

Merging og Hashing

Tilgang til data

To udbredte metoder for at tilgå data:

- ▶ Sekventiel tilgang
- ▶ Random access: tilgang via ID (også kaldet key, nøgle) for dataelementer.

API for sekventiel tilgang

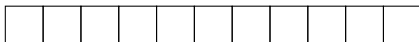
(API = Application Programming Interface: samling af metoder).

API for sekventiel tilgang

(API = Application Programming Interface: samling af metoder).

Læsning:

Operationer: `readNext()`, `isEndOfFile()`, `open()`, `close()`



API for sekventiel tilgang

(API = Application Programming Interface: samling af metoder).

Læsning:

Operationer: `readNext()`, `isEndOfFile()`, `open()`, `close()`



API for sekventiel tilgang

(API = Application Programming Interface: samling af metoder).

Læsning:

Operationer: `readNext()`, `isEndOfFile()`, `open()`, `close()`



API for sekventiel tilgang

(API = Application Programming Interface: samling af metoder).

Læsning:

Operationer: `readNext()`, `isEndOfFile()`, `open()`, `close()`



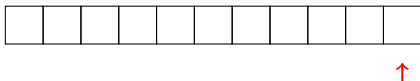
↑ ...

API for sekventiel tilgang

(API = Application Programming Interface: samling af metoder).

Læsning:

Operationer: `readNext()`, `isEndOfFile()`, `open()`, `close()`



API for sekventiel tilgang

(API = Application Programming Interface: samling af metoder).

Læsning:

Operationer: `readNext()`, `isEndOfFile()`, `open()`, `close()`

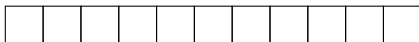


API for sekventiel tilgang

(API = Application Programming Interface: samling af metoder).

Læsning:

Operationer: `readNext()`, `isEndOfFile()`, `open()`, `close()`



Skrivning:

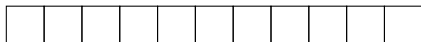
Operationer: `writeNext()`, `open()`, `close()`

API for sekventiel tilgang

(API = Application Programming Interface: samling af metoder).

Læsning:

Operationer: `readNext()`, `isEndOfFile()`, `open()`, `close()`



Skrivning:

Operationer: `writeNext()`, `open()`, `close()`

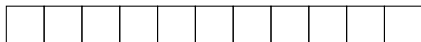


API for sekventiel tilgang

(API = Application Programming Interface: samling af metoder).

Læsning:

Operationer: `readNext()`, `isEndOfFile()`, `open()`, `close()`



Skrivning:

Operationer: `writeNext()`, `open()`, `close()`

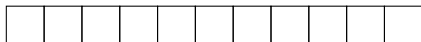


API for sekventiel tilgang

(API = Application Programming Interface: samling af metoder).

Læsning:

Operationer: `readNext()`, `isEndOfFile()`, `open()`, `close()`



Skrivning:

Operationer: `writeNext()`, `open()`, `close()`

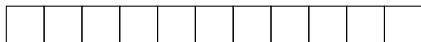


API for sekventiel tilgang

(API = Application Programming Interface: samling af metoder).

Læsning:

Operationer: `readNext()`, `isEndOfFile()`, `open()`, `close()`



Skrivning:

Operationer: `writeNext()`, `open()`, `close()`

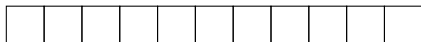


API for sekventiel tilgang

(API = Application Programming Interface: samling af metoder).

Læsning:

Operationer: `readNext()`, `isEndOfFile()`, `open()`, `close()`



Skrivning:

Operationer: `writeNext()`, `open()`, `close()`



API for random access tilgang

Tilgang via ID (også kaldet key, nøgle) for elementer.

API for random access tilgang

Tilgang via ID (også kaldet key, nøgle) for elementer.

