Prioritetskøer

Prioritetskøer?

En prioritetskø er en datastruktur.

Datastrukturer

$$\boxed{\text{Datastruktur} = \text{data} + \text{operationer herpå}}$$

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 Java: et interface).
- Datastrukturer niveau 2: en konkret implementation af de tilbudte operationer (i Java: en 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 samt evt. yderligere data.

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

Trivielle operationer for alle datastrukturer:

- Q.CreateNewEmpty(), Q.RemoveEmpty(), Q.IsEmpty?().

Vil ikke blive nævnt fremover.

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 lister i Python. Man kan også implementere heaptræet via pointere/referencer.]

Vi har allerede:

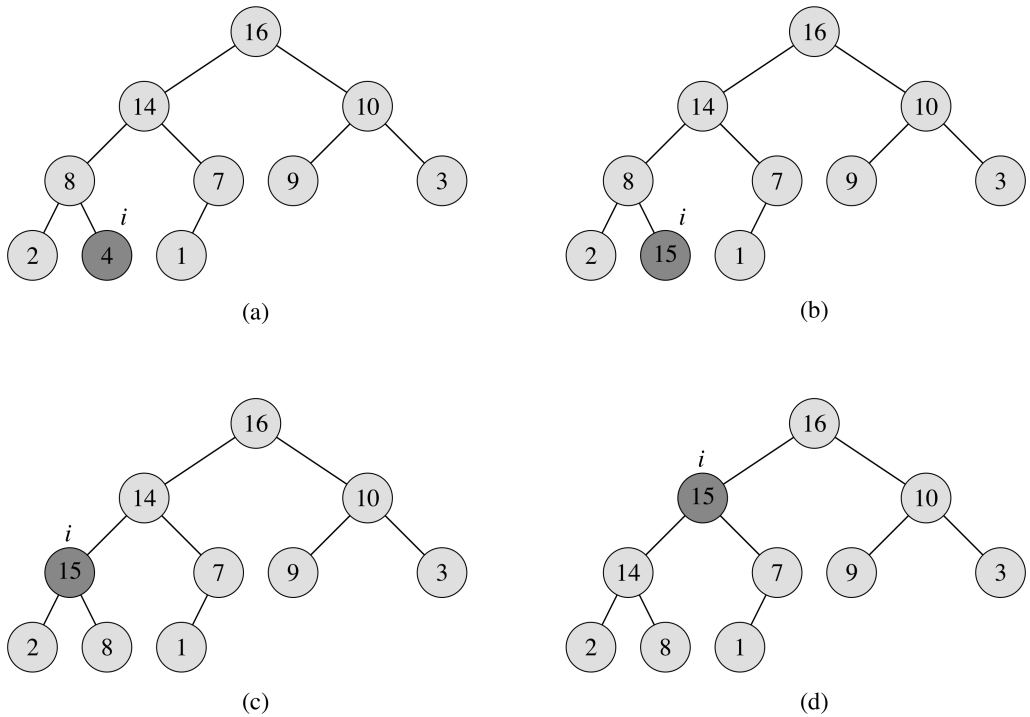
- Extract-Max: Er essentielt hvad der bruges under anden del af Heapsort – fjern rod, flyt sidste blad op som rod, kald Heapify. Køretid: $O(\log n)$.
- Build: Brug Heapify gentagne gange bottom-up. Køretid: $O(n)$.

Mangler:

- Insert
- Increase-Key

Increase-Key

1. Ændre nøgle for element.
2. Genopret heaporden: sålænge elementet er større end forælder, skift plads med denne.



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: sålænge elementet er større end forælder, skift plads med denne.

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

Forskellige implementationer af prioritetskøer

	Heap	Usorteret liste	Sorteret liste
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 operationer er for max-prioritetskøer. Der er naturligvis nemt at lave min-prioritetskøer med operationerne Extract-Min, Build, Decrease-Key og Insert, blot ved at vende alle uligheder mellem nøgler i definitioner og algoritmer.