

*“A Comparative Analysis of Three Different
Priority Deques”*

af: Søren Skov & Jesper Holm Olsen

Agenda:

- Hvad er en “Priority Deque”?
- Hvad kan det bruges til?
- De tre datastrukturer:
 - “MinMax-heap”
 - “The Deap” (påpeget fejl!)
 - “Interval Heap”
- Testresultater
- Implementation
- Konklusion
- Videre arbejde

Hvad er en Priority Deque?

En standard prioritets-kø (heap) kan give det største eller det mindste element baseret på en *ordnings-operator*. En "priority-deque" kan give *begge dele*.

Funktioner:

Std. heap	Priority deque
pop()	pop_top() & top() (<i>min-element</i>)
top()	pop_bottom() & bottom() (<i>max-element</i>)

Køretid for std. heaps (C++ standard-krav)

Operation	Kompleksitet
Hent element	$O(1)$
Konstruktion	$O(N)$
Slet element	$O(\log N)$
Indsæt nyt element	$O(\log N)$

Anvendelse

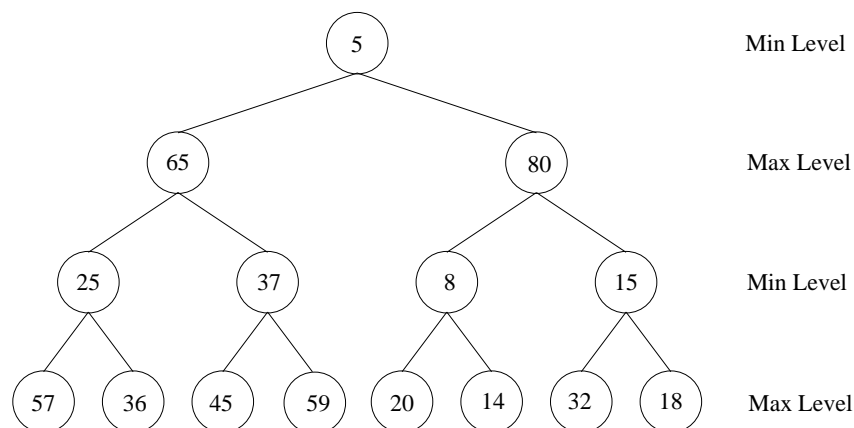
Priority dequeues kan anvendes bla. til følgende:

- Ekstern quicksort
- Konstant-tids "find-median"
- Logaritmisk-tids "slet-median"
- Geometri-problemer

Èn priority deque \neq to standard heaps!

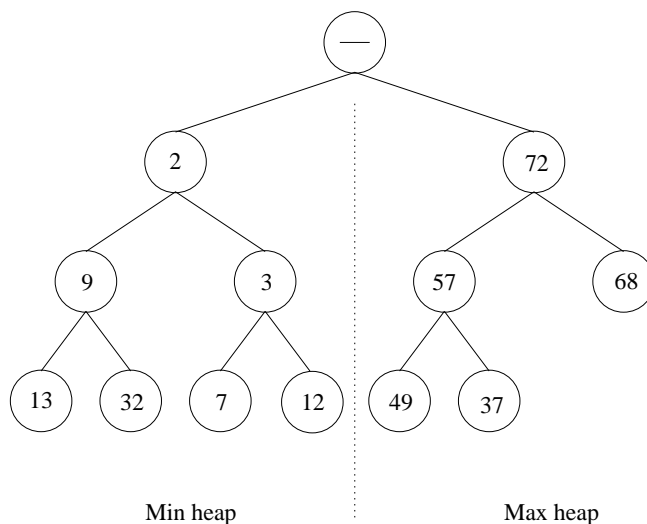
MinMax-heap

- Skiftende min/max-niveauer
- Invariant: Toppen af et min-træ indeholder det mindste element, mens toppen af et max-træ indeholder det største element.
- Min/max-element i toppen eller niveauet under
- Kontruktion: "heapify" nedefra og op
- Ved pop(): Nederste bladknode flyttes op og bliver heapify'et
- Ved Insert(): Indsæt i nederste bladknode, ombyt evt. med forælder, kør op.



