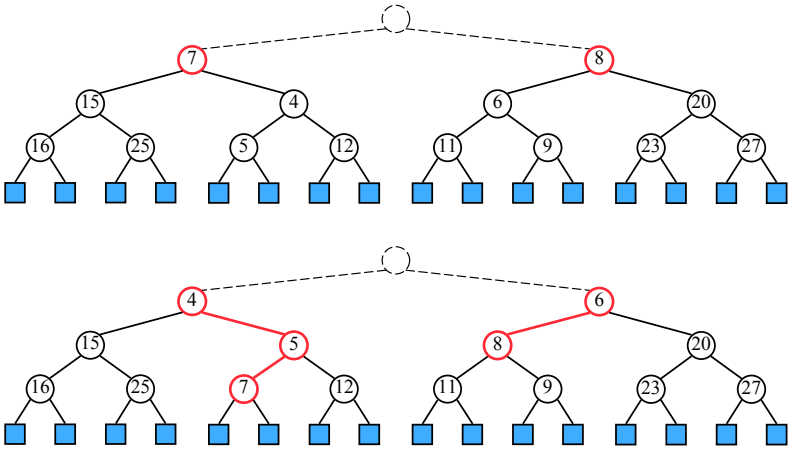


## Eksempel (fortsat)



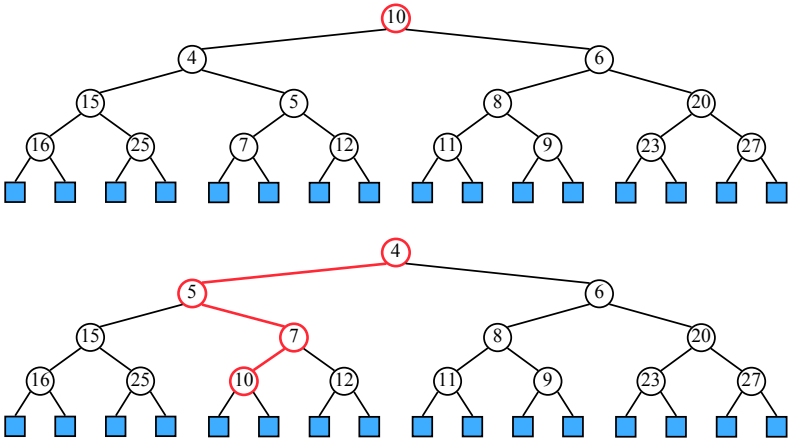



## Eksempel (fortsat)






## Eksempel (slut)



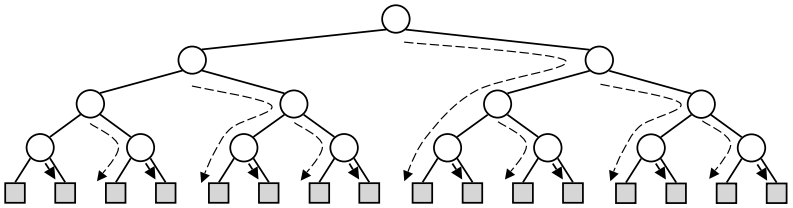



## Analyse

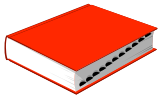
Vi visualiserer det værste tilfælde for en downheap-operation med veje, der først går til højre og derefter gentagne gange til venstre, indtil bunden af hoben nås

Da en knude højst gennemløbes af 2 sådanne veje, er antallet af knuder på disse veje $O(n)$

Derfor udføres bund-til-top hob-konstruktion i $O(n)$ tid



# Ordbog ADT (Dictionary)



En **ordbog** lagrer en samling af emner, der kan søges i

Hvert emne er et par (nøgle, element)

De centrale operationer for en ordbog er søgning, indsættelse og fjernelse af emner

Anvendelser:

- Adressebog
- Kontokort-autorisering
- Afbildning fra værtsnavne (f.eks. akira.ruc.dk) til internetadresser (f.eks. 130.225.220.8)

Metoder:

**findElement**(k):
Hvis ordbogen indeholder et emne med nøgle k, så returneres dets element. Ellers returneres det specielle element NO_SUCH_KEY

**insertItem**(k, o):
Indsætter emnet (k, o) i ordbogen

**removeElement**(k):
Hvis ordbogen har et emne med nøglen k, så fjernes det, og dets element returneres. Ellers returneres det specielle element NO_SUCH_KEY

**size**(), **isEmpty**()

**keys**(), **elements**()



# Logfil

En **logfil** er en ordbog, der er implementeret ved hjælp af en usorteret sekvens

Emnerne lagres i vilkårlig orden i en sekvens (baseret på en dobbelt-hægtet liste eller et cirkulært array)

Effektivitet:

