

# Prioritetskøer

# Prioritetskøer?



# Datastrukturer

Datastruktur = data + operationer herpå

# Datastrukturer

Datastruktur = data + operationer herpå

## Data:

- ▶ Ofte en ID + associeret data. ID kaldes også en nøgle (key).
- ▶ ID'er ofte fra et ordnet univers (har en ordning), f.eks. `int`, `float`, `String`.
- ▶ Vi nævner normalt ikke den associerede data. Dvs. elementer er reelt (ID,data) eller (ID,reference til data), men omtales blot som ID.

# Datastrukturer

Datastruktur = data + operationer herpå

## Data:

- ▶ Ofte en ID + associeret data. ID kaldes også en nøgle (key).
- ▶ ID'er ofte fra et ordnet univers (har en ordning), f.eks. `int`, `float`, `String`.
- ▶ Vi nævner normalt ikke den associerede data. Dvs. elementer er reelt (ID,data) eller (ID,reference til data), men omtales blot som ID.

## Operationer:

- ▶ Datastrukturens egenskaber udgøres af **de tilbudte operationer**, samt deres **køretider**.
- ▶ Målene er **fleksibilitet** og **effektivitet** (som regel modstridende mål).

# Datastrukturer

Tænk på en datastruktur som et API for adgang til en samling data.

- ▶ **Datastrukturer niveau 1:** de tilbudte operationer (i Java: et interface).
- ▶ **Datastrukturer niveau 2:** en konkret implementation af de tilbudte operationer (i Java: klasse som implementerer interfacet).

En givent sæt operationer kan have mange forskellig implementationer, ofte med forskellige køretider.

# Datastrukturer

Tænk på en datastruktur som et API for adgang til en samling data.

- ▶ **Datastrukturer niveau 1:** de tilbudte operationer (i Java: et interface).
- ▶ **Datastrukturer niveau 2:** en konkret implementation af de tilbudte operationer (i Java: klasse som implementerer interfacet).

En givent sæt operationer kan have mange forskellig implementationer, ofte med forskellige køretider.

DM507: katalog af [datastrukturer med bred anvendelse](#) samt [effektive implementationer heraf](#).

# Prioritetskøer

## Data:

- ▶ Element = nøgle (ID) fra et ordnet univers + associeret data.



# Prioritetskøer

## Data:

- ▶ Element = nøgle (ID) fra et ordnet univers + associeret data.

## Definerende operationer (max-version af prioritetskø):

- ▶  $Q$ .EXTRACT-MAX: Returnerer elementet med den største nøgle i prioritetskøen  $Q$  (et vilkårligt sådant element, hvis der er flere lige store). Elementet fjernes fra  $Q$ .
- ▶  $Q$ .INSERT( $e$ : element). Tilføjer elementet  $e$  til prioritetskøen  $Q$ .

# Prioritetskøer

## Data:

- ▶ Element = nøgle (ID) fra et ordnet univers + associeret data.

## Definerende operationer (max-version af prioritetskø):

- ▶  $Q$ .EXTRACT-MAX: Returnerer elementet med den største nøgle i prioritetskøen  $Q$  (et vilkårligt sådant element, hvis der er flere lige store). Elementet fjernes fra  $Q$ .
- ▶  $Q$ .INSERT( $e$ : element). Tilføjer elementet  $e$  til prioritetskøen  $Q$ .

Bemærk: vi kan sortere med disse operationer:

# Prioritetskøer

## Data:

- ▶ Element = nøgle (ID) fra et ordnet univers + associeret data.

## Definerende operationer (max-version af prioritetskø):

- ▶  $Q$ .EXTRACT-MAX: Returnerer elementet med den største nøgle i prioritetskøen  $Q$  (et vilkårligt sådant element, hvis der er flere lige store). Elementet fjernes fra  $Q$ .
- ▶  $Q$ .INSERT( $e$ : element). Tilføjer elementet  $e$  til prioritetskøen  $Q$ .

Bemærk: vi kan sortere med disse operationer:

$n \times$  INSERT

$n \times$  EXTRACT-MAX

# Prioritetskøer

## Data:

- ▶ Element = nøgle (ID) fra et ordnet univers + associeret data.

## Definerende operationer (max-version af prioritetskø):

- ▶  $Q$ .EXTRACT-MAX: Returnerer elementet med den største nøgle i prioritetskøen  $Q$  (et vilkårligt sådant element, hvis der er flere lige store). Elementet fjernes fra  $Q$ .
- ▶  $Q$ .INSERT( $e$ : element). Tilføjer elementet  $e$  til prioritetskøen  $Q$ .

Bemærk: vi kan sortere med disse operationer:

$n \times$  INSERT

$n \times$  EXTRACT-MAX

# Prioritetskøer

## Ekstra operationer:

- ▶  $Q.INCREASE-KEY(r: \text{reference til et element i } Q, k \text{ nøgle})$ . Ændrer nøglen til  $\max\{k, \text{gamle nøgle}\}$  for elementet refereret til af  $r$ .
- ▶  $Q.BUILD(L: \text{liste af elementer})$ . Bygger en prioritetskø indeholdende elementerne.

