

# Merging og hashing

# Mål

Målet for disse slides er at diskutere nogle metoder til at gemme og hente data effektivt.

Dette emne er et uddrag af kurset *DM507 Algoritmer og datastrukturer* (2. semester).

# Mål

Målet for disse slides er at diskutere nogle metoder til at gemme og hente data effektivt.

Dette emne er et uddrag af kurset *DM507 Algoritmer og* *datastrukturer* (2. semester).

# Datastrukturer

**Datastruktur** = metode til at gemme og hente data effektivt.

Software opbygges ofte i moduler med klare grænseflader til hinanden (jvf. objekt-orienteret programmering).

# Datastrukturer

**Datastruktur** = metode til at gemme og hente data effektivt.

Software opbygges ofte i moduler med klare grænseflader til hinanden (jvf. objekt-orienteret programmering).

Datastrukturer kan beskrives på to niveauer:

**Interface:** Hvordan kan man tilgå data set udefra (hvad er de tilbudte metoder)?

**Implementation:** Hvordan skal disse metoder implementeres (kodes)?

# Datastrukturer

**Datastruktur** = metode til at gemme og hente data effektivt.

Software opbygges ofte i moduler med klare grænseflader til hinanden (jvf. objekt-orienteret programmering).

Datastrukturer kan beskrives på to niveauer:

**Interface:** Hvordan kan man tilgå data set udefra (hvad er de tilbudte metoder)?

**Implementation:** Hvordan skal disse metoder implementeres (kodes)?

Dagens emne: to udbredte grundtyper (blandt flere) af interfaces:

▶ **Sekventiel tilgang**

▶ **Random access**

# Interface for sekventiel tilgang

# Interface for sekventiel tilgang

Metoder til læsning:

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



# Interface for sekventiel tilgang

Metoder til læsning:

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

Typisk anvendelse:

```
file = open(filename)
Så længe file.isEndOfFile() svarer False:
    x = file.readNext()
    (...gør noget med x...)
file.close()
```

7	2	3	4	9	4	7	2	8	1	6
---	---	---	---	---	---	---	---	---	---	---

# Interface for sekventiel tilgang

Metoder til læsning:

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

Typisk anvendelse:

```
file = open(filename)
```

```
Så længe file.isEndOfFile() svarer False:
```

```
    x = file.readNext()
```

```
    (...gør noget med x...)
```

```
file.close()
```

7	2	3	4	9	4	7	2	8	1	6
---	---	---	---	---	---	---	---	---	---	---



# Interface for sekventiel tilgang

Metoder til læsning:

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

Typisk anvendelse:

```
file = open(filename)
```

```
Så længe file.isEndOfFile() svarer False:
```

```
    x = file.readNext()
```

```
    (...gør noget med x...)
```

```
file.close()
```

7	2	3	4	9	4	7	2	8	1	6
---	---	---	---	---	---	---	---	---	---	---



# Interface for sekventiel tilgang

Metoder til læsning:

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

Typisk anvendelse:

```
file = open(filename)
Så længe file.isEndOfFile() svarer False:
    x = file.readNext()
    (...gør noget med x...)
file.close()
```

7	2	3	4	9	4	7	2	8	1	6
---	---	---	---	---	---	---	---	---	---	---

↑ ...

# Interface for sekventiel tilgang

Metoder til læsning:

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

Typisk anvendelse:

```
file = open(filename)
```

```
Så længe file.isEndOfFile() svarer False:
```

```
    x = file.readNext()
```

```
    (...gør noget med x...)
```

```
file.close()
```

7	2	3	4	9	4	7	2	8	1	6
---	---	---	---	---	---	---	---	---	---	---

... ↑

# Interface for sekventiel tilgang

Metoder til læsning:

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

Typisk anvendelse:

```
file = open(filename)
```

```
Så længe file.isEndOfFile() svarer False:
```

```
    x = file.readNext()
```

```
    (...gør noget med x...)
```

```
file.close()
```

7	2	3	4	9	4	7	2	8	1	6
---	---	---	---	---	---	---	---	---	---	---



# Interface for sekventiel tilgang

Metoder til skrivning:

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

# Interface for sekventiel tilgang

Metoder til skrivning:

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





# Interface for sekventiel tilgang

Metoder til skrivning:

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

4



# Interface for sekventiel tilgang

Metoder til skrivning:

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

4	6
---	---



# Interface for sekventiel tilgang

Metoder til skrivning:

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

4	6	1
---	---	---



# Interface for sekventiel tilgang

Metoder til skrivning:

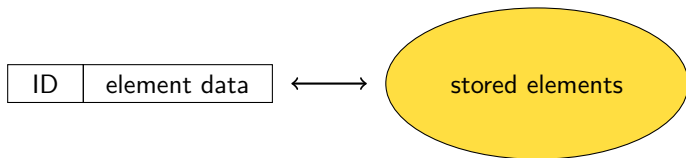
Metoder: `writeNext()`, `open()`, `close()`

4	6	1	7
---	---	---	---



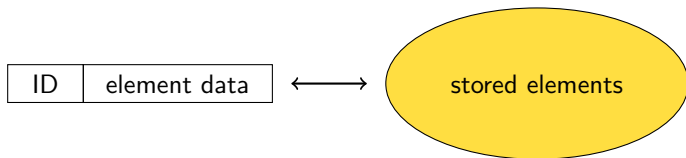
## Interface for random access

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



# Interface for random access

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



Metoder:

`findElm(ID)` (returnerer element data)

`insertElm(ID,data)`

`deleteElm(ID)`

`open()`, `close()`

## Interface for random access

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

Eksempler:

# Interface for random access

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

Eksempler:

- ▶ Databaser (ID = CPR, f.eks.)



# Interface for random access

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

Eksempler:

- ▶ Databaser (ID = CPR, f.eks.)
- ▶ Dictionaries i Python

```
data = myDict[ID]
myDict[ID] = data
myDict.pop(ID) (eller del myDict[ID])
```

# Interface for random access

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

Eksempler:

- ▶ Databaser (ID = CPR, f.eks.)
- ▶ Dictionaries i Python

```
data = myDict[ID]
myDict[ID] = data
myDict.pop(ID) (eller del myDict[ID])
```
- ▶ Lister i Python kan ses som et specialtilfælde hvor ID = index

```
data = myList[7]
myList[7] = data
myList[7] = None
```

# Dagens spørgsmål

1. Hvad kan det simple interface Sekventiel tilgang bruges til?

# Dagens spørgsmål

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

# Dagens spørgsmål

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

Bemærk: Alle datakilder kan tilgås med sekventiel tilgang. (Og for nogle kan *kun* sekventiel tilgang laves effektivt.)

- ▶ Harddisk
- ▶ SSD (Solid State Disk)
- ▶ CD
- ▶ Bånd
- ▶ Streaming over net
- ▶ Data genereret on-the-fly af et andet program
- ▶ Data i en liste (Python) eller i et array (Java).

Spørgsmål 1: Hvad kan laves med sekventiel tilgang?

# Spørgsmål 1: Hvad kan laves med sekventiel tilgang?

F.eks.:

- ▶ Lineær søgning

# Spørgsmål 1: Hvad kan laves med sekventiel tilgang?

F.eks.:

- ▶ Lineær søgning
- ▶ Find største element



# Spørgsmål 1: Hvad kan laves med sekventiel tilgang?

F.eks.:

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

# Spørgsmål 1: Hvad kan laves med sekventiel tilgang?

F.eks.:

- ▶ Lineær søgning
- ▶ Find største element
- ▶ Find sum og antal af elementer (og dermed gennemsnit)
- ▶ Merge: slå to sortererede lister sammen til én sorteret liste (detaljer om lidt)

# Spørgsmål 1: Hvad kan laves med sekventiel tilgang?

F.eks.:

- ▶ Lineær søgning
- ▶ Find største element
- ▶ Find sum og antal af elementer (og dermed gennemsnit)
- ▶ Merge: slå to sorterede lister sammen til én sorteret liste (detaljer om lidt)
- ▶ Mergesort (detaljer om lidt)