“The Deap”

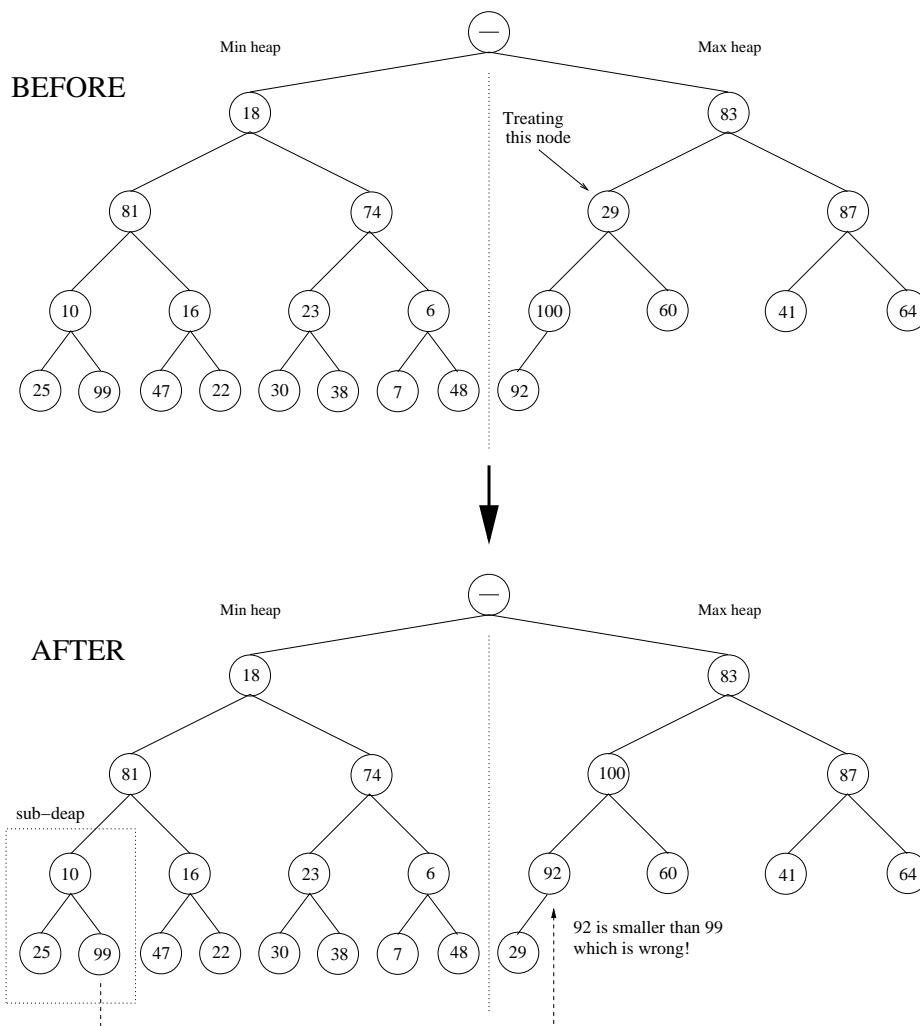
- Intet top-element, min/max-side
- Invariant: En knude i min-træet er mindre end dens *korresponderende* knude i max-træet.
- Min/max-element i toppen af siden
- Ved Insert(): Sammenlign med korresponderende element, køр op.
- Kontruktion: Lav ordning mellem min/max-siden nedefra og op. Fjern element, hiv op, indsæt element i bunden
- Ved pop(): Hiv sidste element op og køр ned



Problemer med Deap'en #1

Glemmer et potentielt swap med et korresponderende element under konstruktion:

Hiv "29" ud, ryk alle elementer op, indsæt "29" i bunden → ødelægger invarianten

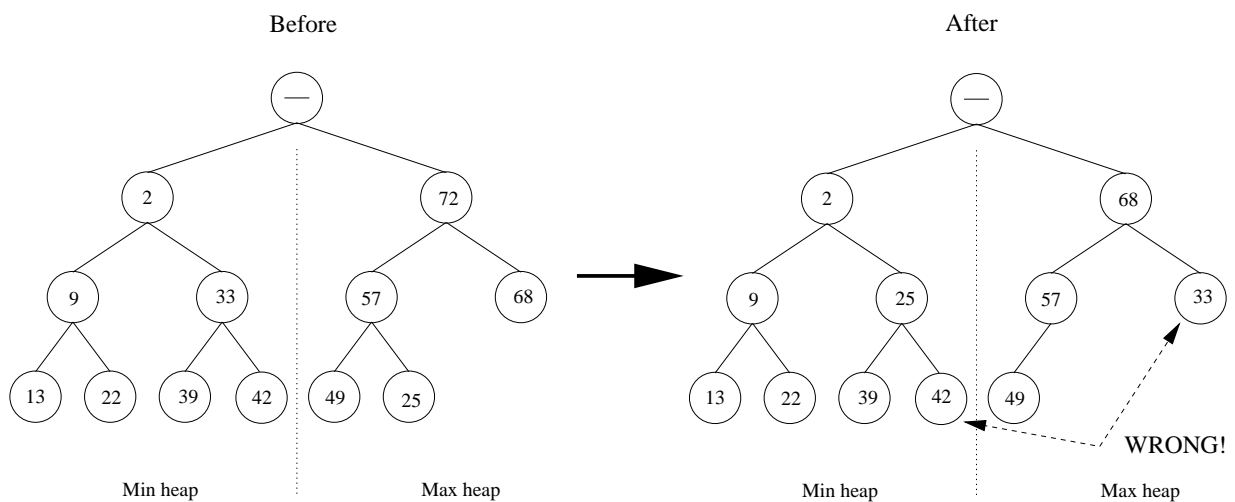


Løsning: Lav et ekstra check

Problemer med Deap'en #2

Vælger det forkerte korresponderende element ved pop() af max-elementet:

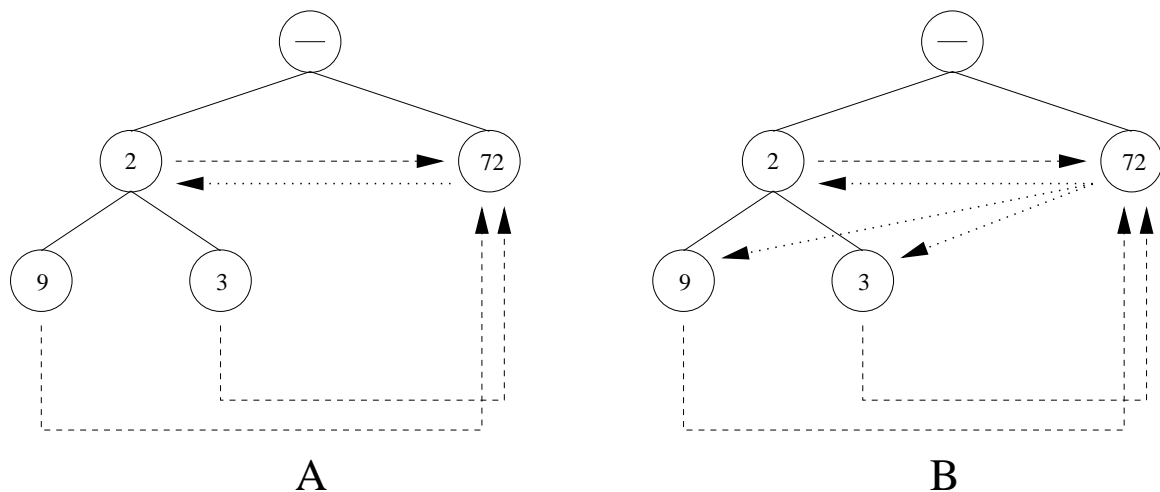
Fjern "72", flyt "25" op og kør ned, ombyt med korresponderende → ødelægger invarianten