# Prioritetskøer

## Ekstra operationer:

- ▶  $Q.INCREASE-KEY(r: \text{reference til et element i } Q, k \text{ nøgle})$ . Ændrer nøglen til  $\max\{k, \text{gamle nøgle}\}$  for elementet refereret til af  $r$ .
- ▶  $Q.BUILD(L: \text{liste af elementer})$ . Bygger en prioritetskø indeholdende elementerne.

## Trivielle operationer for alle datastrukturer:

- ▶  $Q.ISEMPTY()$ ,  $Q.CREATENEW()$ ,  $Q.REMOVEEMPTY()$ .

Vil ikke blive nævnt fremover.

# Implementation via Heaps

En mulig implementation: brug heapstrukturen fra Heapsort.

# Implementation via Heaps

En mulig implementation: brug heapstrukturen fra Heapsort.

(NB: Arrayversionen af heaps kræver et kendt maximum for størrelsen  $n$  af køen. Alternativt kan array erstattes af et extendible array, f.eks. `java.util.ArrayList` i Java, eller man kan implementere heaptræet pointerbaseret.)



# Implementation via Heaps

En mulig implementation: brug heapstrukturen fra Heapsort.

(NB: Arrayversionen af heaps kræver et kendt maximum for størrelsen  $n$  af køen. Alternativt kan array erstattes af et extendible array, f.eks. `java.util.ArrayList` i Java, eller man kan implementere heaptræet pointerbaseret.)

Vi har allerede:

- ▶ EXTRACT-MAX:

# Implementation via Heaps

En mulig implementation: brug heapstrukturen fra Heapsort.

(NB: Arrayversionen af heaps kræver et kendt maximum for størrelsen  $n$  af køen. Alternativt kan array erstattes af et extendible array, f.eks. `java.util.ArrayList` i Java, eller man kan implementere heaptræet pointerbaseret.)

Vi har allerede:

- ▶ **EXTRACT-MAX**: Er essentielt hvad der bruges under anden del af Heapsort – fjern rod, flyt sidste blad op som rod, kald **HEAPIFY**.

# Implementation via Heaps

En mulig implementation: brug heapstrukturen fra Heapsort.

(NB: Arrayversionen af heaps kræver et kendt maximum for størrelsen  $n$  af køen. Alternativt kan array erstattes af et extendible array, f.eks. `java.util.ArrayList` i Java, eller man kan implementere heaptræet pointerbaseret.)

Vi har allerede:

- ▶ **EXTRACT-MAX**: Er essentielt hvad der bruges under anden del af Heapsort – fjern rod, flyt sidste blad op som rod, kald `HEAPIFY`.  
Køretid:

# Implementation via Heaps

En mulig implementation: brug heapstrukturen fra Heapsort.

(NB: Arrayversionen af heaps kræver et kendt maximum for størrelsen  $n$  af køen. Alternativt kan array erstattes af et extendible array, f.eks. `java.util.ArrayList` i Java, eller man kan implementere heaptræet pointerbaseret.)

Vi har allerede:

- ▶ **EXTRACT-MAX**: Er essentielt hvad der bruges under anden del af Heapsort – fjern rod, flyt sidste blad op som rod, kald `HEAPIFY`.  
Køretid:  $O(\log n)$ .
- ▶ **BUILD**:

# Implementation via Heaps

En mulig implementation: brug heapstrukturen fra Heapsort.

(NB: Arrayversionen af heaps kræver et kendt maximum for størrelsen  $n$  af køen. Alternativt kan array erstattes af et extendible array, f.eks. `java.util.ArrayList` i Java, eller man kan implementere heaptræet pointerbaseret.)

Vi har allerede:

- ▶ **EXTRACT-MAX**: Er essentielt hvad der bruges under anden del af Heapsort – fjern rod, flyt sidste blad op som rod, kald `HEAPIFY`.  
Køretid:  $O(\log n)$ .
- ▶ **BUILD**: Brug `HEAPIFY` gentagne gange bottom-up.

# Implementation via Heaps

En mulig implementation: brug heapstrukturen fra Heapsort.

(NB: Arrayversionen af heaps kræver et kendt maximum for størrelsen  $n$  af køen. Alternativt kan array erstattes af et extendible array, f.eks. `java.util.ArrayList` i Java, eller man kan implementere heaptræet pointerbaseret.)

Vi har allerede:

- ▶ **EXTRACT-MAX:** Er essentielt hvad der bruges under anden del af Heapsort – fjern rod, flyt sidste blad op som rod, kald `HEAPIFY`. Køretid:  $O(\log n)$ .
- ▶ **BUILD:** Brug `HEAPIFY` gentagne gange bottom-up. Køretid:

# Implementation via Heaps

En mulig implementation: brug heapstrukturen fra Heapsort.