# Spørgsmål 1: Hvad kan laves med sekventiel tilgang?

F.eks.:

- ▶ Lineær søgning
- ▶ Find største element
- ▶ Find sum og antal af elementer (og dermed gennemsnit)
- ▶ Merge: slå to sorterede lister sammen til én sorteret liste (detaljer om lidt)
- ▶ Mergesort (detaljer om lidt)
- ▶ Generaliseret merge (eksempel om lidt)

# Spørgsmål 1: Hvad kan laves med sekventiel tilgang?

F.eks.:

- ▶ Lineær søgning
- ▶ Find største element
- ▶ Find sum og antal af elementer (og dermed gennemsnit)
- ▶ Merge: slå to sorterede lister sammen til én sorteret liste (detaljer om lidt)
- ▶ Mergesort (detaljer om lidt)
- ▶ Generaliseret merge (eksempel om lidt)
- ▶ Bubblesort, selectionsort, insertionsort, quicksort (ikke pensum i DM534)

# Spørgsmål 1: Hvad kan laves med sekventiel tilgang?

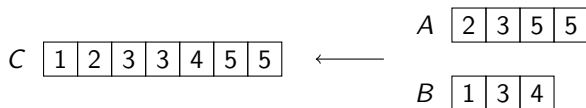
F.eks.:

- ▶ Lineær søgning
- ▶ Find største element
- ▶ Find sum og antal af elementer (og dermed gennemsnit)
- ▶ Merge: slå to sortererede lister sammen til én sorteret liste (detaljer om lidt)
- ▶ Mergesort (detaljer om lidt)
- ▶ Generaliseret merge (eksempel om lidt)
- ▶ Bubblesort, selectionsort, insertionsort, quicksort (ikke pensum i DM534)

**Morale:** Man kan lave forbavsende mange algoritmer baseret på det simple interface Sekventiel tilgang.

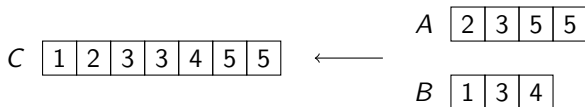
# Merge

Mål: Slå to sorterede lister  $A$  og  $B$  sammen til én sorteret liste  $C$ .



# Merge

Mål: Slå to sorterede lister  $A$  og  $B$  sammen til én sorteret liste  $C$ .

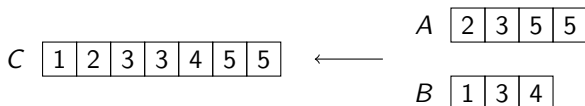


Et **merge-skridt**: Sammenlign (nuværende) forreste element i  $A$  og  $B$ , og flyt det mindste af disse over sidst i  $C$ .



# Merge

Mål: Slå to sorterede lister  $A$  og  $B$  sammen til én sorteret liste  $C$ .



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

Start med tom  $C$

Så længe både  $A$  og  $B$  er ikke-tomme:

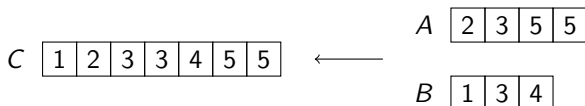
    Udfør et merge-skridt

Hvis enten  $A$  eller  $B$  er ikke-tom:

    Flyt resten af dens elementer over sidst i  $C$

# Merge

Mål: Slå to sorterede lister  $A$  og  $B$  sammen til én sorteret liste  $C$ .



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

Start med tom  $C$

Så længe både  $A$  og  $B$  er ikke-tomme:

    Udfør et merge-skridt

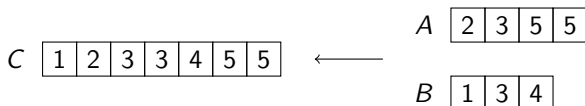
Hvis enten  $A$  eller  $B$  er ikke-tom:

    Flyt resten af dens elementer over sidst i  $C$

Er det en korrekt metode (dvs. er  $C$  ved afslutning sorteret)?

# Merge

Mål: Slå to sorterede lister  $A$  og  $B$  sammen til én sorteret liste  $C$ .



