Prioritetskøer

- Binære heaper
- Venstrevridde heaper (Leftist)
- Skeive heaper (Skew)
- Binomialheaper
- Fibonacciheaper

Prioritetskøer er viktige i bla. operativsystemer (prosesstyring i multitaskingssystemer), og søkealgoritmer (A, A*, D*, etc.).

Prioritetskøer

Prioritetskøer er datastukturer som holder elementer med en prioritet (key) i en kø-aktig struktur, og som implementerer følgende operasjoner:

- **`insert()`** – Legge et element inn i køen
- **`deleteMin()`** – Ta ut elementet med høyest prioritet

Og kanskje også:

- **`buildHeap()`** – Lage en kø av en mengde elementer
- **`increaseKey()/DecreaseKey()`** – Endre prioritet
- **`delete()`** – Slette et element
- **`merge()`** – Slå sammen to prioritetskøer

Prioritetskøer

En usortert lenket liste kan brukes. **`insert()`** legger inn først i listen ($O(1)$) og **`deleteMin()`** søker igjennom listen etter elementet med høyest prioritet ($O(n)$).

En sortert liste kan brukes. (Omvendt kjøretid.)

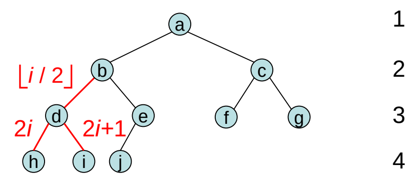
– Ikke så veldig effektivt.

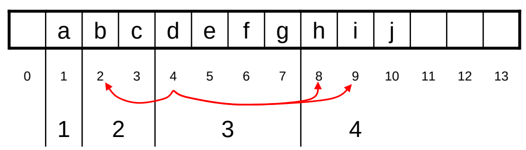
For å lage en effektiv prioritetskø, holder det at elementene i køen er ”nogenlunde sorterte”.

Binære heaper (vanligst)

En *binærheap* er organisert som et komplett binærtre. (Alle nivåer fulle, med evt. unntak av det siste nivået)

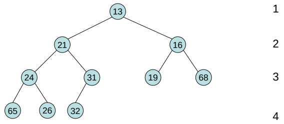
I en *binærheap* skal elementet i roten ha en key som er mindre eller lik key'en til barna, i tillegg skal hvert deltre være en binærheap.





Binære heaper (vanligst)

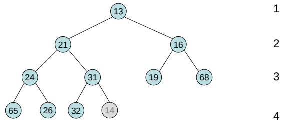
insert(14)



	13	21	16	24	31	19	68	65	26	32			
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

Binære heaper (vanligst)

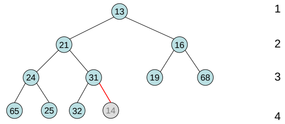
insert(14)



	13	21	16	24	31	19	68	65	26	32	14		
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

Binære heaper (vanligst)

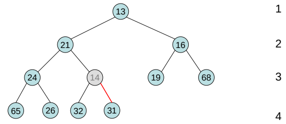
insert(14)



	13	21	16	24	31	19	68	65	26	32	14		
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

Binære heaper (vanligst)

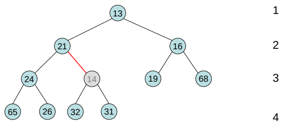
insert(14)



	13	21	16	24	14	19	68	65	26	32	31		
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

Binære heaper (vanligst)

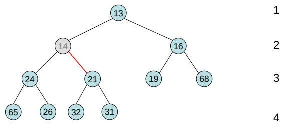
insert(14)



	13	21	16	24	14	19	68	65	26	32	31		
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

Binære heaper (vanligst)

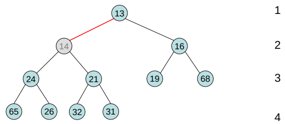
insert(14)



	13	14	16	24	21	19	68	65	26	32	31		
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

Binære heaper (vanligst)

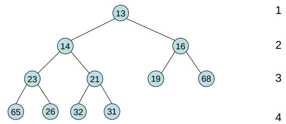
insert(14)



	13	14	16	24	21	19	68	65	26	32	31		
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

Binære heaper (vanligst)

insert(14)

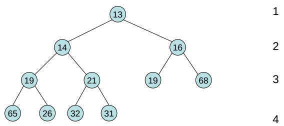


	13	14	16	24	21	19	68	65	26	32	31		
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

"**percolateUp()**"

Binære heaper (vanligst)

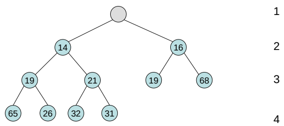
deleteMin()