Operationer:

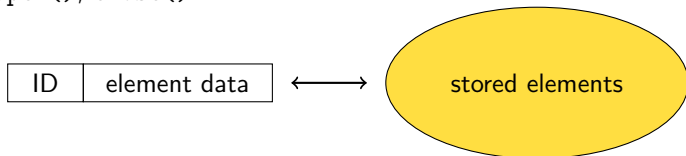
```
findElm(ID),  
insertElm(ID,elementData),  
deleteElm(ID),  
open(), close()
```

API for random access tilgang

Tilgang via ID (også kaldet key, nøgle) for elementer.

Operationer:

```
findElm(ID),  
insertElm(ID,elementData),  
deleteElm(ID),  
open(), close()
```

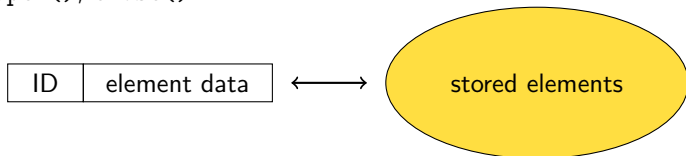


API for random access tilgang

Tilgang via ID (også kaldet key, nøgle) for elementer.

Operationer:

```
findElm(ID),  
insertElm(ID,elementData),  
deleteElm(ID),  
open(), close()
```



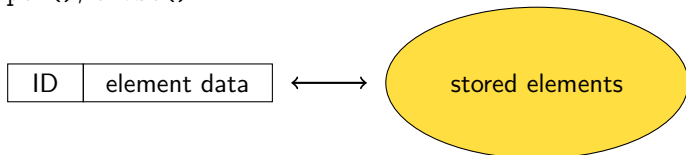
Jvf.:

API for random access tilgang

Tilgang via ID (også kaldet key, nøgle) for elementer.

Operationer:

```
findElm(ID),  
insertElm(ID,elementData),  
deleteElm(ID),  
open(), close()
```



Jvf.:

Dictionary i Python

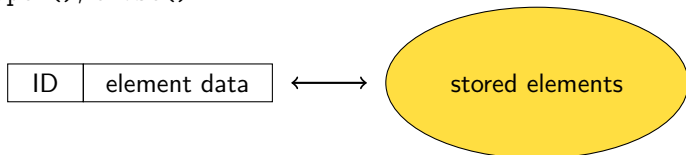
```
(data = dict[ID], dict[ID] = data, del dict[ID])
```

API for random access tilgang

Tilgang via ID (også kaldet key, nøgle) for elementer.

Operationer:

```
findElm(ID),  
insertElm(ID,elementData),  
deleteElm(ID),  
open(), close()
```



Jvf.:

Dictionary i Python

```
(data = dict[ID], dict[ID] = data, del dict[ID])
```

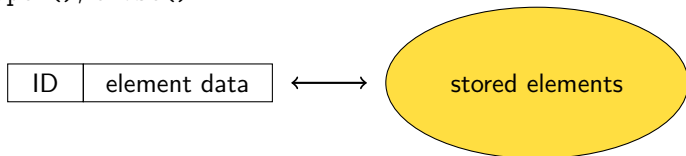
Databases (ID = CPR, f.eks.)

API for random access tilgang

Tilgang via ID (også kaldet key, nøgle) for elementer.

Operationer:

```
findElm(ID),  
insertElm(ID,elementData),  
deleteElm(ID),  
open(), close()
```



Jvf.:

Dictionary i Python

```
(data = dict[ID], dict[ID] = data, del dict[ID])
```

Databases (ID = CPR, f.eks.)

NB: tilgang til liste i Python/arrays i Java kan ses som specialtilfælde hvor ID = index (`data = list[7]`, `list[7] = data`, `list[7] = None`)

Spørgsmål

1. Hvad kan det simple API [Sekventiel tilgang](#) bruges til?

Spørgsmål

1. Hvad kan det simple API **Sekventiel tilgang** bruges til?
2. Hvordan implementeres det mere avancerede API **Random access tilgang**?

Spørgsmål

1. Hvad kan det simple API **Sekventiel tilgang** bruges til?
2. Hvordan implementeres det mere avancerede API **Random access tilgang**?