**insertItem** tager $O(1)$ tid, da vi kan indsætte det nye emne forrest eller bagest i sekvensen

**findElement** og **removeElement** tager $O(n)$ tid, da vi i værste tilfælde skal gennemløbe hele sekvensen for at lede efter et emne med den givne nøgle

En logfil er kun effektiv for små ordbøger, eller for hvilke indsættelse er den mest almindelige operation, mens søgning og sletning er sjældne operationer (f.eks. registrering af login)



# Hashfunktioner og hashtabeller

En **hashfunktion** afbilder nøgler af en given type til heltal i et fast interval $[0, N - 1]$

Eksempel:

$h(x) = x \bmod N$

er en hashfunktion for heltalsnøgler

Heltallet $h(x)$ kaldes for **hashværdien** for nøglen $x$

En **hashtabel** for en given nøgletype består af
- En hashfunktion $h$
- Et array (kaldet tabellen) af størrelse $N$

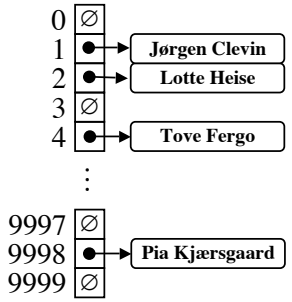
Når en ordbog implementeres ved hjælp af en hashtabel, er målet at lagre emnet $(k, o)$ på indeks $i = h(k)$



# Eksempel

En hashtabel for en ordbog, der indeholder emner (cpr, navn), hvor cpr er et 10-cifret positivt heltal.

Hashtabellen benytter et array af størrelse $N = 10000$ og hashfunktionen $h(x)$ = de sidste 4 cifre i $x$



# Hashfunktioner

En hashfunktion specificeres sædvanligvis som sammensætningen af to funktioner:

**Hashkode-afbildningen**:
$h_1$: nøgler→ heltal

**Komprimerings-afbildningen**:
$h_2$: heltal → $[0, N-1]$

Hashkode-afbildningen anvendes først. Derefter anvendes komprimerings-afbildningen på resultatet

$$h(x) = h_2(h_1(x))$$

Målet for hashfunktionen er at "sprede" nøglerne på en tilsyneladende tilfældig måde



# Hashkode-afbildninger

**Lageradresse**:
Vi fortolker lageradressen for nøgleobjektet som et heltal (standard hashkode for alle Java-objekter)

**Heltalskonvertering**:
Vi fortolker bittene i nøglen som et heltal

Hensigtsmæssig for nøgler med en længde der er mindre end eller lig med antallet af bit i en heltalstype

**Komponentsum**:
Vi opdeler bittene i nøglen i komponenter af fast længde (f.eks. 8, 16 eller 32 bit), og vi summerer komponenterne (idet overløb ignoreres)

Hensigtsmæssig for nøgler med en længde, der er større antallet af bit i en heltalstype

Dog uhensigtsmæssig for tegnstrenge, hvor tegnenes rækkefølge har betydning ("post", "stop" og "spot" kolliderer)



# Hashkode-afbildninger (fortsat)

**Polynomiel akkumulation**:
Vi opdeler bittene i nøglen i komponenter af fast længde (f.eks. 8, 16 eller 32 bit)

$a_0 a_1 \ldots a_{n-1}$

Vi evaluerer polynomiet

$$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$$

for en fast værdi $z$, idet overløb ignoreres

Er særlig hensigtsmæssig for tekststrenge (for eksempel giver valget $z = 33$ kun 6 kollisioner for en mængde af 50000 engelske ord)

Værdien $p(z)$ kan evalueres i $O(n)$ tid ved hjælp af **Horner's regel**:

Følgende værdier beregnes successivt, hver ud fra den foregående værdi i $O(1)$ tid:

$$p_0(z) = a_{n-1}$$
$$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \quad (i = 1, 2, \ldots, n-1)$$

Vi har, at $p(z) = p_{n-1}(z)$

$$p(z) = a_0 + z(a_1 + z(a_2 + \ldots + z\,a_{n-1})))))$$



# Komprimerings-afbildninger

**Division**:

$$h_2(y) = y \bmod N$$

Størrelsen $N$ af hashtabellen vælges sædvanligvis som et primtal

**Multiply, Add and Divide** (MAD):

$$h_2(y) = (ay + b) \bmod N$$

hvor $a$ og $b$ er ikke-negative heltal, således at

$a \bmod N \neq 0$

(ellers ville ethvert heltal $y$ blive afbildet til den samme værdi $b$)

# Kollisioner

**Kollisioner** optræder, når forskellige nøgler afbildes til samme celle

Kollisioner er hyppige:

Hvor mange personer skal være forsamlet i et selskab, for at der er mere end 50% sandsynlighed for, at mindst to personer har fødselsdag på samme dato?

Svar: 24.