	13	14	16	19	21	19	68	65	26	32	31		
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

Binære heaper (vanligst)

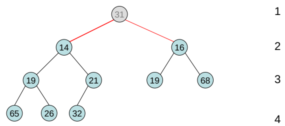
deleteMin()



		14	16	19	21	19	68	65	26	32	31		
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

Binære heaper (vanligst)

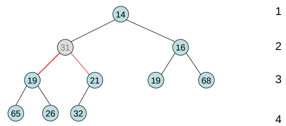
deleteMin()



	31	14	16	19	21	19	68	65	26	32			
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

Binære heaper (vanligst)

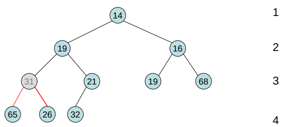
deleteMin()



	14	31	16	19	21	19	68	65	26	32			
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

Binære heaper (vanligst)

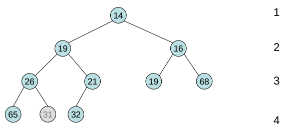
deleteMin()



	14	19	16	19	21	31	68	65	26	32			
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

Binære heaper (vanligst)

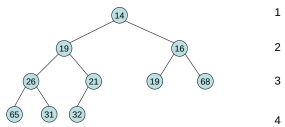
deleteMin()



	14	19	16	19	21	26	68	65	31	32			
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

Binære heaper (vanligst)

deleteMin()



	14	19	16	19	21	26	68	65	31	32			
0	1	2	3	4	5	6	7	8	9	10	11	12	13
	1	2			3				4				

"**percolateDown()**"

Binære heaper (vanligst)

	W.C.	Avg C.
insert()	$O(\log N)$	$O(1)$
deleteMin()	$O(\log N)$	$O(\log N)$

buildHeap() $O(N)$

(Legg elementene inn i tabellen i vilkårlig rekkefølge, og kjør **percolateDown()** på hver rot i deltrærne i heapen (treet) som oppstår, nedenifra og opp.)

(Summen av høydene i et binærtre med N noder er $O(N)$.)

merge() $O(N)$

(N = antall elementer)

Venstrevridde heaper

For å implementere **`merge()`** effektivt, går vi bort fra tabell-metoden, og ser på såkalte *venstrevridde heaper* (*leftist heaps*), som rene trær.

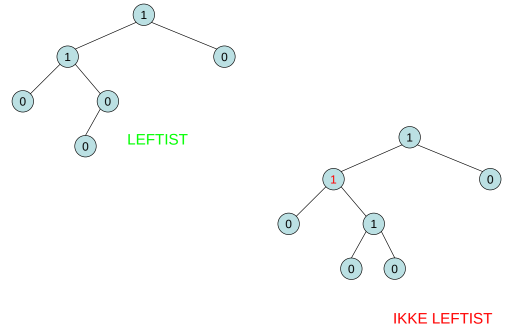
Tanken er å gjøre heapen (treet) skeivt, slik at vi kan gjøre det meste av arbeidet på den laveste delen av treet.

En *venstrevridd heap* er et binærtre med heap-struktur (røttene i deltrærne skal ha lavere key enn barna.) og et ekstra *skeivhetskrav*.

For alle noder *X* i treet, definerer vi *nullsti-lengede*(*X*) (*null path length*) som lengden av korteste sti fra *X* til en node som ikke har to barn.

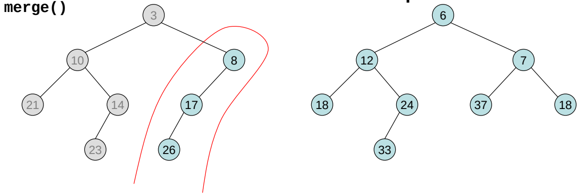
Kravet er at for alle noder, skal nullsti-lengden til det venstre barnet være minst like stor som nullsti-lengden til det høyre barnet.