Bemærk: Alle data kilder kan tilgås med Sekventiel tilgang API (og nogle tillader *kun* det):

- ▶ Harddisk
- ▶ CD
- ▶ Bånd
- ▶ Streaming over net
- ▶ Data genereret on-the-fly af et andet program
- ▶ Data i et array (liste i Python)

Hvad kan laves med sekventiel tilgang?

Hvad kan laves med sekventiel tilgang?

- ▶ Lineær søgning

Hvad kan laves med sekventiel tilgang?

- ▶ Lineær søgning
- ▶ Find største element

Hvad kan laves med sekventiel tilgang?

- ▶ Lineær søgning
- ▶ Find største element
- ▶ Find sum og antal af elementer (og dermed gennemsnit)

Hvad kan laves med sekventiel tilgang?

- ▶ Lineær søgning
- ▶ Find største element
- ▶ Find sum og antal af elementer (og dermed gennemsnit)
- ▶ Selectionsort

Hvad kan laves med sekventiel tilgang?

- ▶ Lineær søgning
- ▶ Find største element
- ▶ Find sum og antal af elementer (og dermed gennemsnit)
- ▶ Selectionsort
- ▶ Merge af to sorterede lister til een (og dermed Mergesort)

Hvad kan laves med sekventiel tilgang?

- ▶ Lineær søgning
- ▶ Find største element
- ▶ Find sum og antal af elementer (og dermed gennemsnit)
- ▶ Selectionsort
- ▶ Merge af to sorterede lister til een (og dermed Mergesort)
- ▶ Generaliseret merge (eksempler senere)

Merge

Slå to (stigende) sorterede lister A og B sammen til en sorteret liste C .

Merge

Slå to (stigende) sorterede lister A og B sammen til en sorteret liste C .

Et **merge-skridt**: Sammenlign nuværende forreste i A og B , og flyt mindste af disse over sidst i C .

Merge

Slå to (stigende) sorterede lister A og B sammen til en sorteret liste C .

Et **merge-skridt**: Sammenlign nuværende forreste i A og B , og flyt mindste af disse over sidst i C .

Start med tom C

Så længe både A og B er ikke-tomme:

 Udfør et merge-skridt

Hvis enten A eller B er ikke-tom:

 Flyt resten af dens elementer over sidst i C

Merge

Slå to (stigende) sorterede lister A og B sammen til en sorteret liste C .

Et **merge-skridt**: Sammenlign nuværende forreste i A og B , og flyt mindste af disse over sidst i C .

Start med tom C

Så længe både A og B er ikke-tomme:

 Udfør et merge-skridt

Hvis enten A eller B er ikke-tom:

 Flyt resten af dens elementer over sidst i C

Korrekt metode (dvs. at C ved afslutning er sorteret)?

Merge

Slå to (stigende) sorterede lister A og B sammen til en sorteret liste C .

Et **merge-skridt**: Sammenlign nuværende forreste i A og B , og flyt mindste af disse over sidst i C .

Start med tom C

Så længe både A og B er ikke-tomme:

 Udfør et merge-skridt

Hvis enten A eller B er ikke-tom:

 Flyt resten af dens elementer over sidst i C

Korrekt metode (dvs. at C ved afslutning er sorteret)?

Korrekthed kan ses ved hjælp af en **invariant**.

Korrekthed og køretid for merge

Slå to (stigende) sorterede lister A og B sammen til en sorteret liste C .

Et **merge-skridt**: Sammenlign nuværende forreste i A og B , og flyt mindste af disse over sidst i C .

Start med tom C

Så længe både A og B er ikke-tomme:

 Udfør et merge-skridt

Hvis enten A eller B er ikke-tom:

 Flyt resten af dens elementer over sidst i C

Korrekthed og køretid for merge

Slå to (stigende) sorterede lister A og B sammen til en sorteret liste C .

Et **merge-skridt**: Sammenlign nuværende forreste i A og B , og flyt mindste af disse over sidst i C .