(NB: Arrayversionen af heaps kræver et kendt maximum for størrelsen  $n$  af køen. Alternativt kan array erstattes af et extendible array, f.eks. `java.util.ArrayList` i Java, eller man kan implementere heaptræet pointerbaseret.)

Vi har allerede:

- ▶ **EXTRACT-MAX**: Er essentielt hvad der bruges under anden del af Heapsort – fjern rod, flyt sidste blad op som rod, kald `HEAPIFY`. Køretid:  $O(\log n)$ .
- ▶ **BUILD**: Brug `HEAPIFY` gentagne gange bottom-up. Køretid:  $O(n)$ .

# Implementation via Heaps

En mulig implementation: brug heapstrukturen fra Heapsort.

(NB: Arrayversionen af heaps kræver et kendt maximum for størrelsen  $n$  af køen. Alternativt kan array erstattes af et extendible array, f.eks. `java.util.ArrayList` i Java, eller man kan implementere heaptræet pointerbaseret.)

Vi har allerede:

- ▶ **EXTRACT-MAX**: Er essentielt hvad der bruges under anden del af Heapsort – fjern rod, flyt sidste blad op som rod, kald `HEAPIFY`. Køretid:  $O(\log n)$ .
- ▶ **BUILD**: Brug `HEAPIFY` gentagne gange bottom-up. Køretid:  $O(n)$ .

Mangler:

- ▶ **INSERT**
- ▶ **INCREASE-KEY**



# Increase-Key

# Increase-Key

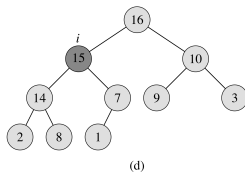
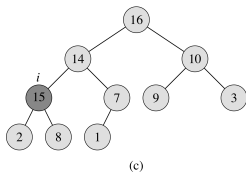
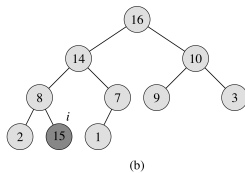
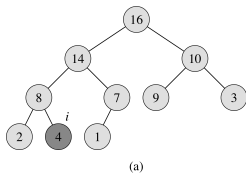
1. Ændre nøgle for element.

# Increase-Key

1. Ændre nøgle for element.
2. Genopret heaporden.

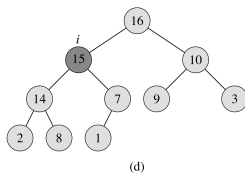
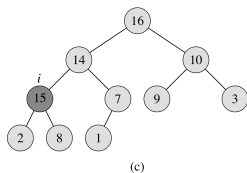
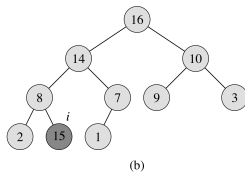
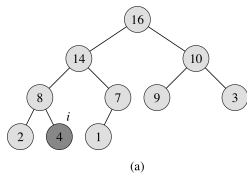
# Increase-Key

1. Ændre nøgle for element.
2. Genopret heaporden.



# Increase-Key

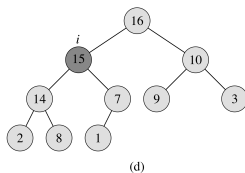
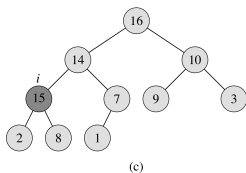
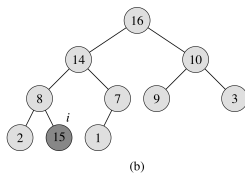
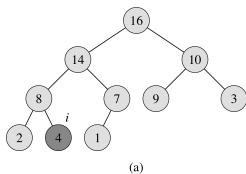
1. Ændre nøgle for element.
2. Genopret heaporden.



Køretid:

# Increase-Key

1. Ændre nøgle for element.
2. Genopret heaporden.



Køretid:  $O(\log n)$ .

# Insert

# Insert

1. Indsæt nye element sidst ( $\Rightarrow$  heapfacon i orden).



# Insert

1. Indsæt nye element sidst ( $\Rightarrow$  heapfacon i orden).
2. Genopret heaporden som i Increase-Key.

# Insert

1. Indsæt nye element sidst ( $\Rightarrow$  heapfacon i orden).
2. Genopret heaporden som i Increase-Key.

Køretid:

# Insert

1. Indsæt nye element sidst ( $\Rightarrow$  heapfacon i orden).
2. Genopret heaporden som i Increase-Key.

Køretid:  $O(\log n)$ .

## Forskellige implementationer er Prioritetskøer

	Heap	Lænket liste (usorteret)
EXTRACT-MAX	$O(\log n)$	$O(n)$
BUILD	$O(n)$	$O(1)$
INCREASE-KEY	$O(\log n)$	$O(1)$
INSERT	$O(\log n)$	$O(1)$