Venstrevridde heaper



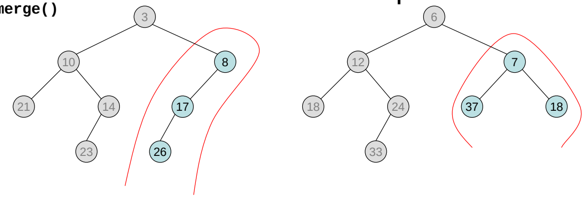
Venstrevridde heaper

`merge()`



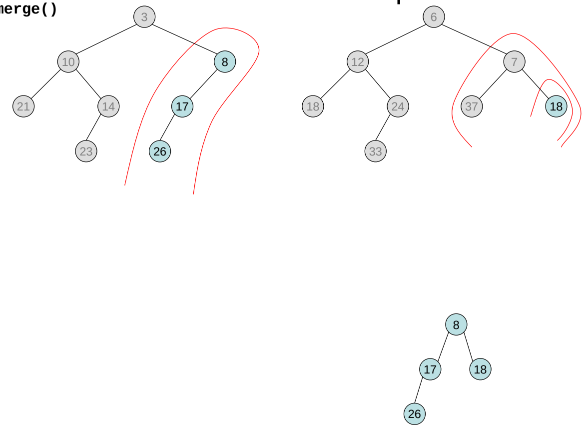
Venstrevridde heaper

`merge()`



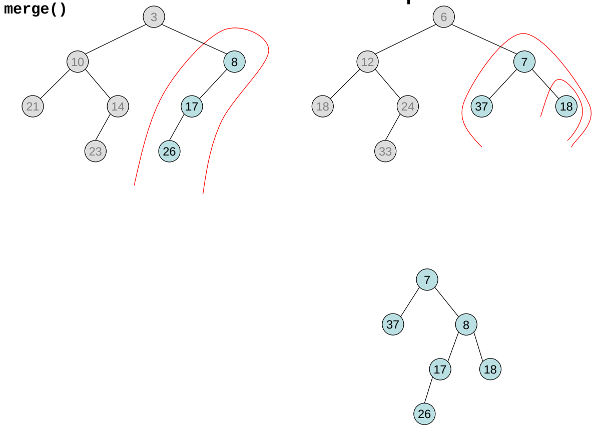
Venstrevridde heaper

`merge()`



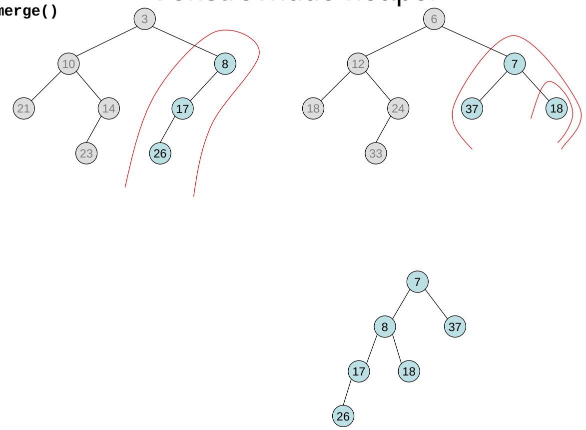
Venstrevridde heaper

`merge()`



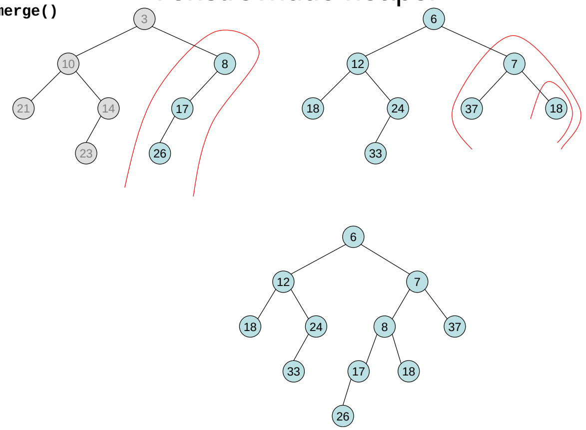
Venstrevridde heaper

`merge()`



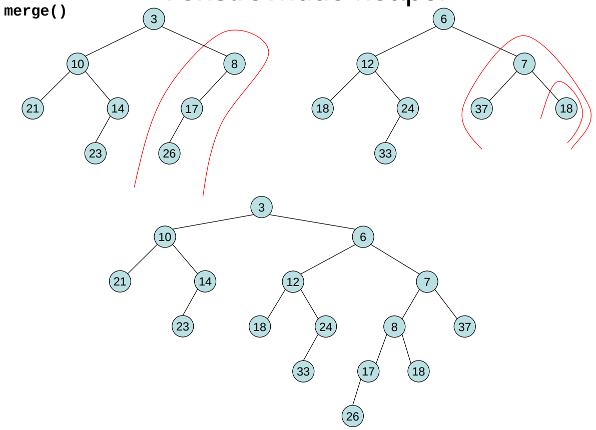
Venstrevridde heaper

`merge()`



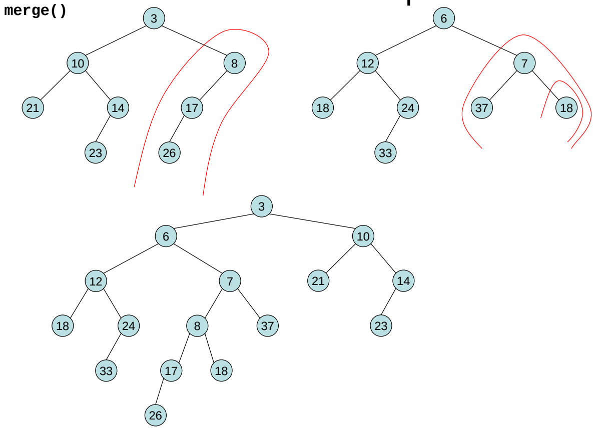
Venstrevridde heaper

`merge()`



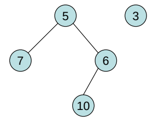
Venstrevridde heaper

`merge()`



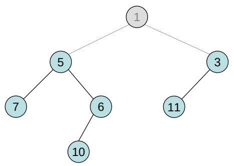
Venstrevridde heaper

`insert(3)` `merge()`



`deleteMin()`

`merge()`



Venstrevridde heaper