Start med tom C

Så længe både A og B er ikke-tomme:

 Udfør et merge-skridt

Hvis enten A eller B er ikke-tom:

 Flyt resten af dens elementer over sidst i C

Korrekthed (at C ved afslutning er sorteret) ses ved flg. **invariant**:

Nuværende version af A , B , C er sorterede, og ingen elementer i A , B er mindre end noget element i C .

Korrekthed og køretid for merge

Slå to (stigende) sorterede lister A og B sammen til een sorteret liste C .

Et **merge-skridt**: Sammenlign nuværende forreste i A og B , og flyt mindste af disse over sidst i C .

Start med tom C

Så længe både A og B er ikke-tomme:

 Udfør et merge-skridt

Hvis enten A eller B er ikke-tom:

 Flyt resten af dens elementer over sidst i C

Korrekthed (at C ved afslutning er sorteret) ses ved flg. **invariant**:

Nuværende version af A , B , C er sorterede, og ingen elementer i A , B er mindre end noget element i C .

Tid:

Korrekthed og køretid for merge

Slå to (stigende) sorterede lister A og B sammen til en sorteret liste C .

Et **merge-skridt**: Sammenlign nuværende forreste i A og B , og flyt mindste af disse over sidst i C .

Start med tom C

Så længe både A og B er ikke-tomme:

 Udfør et merge-skridt

Hvis enten A eller B er ikke-tom:

 Flyt resten af dens elementer over sidst i C

Korrekthed (at C ved afslutning er sorteret) ses ved flg. **invariant**:

Nuværende version af A , B , C er sorterede, og ingen elementer i A , B er mindre end noget element i C .

Tid: $\Theta(|A| + |B|)$

Mergesort

Merge lister af længde 1 (disse er automatisk sorterede) parvis sammen til sorterede lister af længde 2. Arbejdet svarer til at røre alle elementer een gang, dvs. er $\Theta(n)$.

Derefter merge de sorterede lister af længde 2 til sorterede lister af længde 4. Arbejdet svarer til at røre alle elementer een gang, dvs. er $\Theta(n)$.

Derefter længde 4 til længde 8, . . . ,

Indtil længde n nås, dvs. alle elementer er i een sorteret liste. Dvs. vi har sorteret elementerne.

Mergesort

Merge lister af længde 1 (disse er automatisk sorterede) parvis sammen til sorterede lister af længde 2. Arbejdet svarer til at røre alle elementer een gang, dvs. er $\Theta(n)$.

Derefter merge de sorterede lister af længde 2 til sorterede lister af længde 4. Arbejdet svarer til at røre alle elementer een gang, dvs. er $\Theta(n)$.

Derefter længde 4 til længde 8, . . . ,

Indtil længde n nås, dvs. alle elementer er i een sorteret liste. Dvs. vi har sorteret elementerne.

Der er lavet $\log n$ runder når længde n nås [da $2^k = n \Leftrightarrow k = \log n$].

Mergesort

Merge lister af længde 1 (disse er automatisk sorterede) parvis sammen til sorterede lister af længde 2. Arbejdet svarer til at røre alle elementer een gang, dvs. er $\Theta(n)$.

Derefter merge de sorterede lister af længde 2 til sorterede lister af længde 4. Arbejdet svarer til at røre alle elementer een gang, dvs. er $\Theta(n)$.

Derefter længde 4 til længde 8, . . . ,

Indtil længde n nås, dvs. alle elementer er i een sorteret liste. Dvs. vi har sorteret elementerne.

Der er lavet $\log n$ runder når længde n nås [da $2^k = n \Leftrightarrow k = \log n$].

Tid i alt:

Mergesort

Merge lister af længde 1 (disse er automatisk sorterede) parvis sammen til sorterede lister af længde 2. Arbejdet svarer til at røre alle elementer een gang, dvs. er $\Theta(n)$.

Derefter merge de sorterede lister af længde 2 til sorterede lister af længde 4. Arbejdet svarer til at røre alle elementer een gang, dvs. er $\Theta(n)$.

