Merging og Hashing (del II)

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

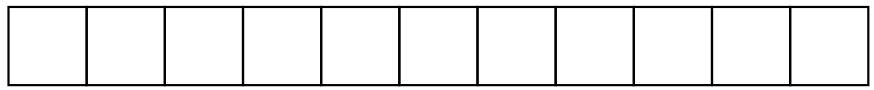
(API = Application Programming Interface: samling af metoder).

API for sekventiel tilgang

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

Læsning:

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

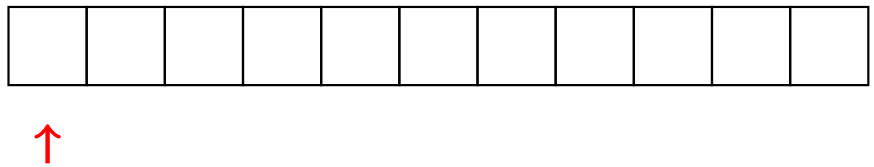


API for sekventiel tilgang

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

Læsning:

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

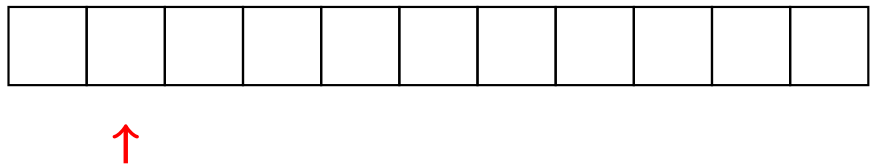


API for sekventiel tilgang

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

Læsning:

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

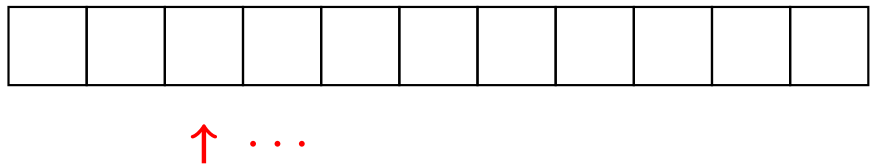


API for sekventiel tilgang

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

Læsning:

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

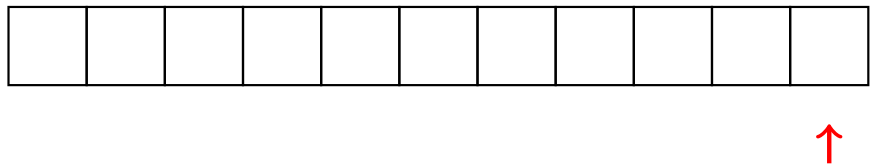


API for sekventiel tilgang

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

Læsning:

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

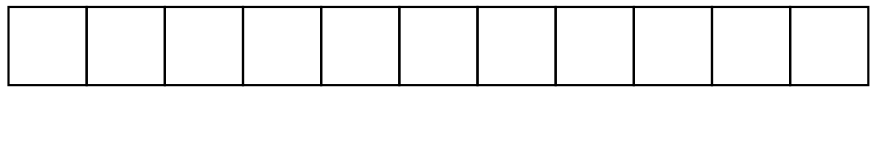


API for sekventiel tilgang

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

Læsning:

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

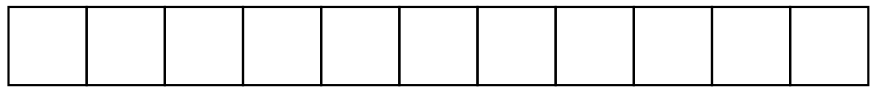


API for sekventiel tilgang

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

Læsning:

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



Skrivning:

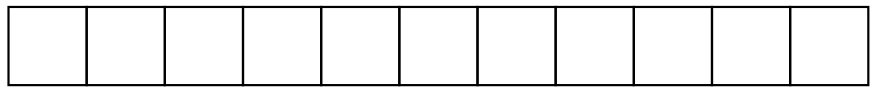
Operationer: `writeNext()`, `open()`, `close()`

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

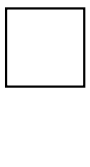
(API = Application Programming Interface: samling af metoder).

Læsning:

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

Skrivning:

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



API for sekventiel tilgang

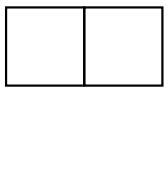
(API = Application Programming Interface: samling af metoder).

Læsning:

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

Skrivning:

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



API for sekventiel tilgang

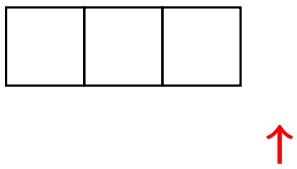
(API = Application Programming Interface: samling af metoder).