	W.C.
`merge()`	$O(\log N)$
`insert()`	$O(\log N)$
`deleteMin()`	$O(\log N)$
`buildHeap()`	$O(N)$

(N = antall elementer)

I venstrevridde heaper med N noder, er høyre sti maksimalt $\lfloor \log (N+1) \rfloor$ lang.

Skeive heaper

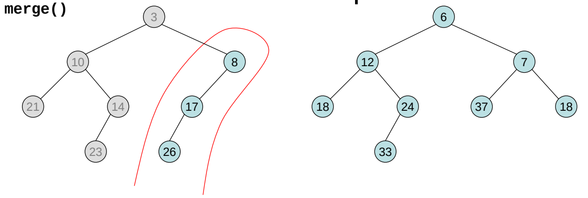
Skeive heaper (*skew heaps*) er en selvbalanserende variant av venstrevridde heaper.

Vi har ikke lengre noe eksplisitt skeivhetskrav, som for de venstrevridde heapene, og vi vedlikeholder ingen nullsti-lengde, heapen er selvjusterende (analogt med AVL/Splay-trær).

For skeive heaper er **`merge()`**-rutinen den samme som for venstrevridde heaper, men vi bytter <u>alltid</u> om på barna, bortsett fra noden med høyest key i de høyre stiene (denne har ikke noe høyre barn).