Derefter længde 4 til længde 8, . . . ,

Indtil længde n nås, dvs. alle elementer er i een sorteret liste. Dvs. vi har sorteret elementerne.

Der er lavet $\log n$ runder når længde n nås [da $2^k = n \Leftrightarrow k = \log n$].

Tid i alt: $\Theta(n \log n)$ [da vi $\log n$ gange laver $\Theta(n)$ arbejde].

Mergesort

Merge lister af længde 1 (disse er automatisk sorterede) parvis sammen til sorterede lister af længde 2. Arbejdet svarer til at røre alle elementer een gang, dvs. er $\Theta(n)$.

Derefter merge de sorterede lister af længde 2 til sorterede lister af længde 4. Arbejdet svarer til at røre alle elementer een gang, dvs. er $\Theta(n)$.

Derefter længde 4 til længde 8, . . . ,

Indtil længde n nås, dvs. alle elementer er i een sorteret liste. Dvs. vi har sorteret elementerne.

Der er lavet $\log n$ runder når længde n nås [da $2^k = n \Leftrightarrow k = \log n$].

Tid i alt: $\Theta(n \log n)$ [da vi $\log n$ gange laver $\Theta(n)$ arbejde].

Se illustration af algoritmen og dens analyse næste side.

Mergesort

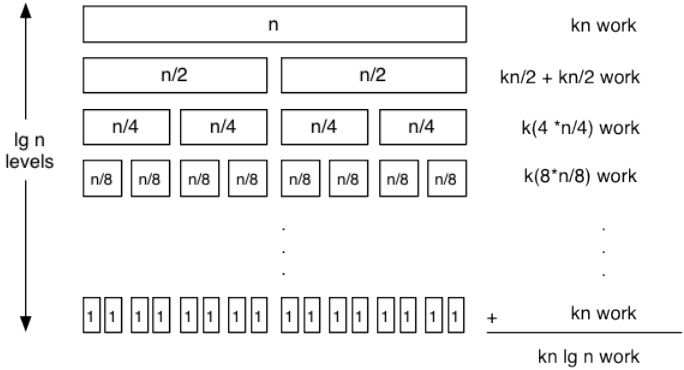


Figure: Andrew Myers, Cornell University

Generaliseret merge

Under merge af to sorterede lister A og B vil elementer med samme værdi "møde" hinanden under merge-skridtene.

Generaliseret merge

Under merge af to sorterede lister A og B vil elementer med samme værdi "møde" hinanden under merge-skridtene.

Eksempel: A er datafil med (ID,data) elementer, sorteret efter ID, B er liste af (ID,opdatering) elementer som angiver opdateringer, sorteret efter ID.

At gennemløbe A og B via merge-skridt vil tillade os at opdatere data i A . NB: de to lister behøver ikke være lige lange.

Generaliseret merge

Under merge af to sorterede lister A og B vil elementer med samme værdi "møde" hinanden under merge-skridtene.

Eksempel: A er datafil med (ID,data) elementer, sorteret efter ID, B er liste af (ID,opdatering) elementer som angiver opdateringer, sorteret efter ID.

At gennemløbe A og B via merge-skridt vil tillade os at opdatere data i A . NB: de to lister behøver ikke være lige lange.

Eksempel: To lister A og B repræsenterer mængder (og er derfor hver især uden dubletter).

At gennemløbe A og B via merge-skridt vil tillade os at generere f.eks. $A \cap B$: Hvis et merge-step ser to ens elementer som forreste i A og B , flyttes det ene over sidst i C , og den anden smides væk. Hvis et merge-step ser to forskellige, smides den mindste væk.

Hvordan laves API Random access tilgang?

Een metode: Hashing.

Hvordan laves API Random access tilgang?

Een metode: Hashing.



Hvordan laves API Random access tilgang?

Een metode: Hashing.



Hash \sim hacher [fransk] \sim hakke i stykker.