Et **merge-skridt**: Sammenlign (nuværende) forreste element i  $A$  og  $B$ , og flyt det 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  $C$

Er det en korrekt metode (dvs. er  $C$  ved afslutning sorteret)?

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

# Korrekthed og køretid for merge

Mål: Slå to (stigende) sorterede lister  $A$  og  $B$  sammen til én sorteret liste  $C$ .

Et merge-skridt: Sammenlign (nuværende) forreste element i  $A$  og  $B$ , og flyt det 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

Mål: Slå to (stigende) sorterede lister  $A$  og  $B$  sammen til én sorteret liste  $C$ .

Et merge-skridt: Sammenlign (nuværende) forreste element i  $A$  og  $B$ , og flyt det 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) kan 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

Mål: Slå to (stigende) sorterede lister  $A$  og  $B$  sammen til én sorteret liste  $C$ .

Et merge-skridt: Sammenlign (nuværende) forreste element i  $A$  og  $B$ , og flyt det 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) kan 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

Mål: Slå to (stigende) sorterede lister  $A$  og  $B$  sammen til én sorteret liste  $C$ .

Et merge-skridt: Sammenlign (nuværende) forreste element i  $A$  og  $B$ , og flyt det 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) kan 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:  $O(|A| + |B|)$ , da hvert element røres én gang.

# Mergesort

**Runde 1:** Merge lister af længde **1** (disse er automatisk sorterede) parvis sammen til sorterede lister af længde **2** ( $= 2^1$ ). Arbejdet svarer til at røre alle elementer én gang, dvs. er  $O(n)$ .

**Runde 2:** Merge de sorterede lister af længde 2 til sorterede lister af længde **4** ( $= 2^2$ ). Arbejdet svarer til at røre alle elementer én gang, dvs. er  $O(n)$ .

**Runde 3...:** Længde 4 til længde **8** ( $= 2^3$ ),...

Efter  $i$  runder: **længde**  $= 2^i$ .

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



# Mergesort

**Runde 1:** Merge lister af længde **1** (disse er automatisk sorterede) parvis sammen til sorterede lister af længde **2** ( $= 2^1$ ). Arbejdet svarer til at røre alle elementer én gang, dvs. er  $O(n)$ .

**Runde 2:** Merge de sorterede lister af længde 2 til sorterede lister af længde **4** ( $= 2^2$ ). Arbejdet svarer til at røre alle elementer én gang, dvs. er  $O(n)$ .

**Runde 3...:** Længde 4 til længde **8** ( $= 2^3$ ),...

Efter  $i$  runder: længde  $= 2^i$ .

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

Der er lavet  $\log_2 n$  runder når længde  $n$  nås [da  $2^i = n \Leftrightarrow i = \log_2 n$ ].

# Mergesort

**Runde 1:** Merge lister af længde **1** (disse er automatisk sorterede) parvis sammen til sorterede lister af længde **2** ( $= 2^1$ ). Arbejdet svarer til at røre alle elementer én gang, dvs. er  $O(n)$ .

**Runde 2:** Merge de sorterede lister af længde 2 til sorterede lister af længde **4** ( $= 2^2$ ). Arbejdet svarer til at røre alle elementer én gang, dvs. er  $O(n)$ .

**Runde 3...:** Længde 4 til længde **8** ( $= 2^3$ ),...

Efter  $i$  runder: længde  $= 2^i$ .

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

Der er lavet  $\log_2 n$  runder når længde  $n$  nås [da  $2^i = n \Leftrightarrow i = \log_2 n$ ].

Tid i alt:

# Mergesort

**Runde 1:** Merge lister af længde **1** (disse er automatisk sorterede) parvis sammen til sorterede lister af længde **2** ( $= 2^1$ ). Arbejdet svarer til at røre alle elementer én gang, dvs. er  $O(n)$ .

**Runde 2:** Merge de sorterede lister af længde 2 til sorterede lister af længde **4** ( $= 2^2$ ). Arbejdet svarer til at røre alle elementer én gang, dvs. er  $O(n)$ .

**Runde 3...:** Længde 4 til længde **8** ( $= 2^3$ ),...

Efter  $i$  runder: længde  $= 2^i$ .