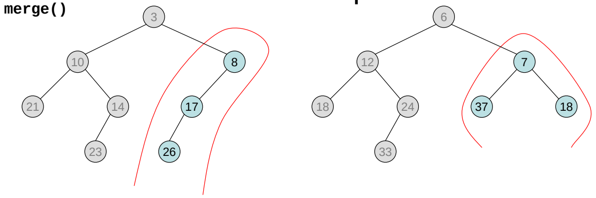
Skeive heaper

`merge()`



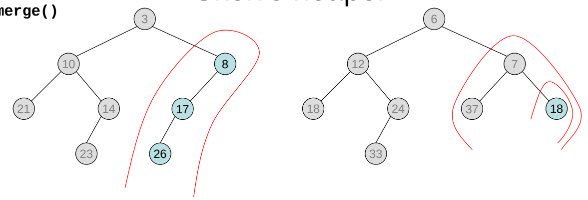
Skeive heaper

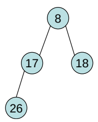
`merge()`



Skeive heaper

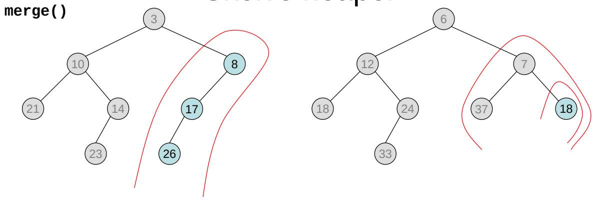
`merge()`

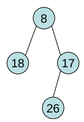




Skeive heaper

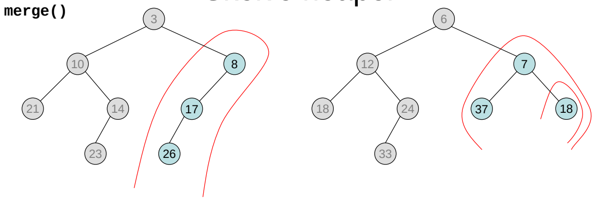
`merge()`

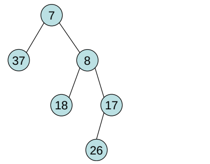




Skeive heaper

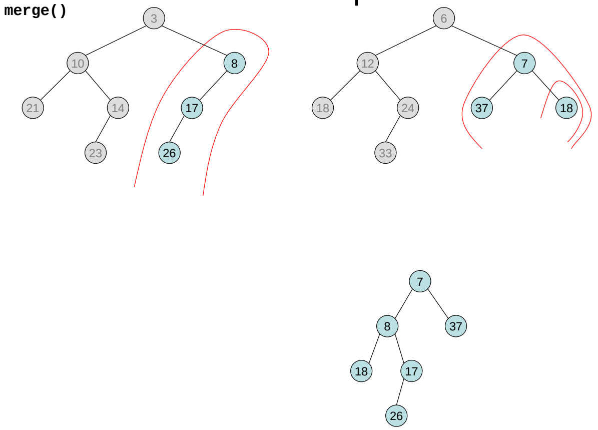
`merge()`





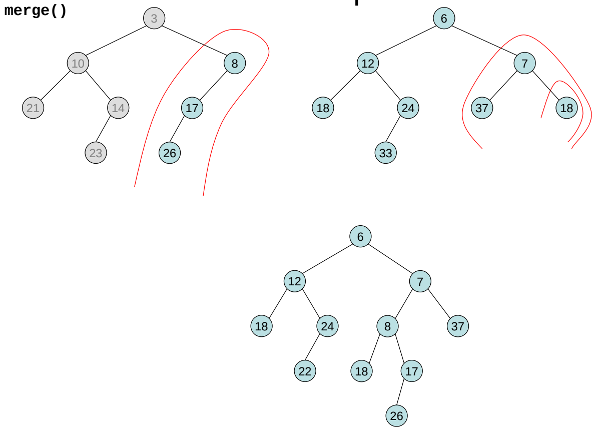
Skeive heaper

merge()



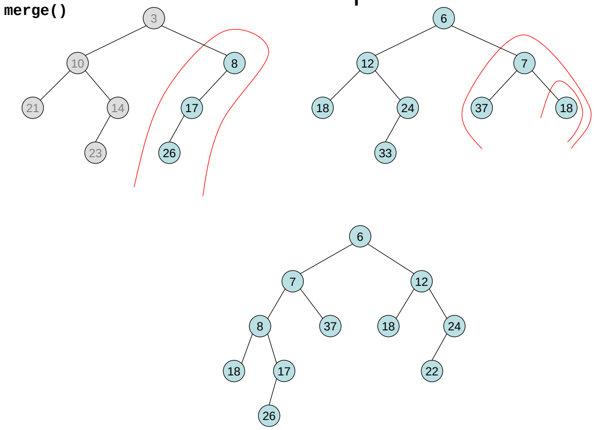
Skeive heaper

merge()



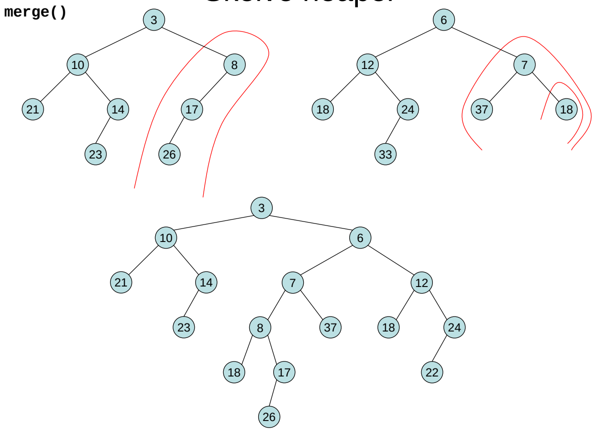
Skeive heaper

merge()



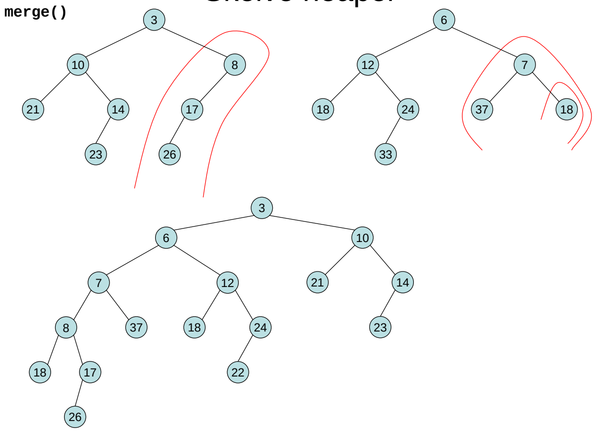
Skeive heaper

merge()



Skeive heaper

merge()



Binomialheaper

Venstrevridde / skeive heaper:

`merge()`, **`insert()`** og **`deleteMin()`** i tid $O(\log N)$.