Løsning: Check for alle korresponderende

Løsning

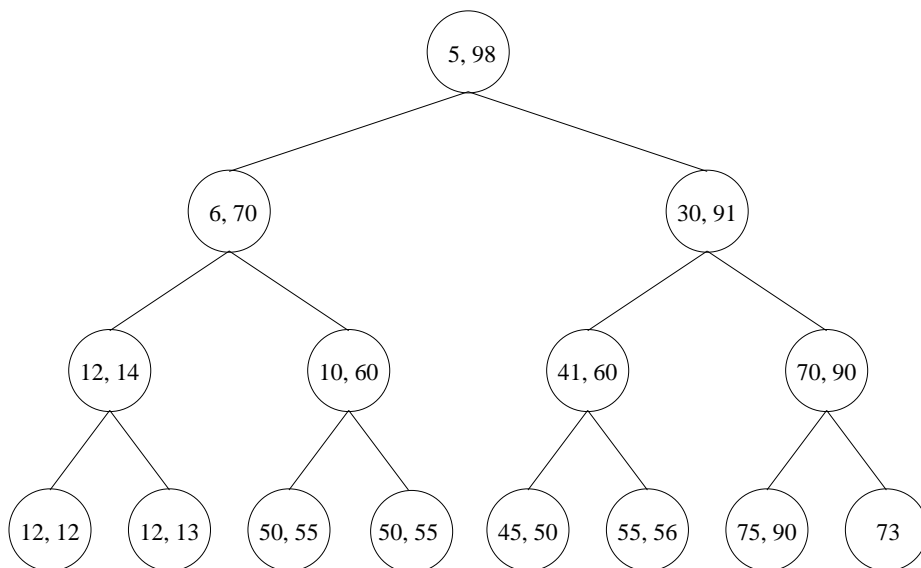
Omdefinér sammenhængen mellem korresponderende elementer:



“Let a be half the width of the level of node k . If a node k is in the min-heap, then the corresponding node of k is $k + a$ if it exists and otherwise $(k + a)/2$. The last situation occurs if the level is not fully occupied. If k is in the max-heap the corresponding elements is the elements in the min-heap that has k as their corresponding element.”

Interval-heaps

- Hver knude er et *interval* $[a, b]$, hvor $a \leq b$
- Invariant: Intervallerne af børnene E og F af en knude X er indeholdt i X's interval: $E \subset X$ og $F \subset X$. Min/max-elementer er derfor i top-knuden.
- Ét eller to elementer i den sidste knude
- Konstruktion: Sammenlign værdier i intervallet, kør "heapify" for min og max.
- Pop(): Fjern element, ryk sidste op, kør heapify
- Insert(): Indsæt på sidste plads, kør op