Hashing

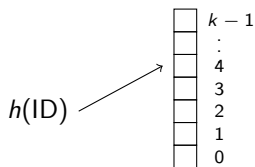
Ide: tildel hver ID et index i i et array A hvor element gemmes.
Hash-funktion h :

Hashing

Ide: tildel hver ID et index i i et array A hvor element gemmes.

Hash-funktion h :

$$h(\text{ID}) = \text{index } i \text{ i } A$$

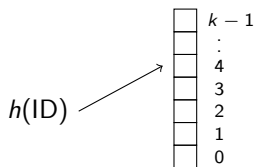


Hashing

Ide: tildel hver ID et index i et array A hvor element gemmes.

Hash-funktion h :

$$h(\text{ID}) = \text{index i } A$$



Eksempel (antag ID'er er heltal):

$$h(x) = x \bmod k$$

hvor $k = |A|$.

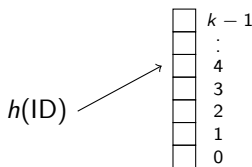
Bemærk at $h(x) \in \{0, 1, 2, \dots, k-1\}$, så det er altid et lovligt index.

Hashing

Ide: tildel hver ID et index i et array A hvor element gemmes.

Hash-funktion h :

$$h(\text{ID}) = \text{index i } A$$



Eksempel (antag ID'er er heltal):

$$h(x) = x \bmod k$$

hvor $k = |A|$.

Bemærk at $h(x) \in \{0, 1, 2, \dots, k-1\}$, så det er altid et lovligt index.

Med $k = 41$:	$h(46) = 5$	(da $1 \cdot 41 + 5 = 46$)
	$h(12) = 12$	(da $0 \cdot 41 + 12 = 12$)
	$h(100) = 18$	(da $2 \cdot 41 + 18 = 100$)
	$h(479869) = 5$	(da $11704 \cdot 41 + 5 = 479869$)

Hashing

Hvorfor ikke bare

$$h(x) = x?$$

Hashing

Hvorfor ikke bare

$$h(x) = x?$$

Eksempel: gem 5 CPR-numre.

Hashing

Hvorfor ikke bare

$$h(x) = x?$$

Eksempel: gem 5 CPR-numre.

CPR-numre: $180781-2345 \in \{0, 1, 2, \dots, 10^{10} - 1\}$

Så størrelse af array A skal være 10^{10} for at gemme 5 tal. Spild af plads (10^{10} bytes = 4 Gb).

Hashing

Hvorfor ikke bare

$$h(x) = x?$$

Eksempel: gem 5 CPR-numre.

CPR-numre: $180781-2345 \in \{0, 1, 2, \dots, 10^{10} - 1\}$

Så størrelse af array A skal være 10^{10} for at gemme 5 tal. Spild af plads (10^{10} bytes = 4 Gb).

Ofte er nøgler heltal (32 eller 64 bits), dvs. har $2^{32} \approx 10^{10}$ eller $2^{64} \approx 10^{20}$ muligheder.

Kollisioner

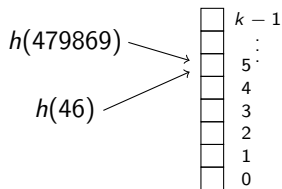
$$h(x) = x \bmod 41:$$

$h(46) = 5$	(da $1 \cdot 41 + 5 = 46$)
$h(12) = 12$	(da $0 \cdot 41 + 12 = 12$)
$h(100) = 18$	(da $2 \cdot 41 + 18 = 100$)
$h(479869) = 5$	(da $11704 \cdot 41 + 5 = 479869$)

Kollisioner

$h(x) = x \bmod 41$:

$$\begin{aligned}h(46) &= 5 && \text{(da } 1 \cdot 41 + 5 = 46\text{)} \\h(12) &= 12 && \text{(da } 0 \cdot 41 + 12 = 12\text{)} \\h(100) &= 18 && \text{(da } 2 \cdot 41 + 18 = 100\text{)} \\h(479869) &= 5 && \text{(da } 11704 \cdot 41 + 5 = 479869\text{)}\end{aligned}$$

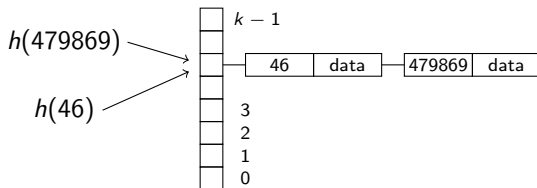


Een oplossing: Chaining

Gem alle elementer for en array-celle i en l nket liste.

