

Prioritetskøer

Prioritetskøer?



En prioritetskø er en **datastruktur**.

Datastrukturer

Datastruktur = data + operationer herpå

Data:

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

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

En givent sæt operationer (niveau 1) kan have mange forskellig implementationer (niveau 2), ofte med forskellige køretider.

I dette kursus: katalog af **datastrukturer (niveau 1) med bred anvendelse** samt **effektive implementationer (niveau 2)** heraf.

Prioritetskøer

Prioritetskøer, niveau 1:

Data:

- ▶ Element = nøgle (ID) fra et ordnet univers samt evt. yderligere data.

Centrale operationer (max-version af prioritetskø):

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

Bemærk: vi kan sortere med disse operationer:

$n \times \text{INSERT}$

$n \times \text{EXTRACT-MAX}$

Prioritetskøer

Prioritetskøer, niveau 1:

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 i listen L .

Trivielle operationer for alle datastrukturer:

- ▶ $Q.CREATENEWEMPTY()$, $Q.DESTRUCTEMPTY()$,
 $Q.ISEMPTY?()$.

Vil ikke blive nævnt fremover.

Implementation via heaps

En mulig implementation (niveau 2): brug heapstrukturen fra Heapsort.

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:

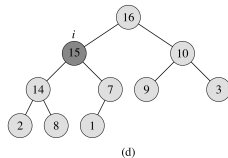
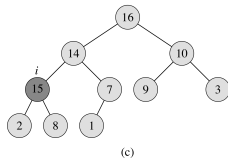
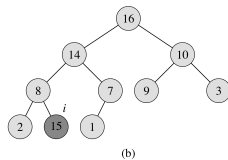
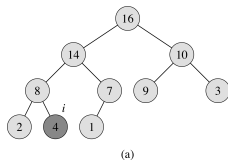
- ▶ **INSERT**
- ▶ **INCREASE-KEY**

[Detalje: I C# og Java kræver array-versionen af heaps et kendt maximum for størrelsen af køen. Alternativt kan array erstattes af et extendible array, f.eks. `List` i C#, `ArrayList` i Java eller lister i Python. Man kan også implementere heaptræet via pointere/referencer, lige som vi gør med søgetræer lidt senere.]

Increase-Key

1. Ændre nøgle for element.
2. Genopret heaporden: så længe elementet er stærkere end forælder, skift plads med denne.

Eksempel med $\text{INCREASE-KEY}(i, 15)$:



Køretid: Højden af træet, dvs. $O(\log n)$.

Insert

1. Indsæt det nye element sidst (\Rightarrow heapfacon i orden).
2. Genopret heaporden præcis som i Increase-Key: så længe elementet er større end forælder, skift plads med denne.

Køretid: Højden af træet, dvs. $O(\log n)$.

Forskellige implementationer af prioritetskøer

Samme niveau 1, forskellige niveau 2:

	<i>Heap</i>	<i>Usorteret liste</i>	<i>Sorteret liste</i>
EXTRACT-MAX	$O(\log n)$	$O(n)$	$O(1)$
BUILD	$O(n)$	$O(1)$	$O(n \log n)$
INCREASE-KEY	$O(\log n)$	$O(1)$	$O(n)$
INSERT	$O(\log n)$	$O(1)$	$O(n)$

Ovenstående samling operationer (niveau 1) er for max-prioritetskøer. Der er naturligvis nemt at lave min-prioritetskøer med operationerne

EXTRACT-MIN, BUILD, DECREASE-KEY, INSERT,

blot ved at vende alle uligheder mellem nøgler i definitioner og algoritmer.