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

Der er lavet  $\log_2 n$  runder når længde  $n$  nås [da  $2^i = n \Leftrightarrow i = \log_2 n$ ].

Tid i alt:  $O(n \log_2 n)$  [da vi  $\log_2 n$  gange laver  $O(n)$  arbejde].

# Mergesort

Illustration af algoritme og analyse:

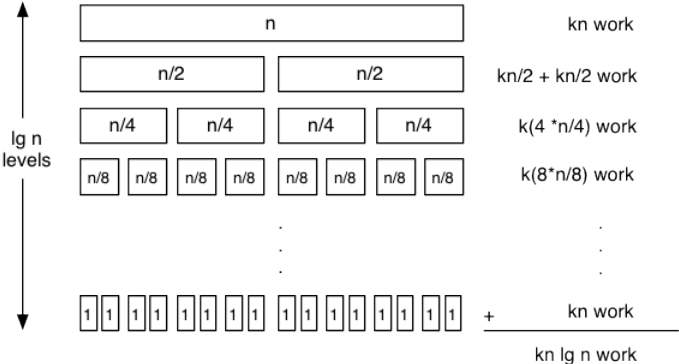


Figure: Andrew Myers, Cornell University

## Generaliseret merge

Under merge af to sorterede lister  $A$  og  $B$  vil elementer med samme værdi “møde” hinanden under merge-skridtene. Dette kan udnyttes:

## Generaliseret merge

Under merge af to sorterede lister  $A$  og  $B$  vil elementer med samme værdi “møde” hinanden under merge-skridtene. Dette kan udnyttes:

**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. Dette kan udnyttes:

**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 det andet smides væk. Hvis et merge-step ser to forskellige, smides det mindste væk.

## Spørgsmål 2: Hvordan laves interface Random access?

Én metode: Hashing.



## Spørgsmål 2: Hvordan laves interface Random access?

Én metode: Hashing.



## Spørgsmål 2: Hvordan laves interface Random access?

Én metode: Hashing.



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

# Hashing

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

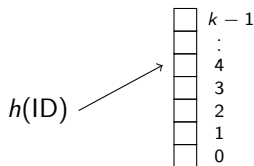
Hash-funktion  $h$ :

# Hashing

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

Hash-funktion  $h$ :

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

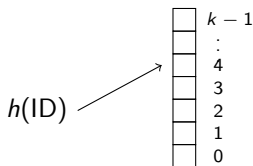


# Hashing

Idé: 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):

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

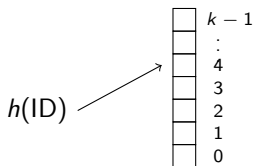
hvor  $k = |A|$  og mod (modulus) betegner rest ved heltalsdivision.

# Hashing

Idé: 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  $k = |A|$  og mod (modulus) betegner rest ved heltalsdivision.

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 $11704 \cdot 41 + 5 = 479869$ )

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

# Hashing

Når nu vi antager, at ID'er er heltal, hvorfor så ikke bare bruge

$$h(x) = x?$$

# Hashing

Når nu vi antager, at ID'er er heltal, hvorfor så ikke bare bruge

$$h(x) = x?$$

Eksempel: gem 5 CPR-numre.



# Hashing

Når nu vi antager, at ID'er er heltal, hvorfor så ikke bare bruge

$$h(x) = x?$$

Eksempel: gem 5 CPR-numre.

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

Så med ovenstående  $h(x)$  skal størrelse af array  $A$  skal være  $10^{10}$  for at gemme 5 tal. Spild af plads ( $10^{10}$  bytes  $\sim 4$  Gb).

# Hashing

Når nu vi antager, at ID'er er heltal, hvorfor så ikke bare bruge

$$h(x) = x?$$

Eksempel: gem 5 CPR-numre.

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

Så med ovenstående  $h(x)$  skal størrelse af array  $A$  skal være  $10^{10}$  for at gemme 5 tal. Spild af plads ( $10^{10}$  bytes  $\sim 4$  Gb).