Lad $N$ være størrelsen af hashtabellen. Hvor mange indsættelser kan der da i gennemsnit foretages, før der opstår en kollision?

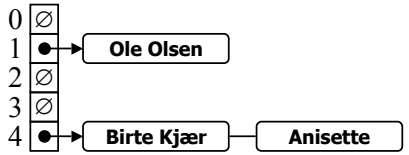
| $N$ | $\sqrt{\pi N/2}$ |
|---|---|
| 100 | 12 |
| 365 | 24 |
| 1000 | 40 |
| 10000 | 125 |
| 100000 | 396 |
| 1000000 | 2353 |



# Separat kædning

**Separat kædning**: Lad hver celle i tabellen pege på en hægtet liste af de elementer, der afbildes til cellen

Separat kædning er simpel, men kræver ekstra lagerplads, udover tabellen






# Lineær afprøvning

**Åben adressering**: Et kolliderende emne placeres i en anden celle i tabellen

**Linær afprøvning** håndterer kollisioner ved at placere det kolliderende emne i den næste (cirkulært) ledige celle

Hver celle, der undersøges, kaldes en "afprøvning"

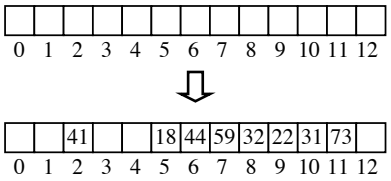
Kolliderende emner klumper sammen, hvilket bevirker, at fremtidige kollisioner skal bruge længere sekvenser af afprøvninger

Eksempel:

$h(k) = k \bmod 13$

Indsæt nøglerne 18, 41, 22, 44, 59, 32, 31, 73 i nævnte rækkefølge

| $k$ | $h(k)$ |
|---|---|
| 18 | 5 |
| 41 | 2 |
| 22 | 9 |
| 44 | 5 |
| 59 | 7 |
| 32 | 6 |
| 31 | 5 |
| 73 | 8 |






# Søgning med lineær afprøvning

Betragt en hashtabel $A$, der benytter linær afprøvning

**findElement**($k$)

Start i celle $h(k)$

Prøv i konsekutive celler, indtil en af følgende hændelser indtræffer

- En tom celle er fundet
- Et emne med nøglen $k$ er fundet
- $N$ celler er blevet prøvet

```
Algorithm findElement(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅ then
            return NO_SUCH_KEY
        else if c.key() = k then
            return c.element()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until p = N
    return NO_SUCH_KEY
```

# Opdatering med lineær afprøvning

For at håndtere fjernelse introduceres et specielt objekt, kaldet ***AVAILABLE***, som erstatter fjernede elementer

**removeElement**(***k***)

Søg efter et emne med nøgle ***k***

Hvis et sådan emne (***k, o***) findes, så erstat det med det specielle emne ***AVAILABLE*** og returner elementet ***o***

Ellers returneres ***NO_SUCH_KEY***

**insertItem**(***k, o***)

Kast en undtagelse, hvis tabellen er fuld

Start i celle ***h***(***k***)

Prøv i konsekutive celler, indtil en af følgende hændelser indtræffer

- Der mødes en celle ***i***, som er tom eller indeholder ***AVAILABLE***
- ***N*** celler er blevet prøvet

Gem emnet (***k, o***) in celle ***i***



# Argumentation for tendens til klyngedannelse

Antag, at alle positioner [*i*:*j*] indeholder poster, mens *i*-1, *j*+1 og *j*+2 er tomme


Så vil chancen for, at en ny post placeres på position *j*+1 være lig med chancen for, at en ny post ønskes placeret i intervallet [*i*:*j*+1]

For at den nye post placeres på *j*+2, skal dens hashværdi derimod være præcis *j*+2



# Klyngedannelse

Uheldigt fænomen

Lange klynger har en tendens til at blive længere

Søgelængen vokser drastisk, efterhånden som tabellen fyldes

Lineær afprøvning er for langsom, når tabellen bliver 70-80% fuld



# Kvadratisk afprøvning

(reducerer risikoen for klyngedannelse)

Afprøvningssekvens (startende i $pos = h(k)$):

Lineær prøvning: $pos, pos + 1, pos + 2, pos + 3, ...$

**Kvadratisk prøvning**: $pos, pos + 1^2, pos + 2^2, pos + 3^2, ...$

Kvadrering kan undgås

Lad $H_i$ betegne den $i$'te position ($H_0 = pos$ er startpositionen)