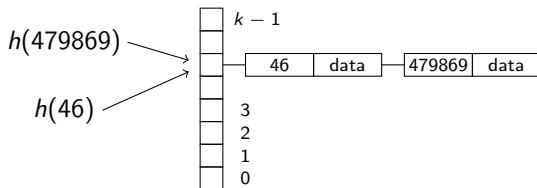
Een oplossing: Chaining

Gem alle elementer for en array-celle i en lænket liste.



Een oplossing: Chaining

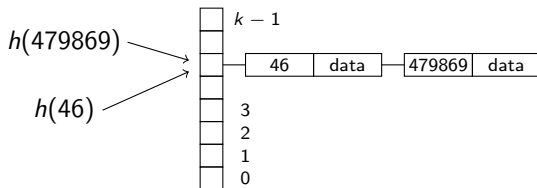
Gem alle elementer for en array-celle i en lænket liste.



De listen skal gennemløbes sekventielt, stiger tiden fra $\Theta(1)$ til $\Theta(|\text{liste}|)$.

Een løsning: Chaining

Gem alle elementer for en array-celle i en lænket liste.



De listen skal gennemløbes sekventielt, stiger tiden fra $\Theta(1)$ til $\Theta(|\text{liste}|)$.

Vi vil derfor gerne have få kollisioner.

Kan kollisioner undgås?

Afhænger af forholdet mellem hash-funktionen h og det konkrete datasæt som er gemt.

Kan kollisioner undgås?

Afhænger af forholdet mellem hash-funktionen h og det konkrete datasæt som er gemt.

I værste fald hash'er alle n elementer til samme celle. Tid: $\Theta(n)$.

Kan kollisioner undgås?

Afhænger af forholdet mellem hash-funktionen h og det konkrete datasæt som er gemt.

I værste fald hash'er alle n elementer til samme celle. Tid: $\Theta(n)$.

Hvis n (antal elementer gemt) er større end k (array-størrelse), er der mindst een kollision

Kan kollisioner undgås?

Afhænger af forholdet mellem hash-funktionen h og det konkrete datasæt som er gemt.

I værste fald hash'er alle n elementer til samme celle. Tid: $\Theta(n)$.

Hvis n (antal elementer gemt) er større end k (array-størrelse), er der mindst een kollision (duehulsprincippet).

Kan kollisioner undgås?

Afhænger af forholdet mellem hash-funktionen h og det konkrete datasæt som er gemt.

I værste fald hash'er alle n elementer til samme celle. Tid: $\Theta(n)$.

Hvis n (antal elementer gemt) er større end k (array-størrelse), er der mindst een kollision (duehulsprincippet).

Hvad hvis vi antager h "smider tallene ind i celler tilfældigt"?

(Set over alle mulige datasæt kan man argumentere for at dette svarer til jævnest mulig fordeling.)

Fødselsdage og Hashing

Situation: n tilfældige mennesker går ind i et rum.

Spørgsmål: Er der nogen der har fødselsdag samme dato i året?

Fødselsdage og Hashing

Situation: n tilfældige mennesker går ind i et rum.

Spørgsmål: Er der nogen der har fødselsdag samme dato i året?

At samle personer med tilfældige fødselsdage svarer til at indsætte tilfældigt i en tabel af størrelse 365 (= antal dage i et år).

Fødselsdagsparadokset

Situation: n tilfældige mennesker går ind i et rum.

Spørgsmål: Er der nogen der har fødselsdag samme dato i året?

Fødselsdagsparadokset

Situation: n tilfældige mennesker går ind i et rum.