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

# Hashing

Når nu vi antager, at ID'er er heltal, hvorfor så ikke bare bruge

$$h(x) = x?$$

Eksempel: gem 5 CPR-numre.

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

Så med ovenstående  $h(x)$  skal størrelse af array  $A$  skal være  $10^{10}$  for at gemme 5 tal. Spild af plads ( $10^{10}$  bytes  $\sim 4$  Gb).

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

Generelt ønsker vi, at tabellængden  $k$  cirka er lig antal gemte elementer  $n$  (så tabellen selv ikke fylder væsentlig mere end de gemte elementer).

# Kollisioner

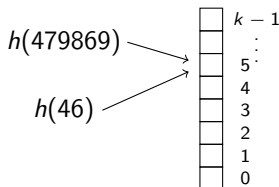
$h(x) = x \bmod 41$ :

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

# Kollisioner

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

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



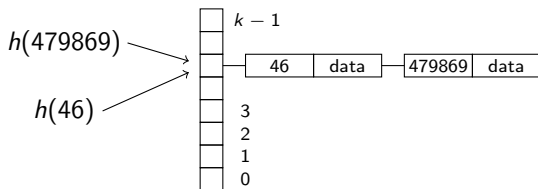
Problem: flere elementer vil bo på samme plads i array.

## Én løsning: Chaining

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

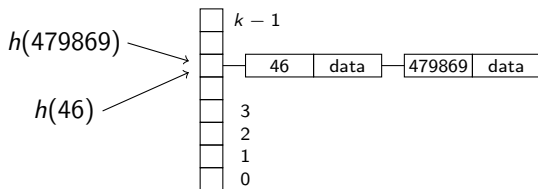
# Én løsning: Chaining

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



# Én løsning: Chaining

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

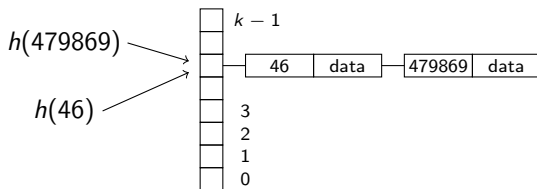


De listen skal gennemløbes sekventielt (under søgning og under sletning), stiger tiden fra  $O(1)$  til  $O(|\text{liste}|)$ .



# Én løsning: Chaining

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



De listen skal gennemløbes sekventielt (under søgning og under sletning), stiger tiden fra  $O(1)$  til  $O(|\text{liste}|)$ .

Vi vil derfor gerne have få kollisioner.

## Kan kollisioner undgås?

Det afhænger af forholdet mellem hash-funktionen  $h$  og det konkrete datasæt, som er gemt.

## Kan kollisioner undgås?

Det 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:  $O(n)$ .

## Kan kollisioner undgås?

Det 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:  $O(n)$ .

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

## Kan kollisioner undgås?

Det 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:  $O(n)$ .

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

## Kan kollisioner undgås?

Det 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:  $O(n)$ .

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

Hvad hvis vi antager at  $h$  "indsætter tallene i celler tilfældigt"?

NB: Dette er ikke en brugbar hash-funktion, da vi ikke kan finde elementerne igen (vi kan ikke huske, hvor de er lagt). Men det giver en jævn spredning i hashtabellen for *alle* datasæt, og kan derfor ses som en opførelse, vi ønsker af hash-funktioner. *Vi studerer derfor nu denne situation*, og ønsker at sige noget om, hvor hurtigt kollisioner opstår, når vi indsætter elementer.

# Fødselsdage og hashing

# Fødselsdage og hashing

Situation:  $n$  tilfældige mennesker går ind i et rum.

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

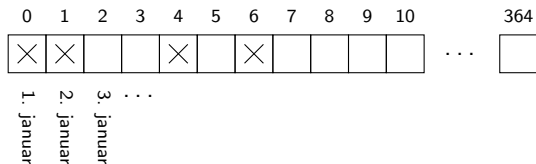


# Fødselsdage og hashing

Situation:  $n$  tilfældige mennesker går ind i et rum.

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

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



Ovenfor er fire personer samlet (hvor ingen har fødselsdag samme dag).