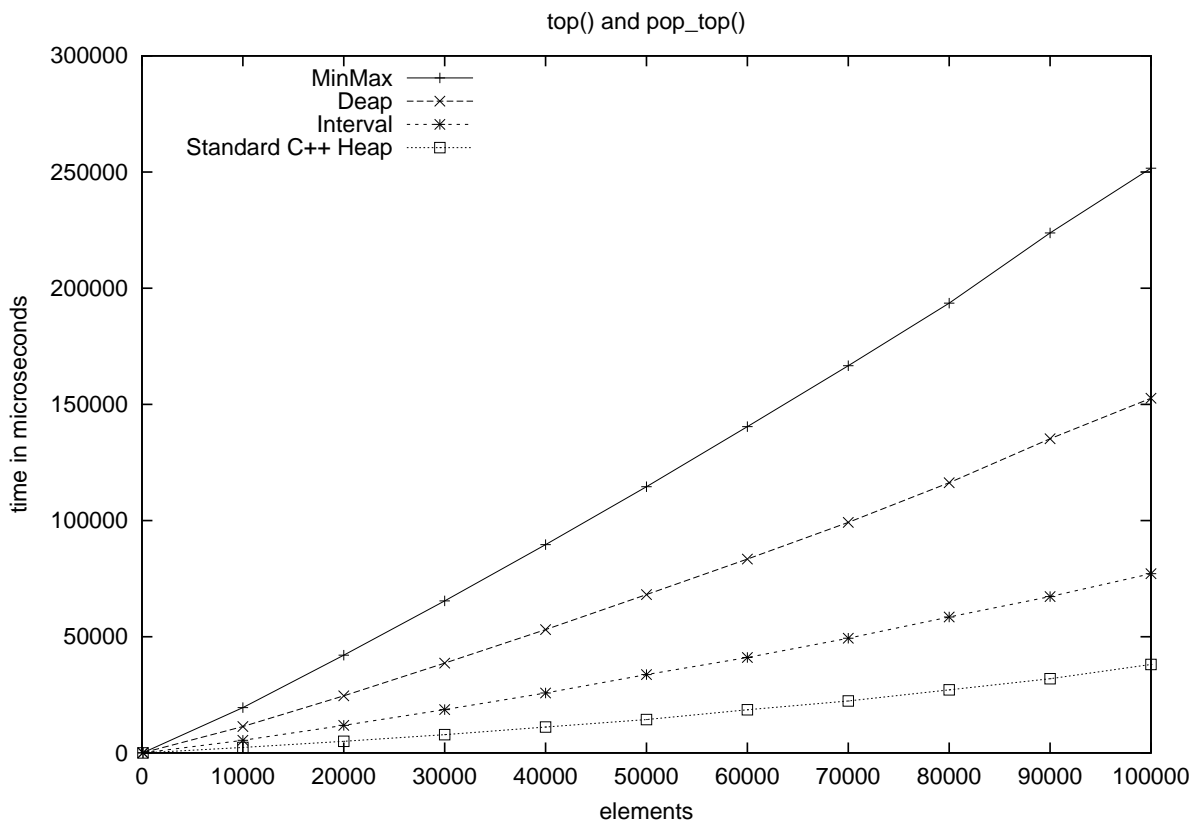


Forventninger til performance

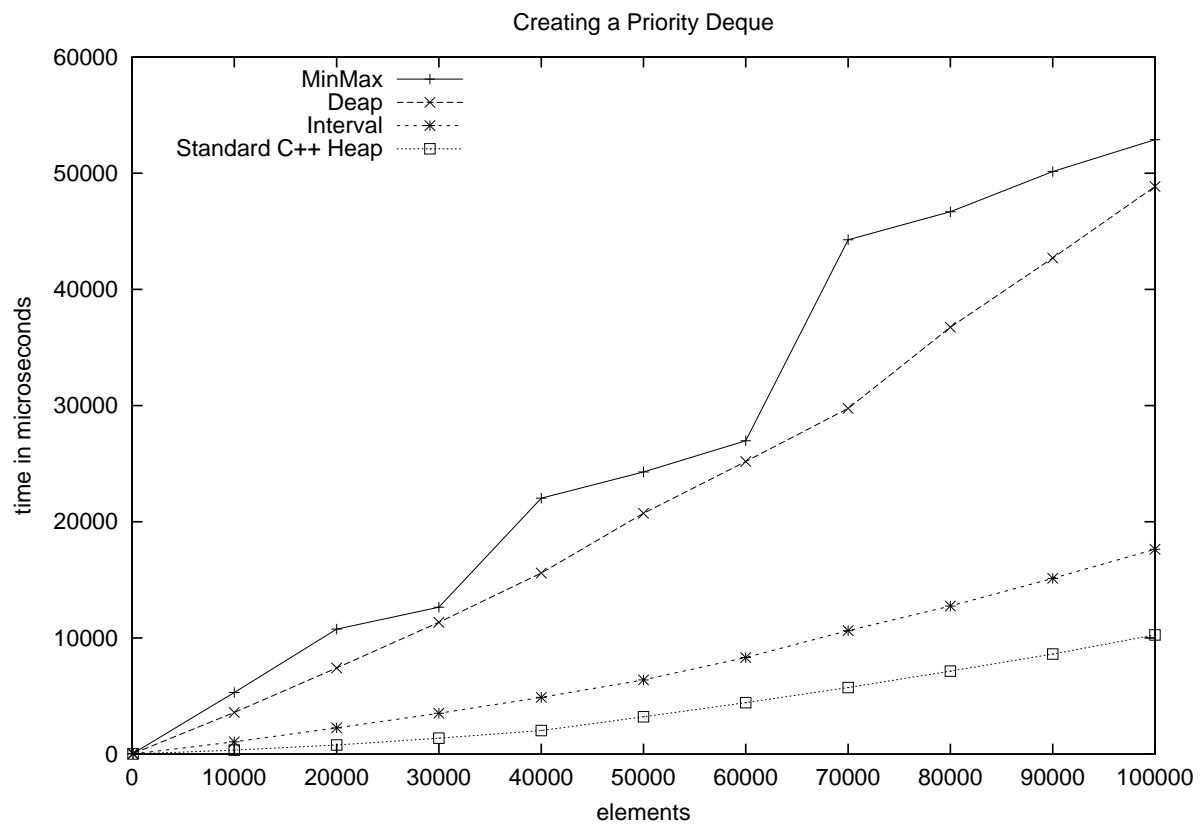
Cache-udnyttelse – hvor mange cachelinier skal de forskellige algoritmer bruge, når de løber op og ned gennem datastrukturen?