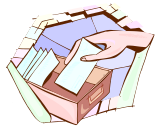
Idet

$$H_{i-1} = pos + (i - 1)^2 =$$
$$pos + i^2 - 2i + 1 = H_i - 2i + 1$$

fås

$$H_i = H_{i-1} + 2i - 1$$

# Dobbelt hashing



Dobbelt hashing benytter en sekundær hashfunktion $d(k)$ og håndterer kollisioner ved at placere et emne på den første ledige celle i rækken

$$(i + jd(k)) \bmod N$$

for $j = 0, 1, \ldots, N-1$

Den sekundære hashfunktion $d(k)$ må ikke have 0-værdier

Tabelstørrelsen skal være et primtal for at muliggøre afprøvning i alle celler

Sædvanligt valg af komprimerings-afbildning for den sekundære hashfunktion:

$$d(k) = q - (k \bmod q)$$

hvor

$q < N$, og $q$ er et primtal

De mulige værdier for $d(k)$ er $1, 2, \ldots, q$



# Eksempel på dobbelt hashing

Betragt en hashtabel af heltallige nøgler, som håndterer kollision ved dobbelt hashing

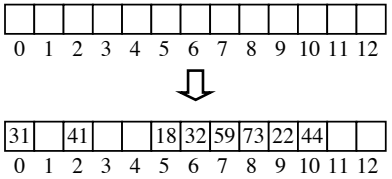
$N = 13$

$h(k) = k \bmod 13$

$d(k) = 7 - (k \bmod 7)$

Indsæt nøglerne 18, 41, 22, 44, 59, 32, 31, 73 i nævnte rækkefølge

| $k$ | $h(k)$ | $d(k)$ | prøvning | | |
|---|---|---|---|---|---|
| 18 | 5 | 3 | 5 | | |
| 41 | 2 | 1 | 2 | | |
| 22 | 9 | 6 | 9 | | |
| 44 | 5 | 5 | 5 | 10 | |
| 59 | 7 | 4 | 7 | | |
| 32 | 6 | 3 | 6 | | |
| 31 | 5 | 4 | 5 | 9 | 0 |
| 73 | 8 | 4 | 8 | | |



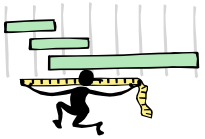



# Fordele ved separat kædning

- Idiotsikker metode (bryder ikke sammen)
- Antallet af poster behøver ikke at være kendt på forhånd
- Sletning er simpel
- Tillader ens nøgler



# Effektiviteten af hashing



I værste tilfælde udføres søgning, indsættelse og fjernelse i $O(n)$ tid. Det værste tilfælde optræder, når alle nøgler, der indsættes i hashtabellen, kolliderer

**Fyldningsgraden** $\alpha = n/N$ har betydning for effektiviteten af en hashtabel

Antag at hashværdierne er som tilfældige tal. Det kan da vises, at det forventede antal afprøvninger for indsættelse i en hashtabel med lineær afprøvning er

$$1 / (1 - \alpha)$$

Den forventede køretid for samtlige hashtabel-operationer er $O(1)$

I praksis er hashing meget hurtig, forudsat at fyldningsgraden ikke er tæt på 100%

Anvendelser:

- Databaser
- Oversættere
- Cache-lagre i browsere

# Universel hashing

En familie af hashfunktioner er **universel**, hvis det for ethvert $0 \le j,k \le M$-1 gælder, at

$$\Pr(h(j)=h(k)) \le 1/N$$

Vælg et primtal $p$ imellem $M$ og $2M$ (et sådan findes altid)

Vælg tilfældigt $0 < a < p$ og $0 < b < p$, og definer

$$h(k) = (ak + b \bmod p) \bmod N$$

**Sætning**:
Mængden af alle funktioner, $h$, som er defineret på denne måde, er universelle

**Sætning**:
Sandsynligheden for kollision ved brug en universel hashfunktion er mindre end eller lig med fyldningsgraden



# Perfekt hashing

Universel hashing har $O(1)$ **gennemsnitlig** effektivitet for ethvert sæt af nøgler

Perfekt hashing garanterer $O(1)$ **værst-tilfælde**-effektivitet for et **statisk** sæt af nøgler

Eksempler på statiske nøglesæt:

- reserverede ord i et programmeringssprog
- filnavnene på en CD-ROM.

*Ide*: Brug en 2-niveau-struktur med universel hashing på hvert niveau

*Niveau 1*: Brug hashing med kædning. De $n$ nøgler hashes til tabellen $T$ ved hjælp af en universel hashfunktion $h$

*Niveau 2*: Hver plads $j$ i $T$ peger på en hashtabel $S_j$ med en tilknyttet universel hashfunktion $h_j$, og med en størrelse, der er kvadratet på antallet af nøgler i $S_j$.



# Grunde til ikke at bruge hashing

Hvorfor bruge andre metoder?

- Der er ingen effektivitetsgaranti
- Hvis nøglerne er lange, kan hashfunktionen være for kostbar at beregne
- Bruger ekstra plads
- Understøtter ikke sortering