Binære heaper:

`insert()` i tid $O(1)$ i gjennomsnitt.

Binomialheaper

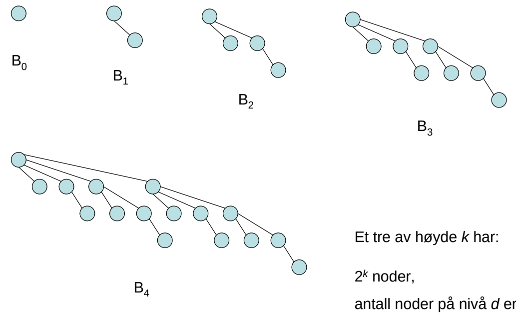
`merge()`, **`insert()`** og **`deleteMin()`** i tid $O(\log N)$ W.C.

`insert()` i tid $O(1)$ i gjennomsnitt.

Binomialheaper er ikke trær, men en skog av trær, hvert tre en heap.

Binomialheaper

Binomialtrær



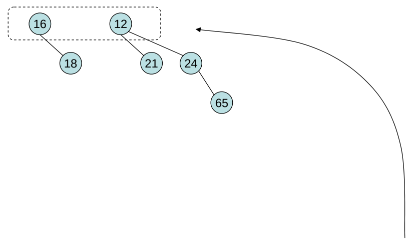
Et tre av høyde k har:

2^k noder,

antall noder på nivå d er $\binom{k}{d}$

Binomialheaper

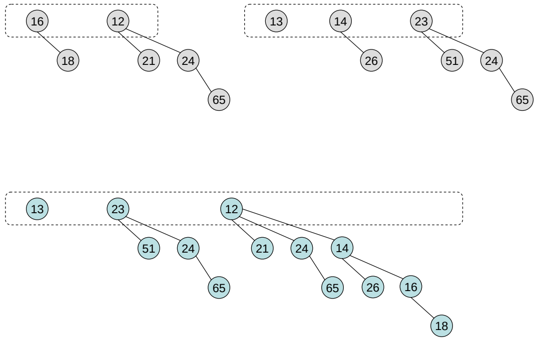
Binomialheap



Lengden av rot-listen for en heap med N elementer er $O(\log N)$.
(Dobbelt lenket, sirkulær liste.)

Binomialheaper

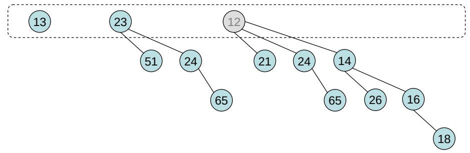
`merge()`



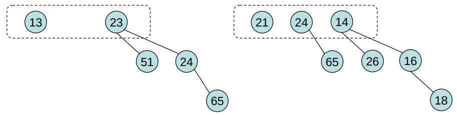
Trærne (rotlisten) holdes sortert etter høyde.

Binomialheaper

`deleteMin()`



`merge()`



Binomialheaper

	W.C.	Avg C.
`merge()`	$O(\log N)$	$O(\log N)$
`insert()`	$O(\log N)$	$O(1)$
`deleteMin()`	$O(\log N)$	$O(\log N)$
`buildHeap()`	$O(N)$	$O(N)$

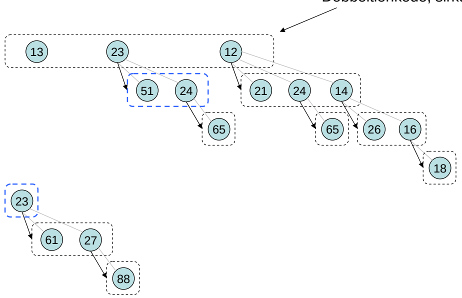
(Kjør N **`insert()`** i en tom heap.)

(N = antall elementer)

Binomialheaper

Implementsjon

Dobbeltlenkede, sirkulære lister



Fibonacciheaper

Meget elegant, og i teorien effektiv, måte å implementere heaper på: De fleste operasjoner har amortisert kjøretid $O(1)$. (Fredman & Tarjan '87)

`insert()`, **`decreaseKey()`** og **`merge()`**	$O(1)$ amortisert tid
`deleteMin()`	$O(\log N)$ amortisert tid

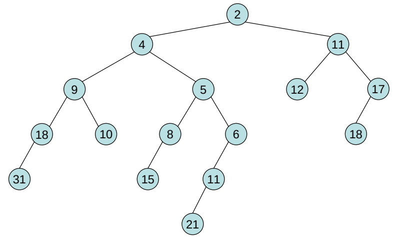
Kombinerer elementer fra venstrevridde heaper og binomialheaper.

I praksis litt komplisert å implementere, og enkelte skjulte konstanter er litt høye.