*Dette er altså et eksempel på den (hashing) situation vi ønsker at studere.*

## 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?

$n$	Sandsynlighed for (mindst) to med samme fødselsdag
0	0
1	0
2	$1/365$
$\vdots$	$\vdots$
<span style="border: 1px solid black; padding: 2px;">?</span>	$1/2$
$\vdots$	$\vdots$
366	1

Spørgsmål: for hvilket  $n$  bliver sandsynligheden for at der nogen med samme fødselsdag større end  $1/2$ ?

# Fødselsdage og hashing

**Spørgsmål**: for hvilket  $n$  bliver sandsynligheden for at der nogen med samme fødselsdag større end  $1/2$ ?

Bemærk:

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

# Fødselsdage og hashing

**Spørgsmål**: for hvilket  $n$  bliver sandsynligheden for at der nogen med samme fødselsdag større end  $1/2$ ?

Bemærk:

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

Så for at svare på spørgsmålet kigger vi i stedet på

$P(\text{ingen med samme fødseldag blandt de } n \text{ første personer}),$

og spørger, hvornår den er mindre end  $1 - 1/2 = 1/2$ .

## Fødselsdage og hashing

Ingen med samme fødseldag blandt de  $n$  første personer



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

OG

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

# Fødselsdage og hashing

Ingen med samme fødseldag blandt de  $n$  første personer



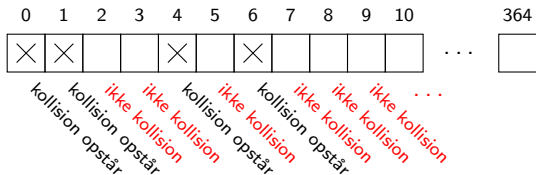
1) Ingen med samme fødselsdag blandt de  $n - 1$  første personer

OG

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

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  $(365 - (n - 1))/365$ .

Situationen for  $n = 5$  (dvs. 4 personer allerede til stede, uden kollision):



# Fødselsdage og hashing

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 én person i rummet er der helt sikkert ingen med samme fødselsdag), ser vi at:

$$s_1 = 1$$

$$s_2 = s_1 \cdot \frac{364}{365} = 1 \cdot \frac{364}{365} = 0.9972 \dots$$

$$s_3 = s_2 \cdot \frac{363}{365} = 1 \cdot \frac{364}{365} \cdot \frac{363}{365} = 0.9917 \dots$$

$$s_4 = s_3 \cdot \frac{362}{365} = 1 \cdot \frac{364}{365} \cdot \frac{363}{365} \cdot \frac{362}{365} = 0.9836 \dots$$

⋮



# Fødselsdagsparadokset

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

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

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

## 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. for hvilket  $n$  at  $s_n \leq 1/2$ .

Af beregningen ovenfor får vi vores svar: allerede når  $n$  bliver 23.

## 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. for hvilket  $n$  at  $s_n \leq 1/2$ .

Af beregningen ovenfor får vi vores svar: allerede når  $n$  bliver 23.

Navnet “fødselsdagsparadokset” bruges, fordi dette er et mindre antal personer end de fleste intuitivt vil gætte på.

# 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. for hvilket  $n$  at  $s_n \leq 1/2$ .

Af beregningen ovenfor får vi vores svar: allerede når  $n$  bliver 23.

Navnet “fødselsdagsparadokset” bruges, fordi dette er et mindre antal personer end de fleste intuitivt vil gætte på.

Af samme slags beregninger fås f.eks.  $s_{50} = 0.0296 \dots$ . Dvs. at der er under 3% sandsynlighed for, at der i en gruppe af 50 personer ikke er to med samme fødselsdato.

## Fødselsdagsparadokset og hashing

Recall: 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 365 med den aktuelle tabelstørrelse.

## Fødseldagsparadokset og hashing

Recall: 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 365 med den aktuelle tabelstørrelse.

**Morale:** vi kan se af sådanne beregninger, at den første kollision opstår overraskende hurtigt, når der indsættes i hashtabeller.

Det er derfor nødvendigt at have et system til at klare kollisioner (f.eks. de lænkede lister fra tidligere) for at kunne bruge hashing i praksis.