Spørgsmål: Er der nogen der har fødselsdag samme dato i året?

n	Sandsynlighed for to med samme fødselsdag
0	0
1	0
2	$1/365$
\vdots	??
?	$1/2$
\vdots	??
366	1

Spørgsmål: for hvilket n bliver sandsynligheden $\geq 1/2$?

Fødselsdagsparadokset

Bemærk:

$$P(\text{nogen har samme fødseldag}) = 1 - P(\text{ingen har samme fødseldag})$$

Fødselsdagsparadokset

Bemærk:

$$P(\text{noget har samme fødseldag}) = 1 - P(\text{ingen har samme fødseldag})$$

Så vi kigger i stedet på $P(\text{ingen med samme fødseldag blandt de } n \text{ første personer})$, og spørger hvornår den er mindre end $1/2$.

Fødselsdagsparadokset

Bemærk:

$$P(\text{nogen har samme fødseldag}) = 1 - P(\text{ingen har samme fødseldag})$$

Så vi kigger i stedet på $P(\text{ingen med samme fødseldag blandt de } n \text{ første personer})$, og spørger hvornår den er mindre end $1/2$.

Ingen med samme fødseldag blandt de n første personer



1) Ingen med samme fødselsedag blandt de $n - 1$ første personer

OG

2) den n 'te persons fødselsdag falder ikke sammen med nogen af disses.

Fødselsdagsparadokset

Bemærk:

$$P(\text{nogen har samme fødseldag}) = 1 - P(\text{ingen har samme fødseldag})$$

Så vi kigger i stedet på $P(\text{ingen med samme fødseldag blandt de } n \text{ første personer})$, og spørger hvornår den er mindre end $1/2$.

Ingen med samme fødseldag blandt de n første personer



1) Ingen med samme fødselsedag blandt de $n - 1$ første personer

OG

2) den n 'te persons fødselsdag falder ikke sammen med nogen af disses.

Hvis 1) gælder, er præcis $n - 1$ forskellige datoer optaget når den n 'te person går ind i rummet. Hvis personerne antages at have tilfældige fødselsdage, er sandsynligheden for 2) derfor $(365 - (n - 1))/365$ når 1) gælder.

Fødselsdagsparadokset

Hvis vi kalder P (ingen med samme fødseldag blandt de n første personer) for s_n , kan vi se af ovenstående at

$$s_n = s_{n-1} \cdot \frac{365 - (n - 1)}{365}$$

Da s_1 naturligvis er 1 (med kun een person i rummet er der ingen med samme fødselsdag), ser vi at:

$$s_1 = 1$$

$$s_2 = 1 \cdot \frac{364}{365} = 0.9972\dots$$

$$s_3 = 1 \cdot \frac{364}{365} \cdot \frac{363}{365} = 0.9917\dots$$

$$s_4 = 1 \cdot \frac{364}{365} \cdot \frac{363}{365} \cdot \frac{362}{365} = 0.9836\dots$$

⋮

Fødselsdagsparadokset

Hvis man udregner disse udtryk (evt. via at skrive et program, der gør det), ses at

$$s_{22} = 0.5243 \dots$$

$$s_{23} = 0.4927 \dots$$

Vi var interesserede i, for hvilket n det var mere sandsynligt at nogen havde samme fødselsdags end at ingen havde.

Dvs. (se bemærkning på tidligere side) for hvilket n at $s_n \leq 1/2$

Af beregningen ovenfor kan vi nu svare: når n bliver 23.

Af samme slags beregninger fås f.eks. $s_{50} = 0.0296 \dots$

Fødselsdagsparadokset og Hashing

At samle personer med tilfældige fødselsdage svarer til at indsætte tilfældigt i en tabel af størrelse 365 (antal dage i et år).

Ovenstående metode kan derfor bruges til at beregne sandsynligheden for, at kollisioner opstår ved indsættelse af n elementer i en hashtabel, hvis det antages at hashfunktionen tildeler tabel-indekser til elementer på en tilfældig måde. (Man skal blot udskifte 356 med den aktuelle tabelstørrelse.)