Best egnet når det er få **`deleteMin()`** i forhold til de andre operasjonene. Datastrukturen utviklet i forbindelse med en korteste sti-algoritme. Også benyttet i spenntre-algoritmer.

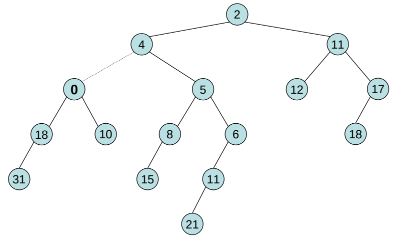
Fibonacciheaper

Fra venstrevridde heaper tar vi med en smart **`decreaseKey()`** metode.



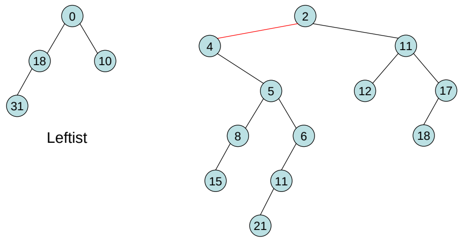
Fibonacciheaper

Fra venstrevridde heaper tar vi med en smart **`decreaseKey()`** metode.



Fibonacciheaper

Fra venstrevridde heaper tar vi med en smart **`decreaseKey()`** metode.

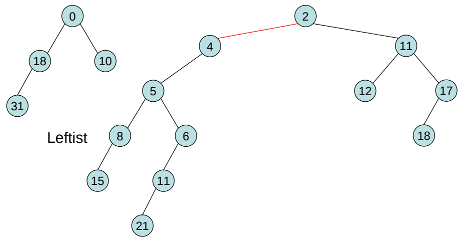


Leftist

Ikke leftist

Fibonacciheaper

Fra venstrevridde heaper tar vi med en smart **`decreaseKey()`** metode.

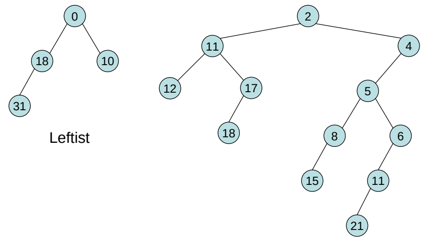


Leftist

Ikke leftist

Fibonacciheaper

Fra venstrevridde heaper tar vi med en smart **`decreaseKey()`** metode.

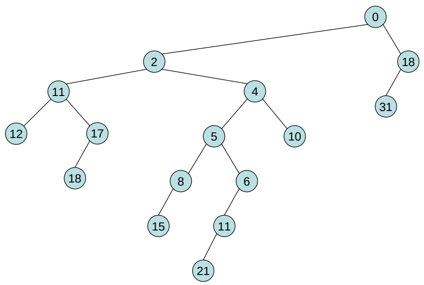


Leftist

Leftist

Fibonacciheaper

Fra venstrevridde heaper tar vi med en smart **`decreaseKey()`** metode.

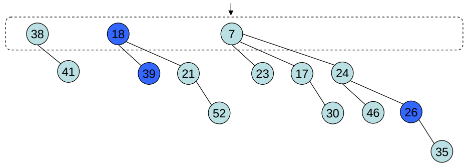


Fibonacciheaper

Fra venstrevridde heaper tar vi med en smart **`decreaseKey()`** metode.

Metoden må imidlertid modifiseres litt, ettersom vi ønsker å bruke trær som nesten er binomialtrær.

- Noder merkes første gang de mister ett barn.
- Andre gang de mister ett barn blir noden kappet av og blir rot i et eget tre, og merket fjernes.

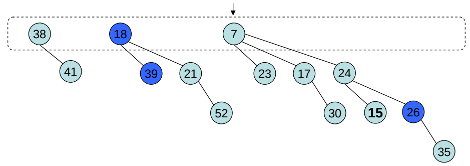


Fibonacciheaper

Fra venstrevridde heaper tar vi med en smart **`decreaseKey()`** metode.

Metoden må imidlertid modifiseres litt, ettersom vi ønsker å bruke trær som nesten er binomialtrær.

- Noder merkes første gang de mister ett barn.
- Andre gang de mister ett barn blir noden kappet av og blir rot i et eget tre, og merket fjernes.

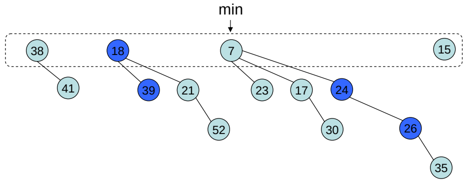


Fibonacciheaper

Fra venstrevridde heaper tar vi med en smart **`decreaseKey()`** metode.