- MinMax-heaps – 3 cachelinier
- The Deap – 2 eller 4 cachelinier
- Interval heaps – 2 cachelinier

Testresultater – udtag mindste element



Testresultater – konstruktion



Bemærk springene ved MinMax = level 1 og level 2 cachestørrelse.

Hvorfor også ved halvdelen?

Implementation

- C++ STL-standard
- Konstant-tids log: `#include "log2.h"`
- Templates for de tre algoritmer

Eksempel:

```
#include "prioritydequeue.h"
using namespace PriorityDequeueNS;
int main(int argc, char** argv)
{
    vector<int> inddata = readfile((argv[1]));

    PriorityDequeue<int, std::vector<int>,
                  std::less<int>,
                  MinMax<int, std::vector<int>,
                  std::less<int> > >
    testheap(inddata.begin(), inddata.end());

    testheap.top();
    testheap.pop_top();
    testheap.bottom();
    testheap.pop_bottom();
} // end Main
```

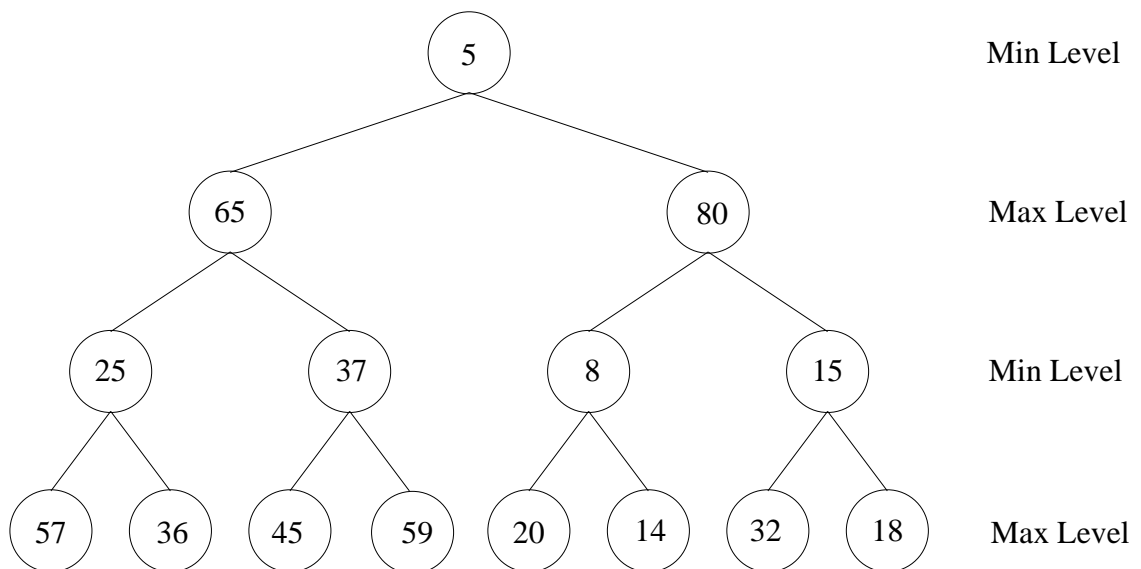
Konklusion

Interval-heaps er bedst, men kun hvis man har brug for Priority-dequeueues. Standard-køer er stadig bedst.