Læsning:

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

Skrivning:

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

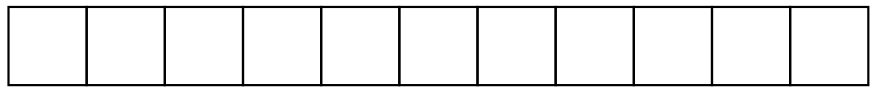


API for sekventiel tilgang

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

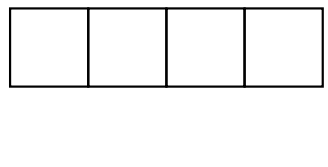
Læsning:

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:

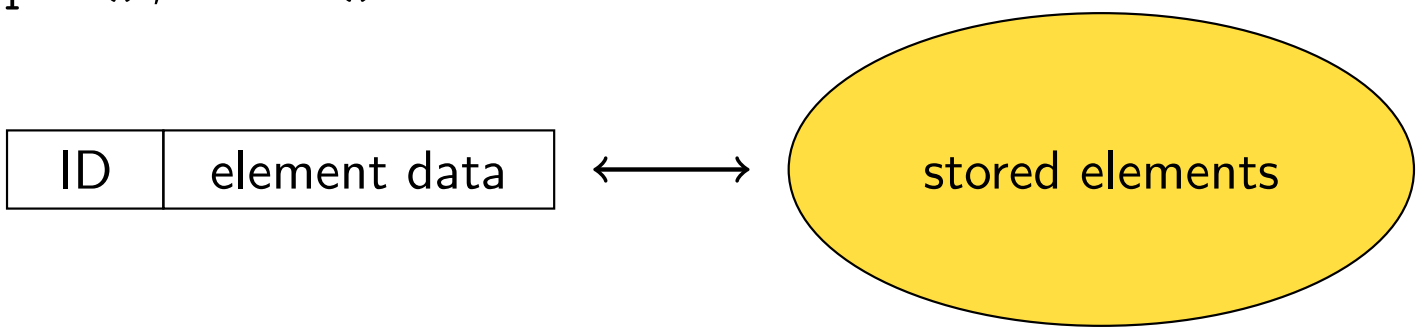
```
findElm(ID),
insertElm(ID,elementData),
deleteElm(ID),
open(), close()
```

API for random access tilgang

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

Operationer:

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

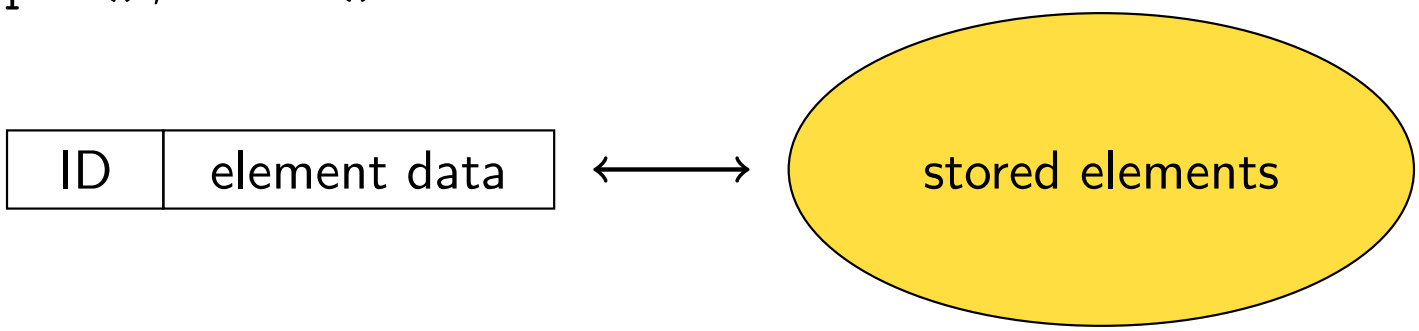


API for random access tilgang

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

Operationer:

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



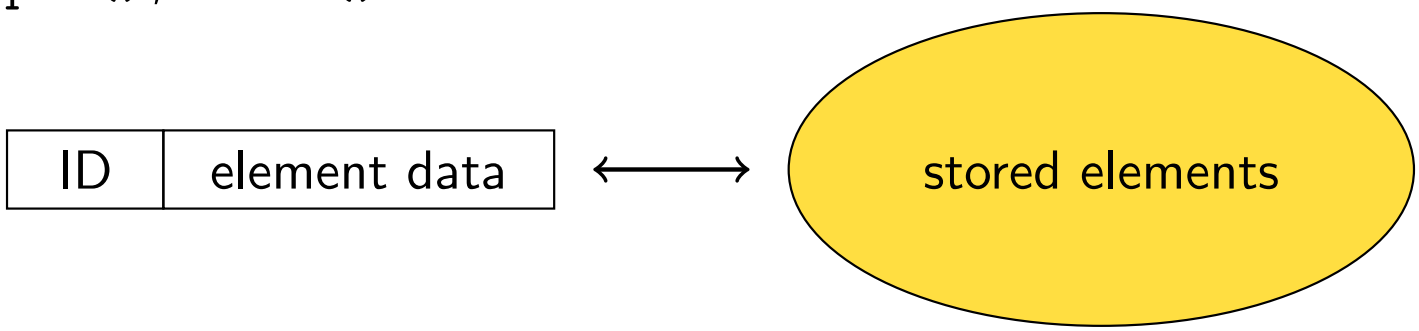
Jvf.:

API for random access tilgang

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

Operationer:

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



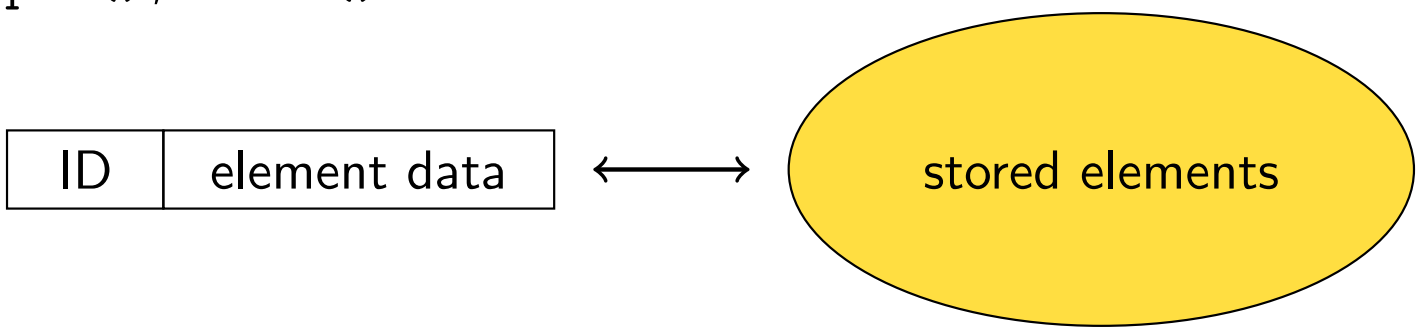
Jvf.:
Dictionary i Python
(`data = dict[ID]`, `dict[ID] = data`, `del dict[ID]`)

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]`)

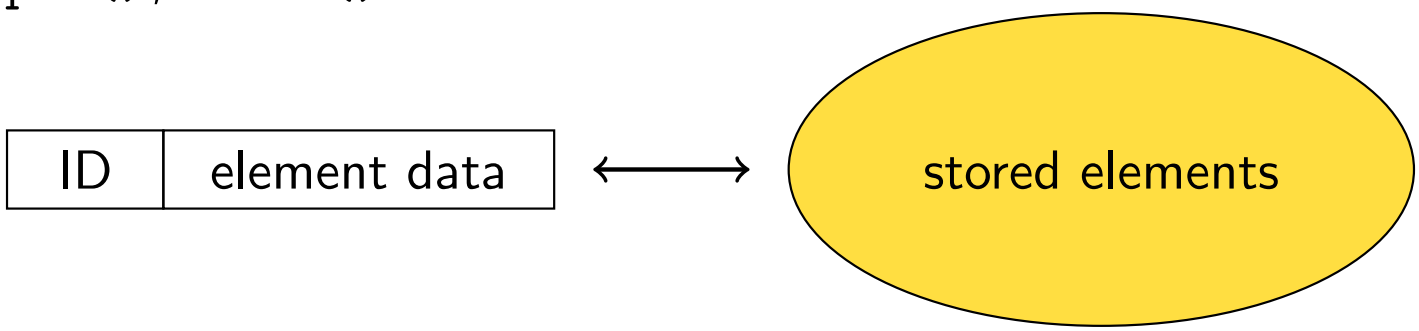
Databases (ID = CPR, f.eks.)

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)

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)

Denne gang: spørgsmål 1.(Forrige gang: spørgsmål 2.)

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 een sorteret liste C.

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.

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
Sålænge 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
Sålænge 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 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
Sålænge 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.

Invarianter

Invariant for algoritme: et udsagn om indholdet af hukommelsen (variable, arrays,. . .) som:

- Er sandt efter alle skridt.
- Ved algoritmens afslutning kan korrekthed af output udledes af udsagnet (samt de omstændigheder som fik algoritmen til at stoppe).

I praksis ofte een invariant for een samling skridt (f.eks. 0 eller flere gennemløb af en `while` eller `for` løkke). En anden invariant kan så om nødvendigt bruges for næste samling skridt af algoritmen.

Induktion

Invarianter vises at holde ved hjælp af induktion:

1) Invariant overholdt i starten 2) Invariant overholdt før et skridt $\Rightarrow$ overholdt efter	$\Rightarrow$	Invariant altid overholdt

(hvor "skridt" ofte er en iteration af en løkke). Dvs: Vis 1) og 2).

Induktion

Invarianter vises at holde ved hjælp af induktion:

1) Invariant overholdt i starten 2) Invariant overholdt før et skridt $\Rightarrow$ overholdt efter	$\Rightarrow$	Invariant altid overholdt

(hvor "skridt" ofte er en iteration af en løkke). Dvs: Vis 1) og 2).

Induktion $\sim$ "Dominoprincippet":

1) Brik 1 falder 2) Brik k falder $\Rightarrow$ brik $k+1$ falder	$\Rightarrow$	Alle brikker falder

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
Sålænge 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 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
> Sålænge 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.

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
> Sålænge 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:

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
> Sålænge 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 $\quad$ [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 $\quad$ [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].

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.