Metoden må imidlertid modifiseres litt, ettersom vi ønsker å bruke trær som nesten er binomialtrær.

- Noder merkes første gang de mister ett barn.
- Andre gang de mister ett barn blir noden kappet av og blir rot i et eget tre, og merket fjernes.

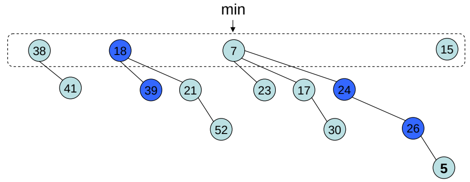


Fibonacciheaper

Fra venstrevridde heaper tar vi med en smart **`decreaseKey()`** metode.

Metoden må imidlertid modifiseres litt, ettersom vi ønsker å bruke trær som nesten er binomialtrær.

- Noder merkes første gang de mister ett barn.
- Andre gang de mister ett barn blir noden kappet av og blir rot i et eget tre, og merket fjernes.

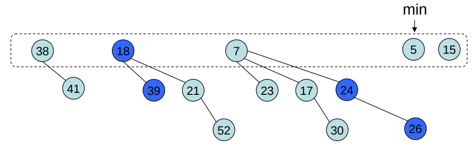


Fibonacciheaper

Fra venstrevridde heaper tar vi med en smart **`decreaseKey()`** metode.

Metoden må imidlertid modifiseres litt, ettersom vi ønsker å bruke trær som nesten er binomialtrær.

- Noder merkes første gang de mister ett barn.
- Andre gang de mister ett barn blir noden kappet av og blir rot i et eget tre, og merket fjernes.

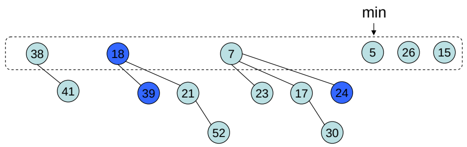


Fibonacciheaper

Fra venstrevridde heaper tar vi med en smart **`decreaseKey()`** metode.

Metoden må imidlertid modifiseres litt, ettersom vi ønsker å bruke trær som nesten er binomialtrær.

- Noder merkes første gang de mister ett barn.
- Andre gang de mister ett barn blir noden kappet av og blir rot i et eget tre, og merket fjernes.

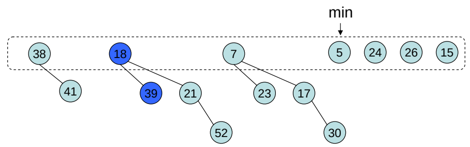


Fibonacciheaper

Fra venstrevridde heaper tar vi med en smart **`decreaseKey()`** metode.

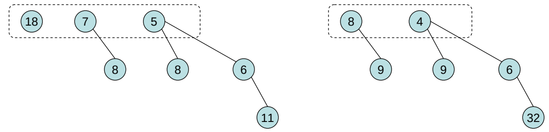
Metoden må imidlertid modifiseres litt, ettersom vi ønsker å bruke trær som nesten er binomialtrær.

- Noder merkes første gang de mister ett barn.
- Andre gang de mister ett barn blir noden kappet av og blir rot i et eget tre, og merket fjernes.



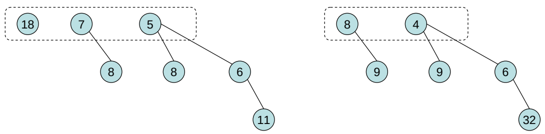
Fibonacciheaper

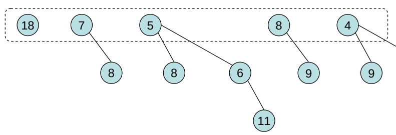
Vi bruker *lazy merging* / *lazy binomial queue*.



Fibonacciheaper

Vi bruker *lazy merging* / *lazy binomial queue*.





Fibonacciheaper

Problemet med **`decreaseKey()`**-metoden vår og *lazy merging* er selvsagt at vi må rydde opp i etterkant. Dette gjøres i **`deleteMin()`**, som derfor får "høy" kjøretid ($O(\log N)$ amortisert tid) .

Alle trærne gjennomgåes, de minste først, og slåes sammen, slik at vi får maks ett tre av hver størrelse. (Hver rot vet hvor mange barn den har.)

Fibonacciheaper

	Amortisert tid
`insert()`	$O(1)$
`decreaseKey()`	$O(1)$
`merge()`	$O(1)$
`deleteMin()`	$O(\log N)$
`buildHeap()`	$O(N)$

(Kjør *N* **`insert()`** i en tom heap.)

(N = antall elementer)