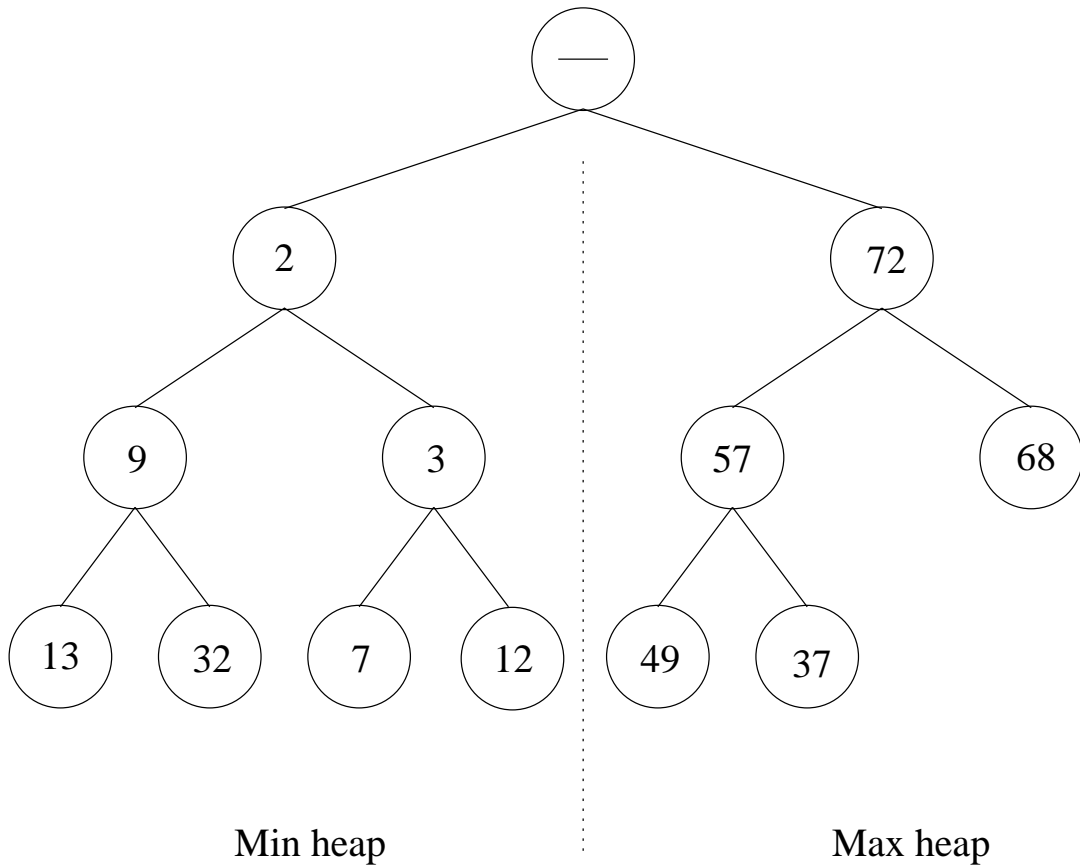
Videre arbejde

- Interval-heaps er gode pga. cachebrug. Put evt. flere elementer i hver knude. Eller lad hver knude have flere børn – eller begge dele.

MinMax



The Deap



Korresponding:

*Let a be half the width of the level of node k .
Then the corresponding node of k is $k + a$ if
it exists and otherwise $(k + a)/2$. The
corresponding node of a node in the max-heap
is computed as $k - a$.*

Interval-heap

