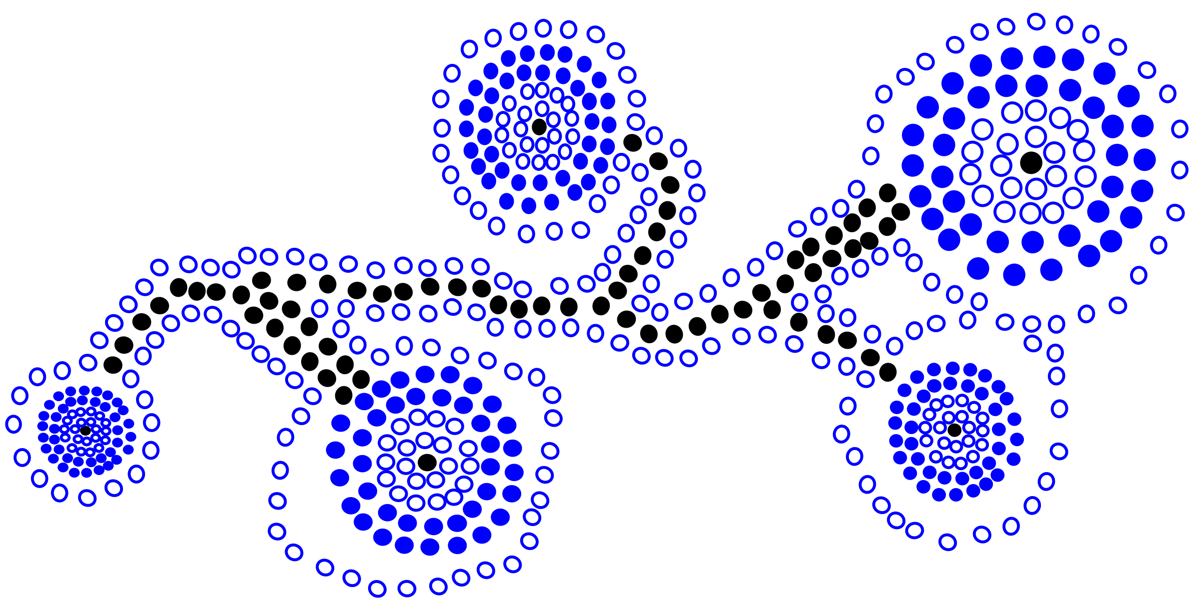
PQ: HEAP

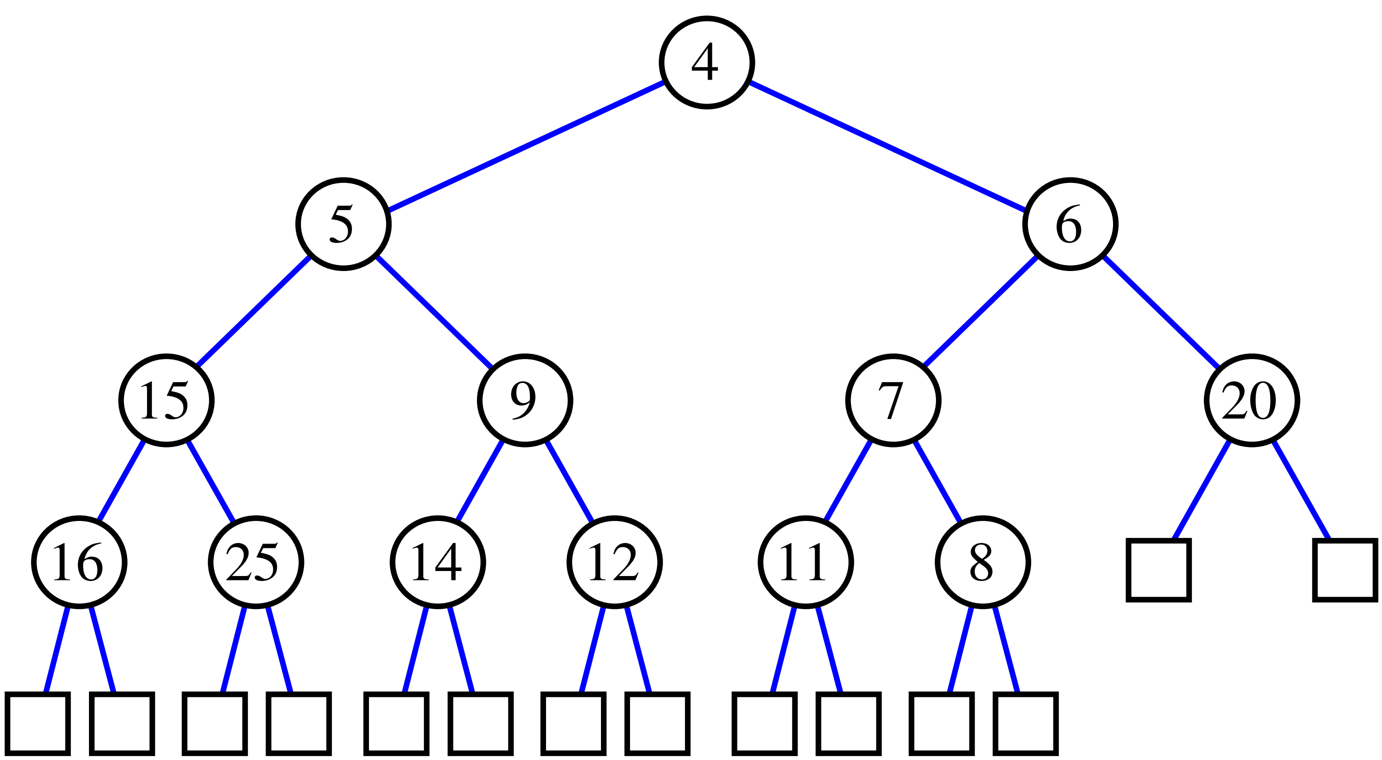
- Ingen sammenheng med ‘memory heap’
- Definisjon og data-invarianter for heap
- InsertKey og RemoveMin for heap
- Kompleksitet for operasjoner: O(log n)
- Prioritetskø impl vha Heap
- HeapSortering i tid O(n log n)
- Bottom-Up Heap Konstruksjon & Locator



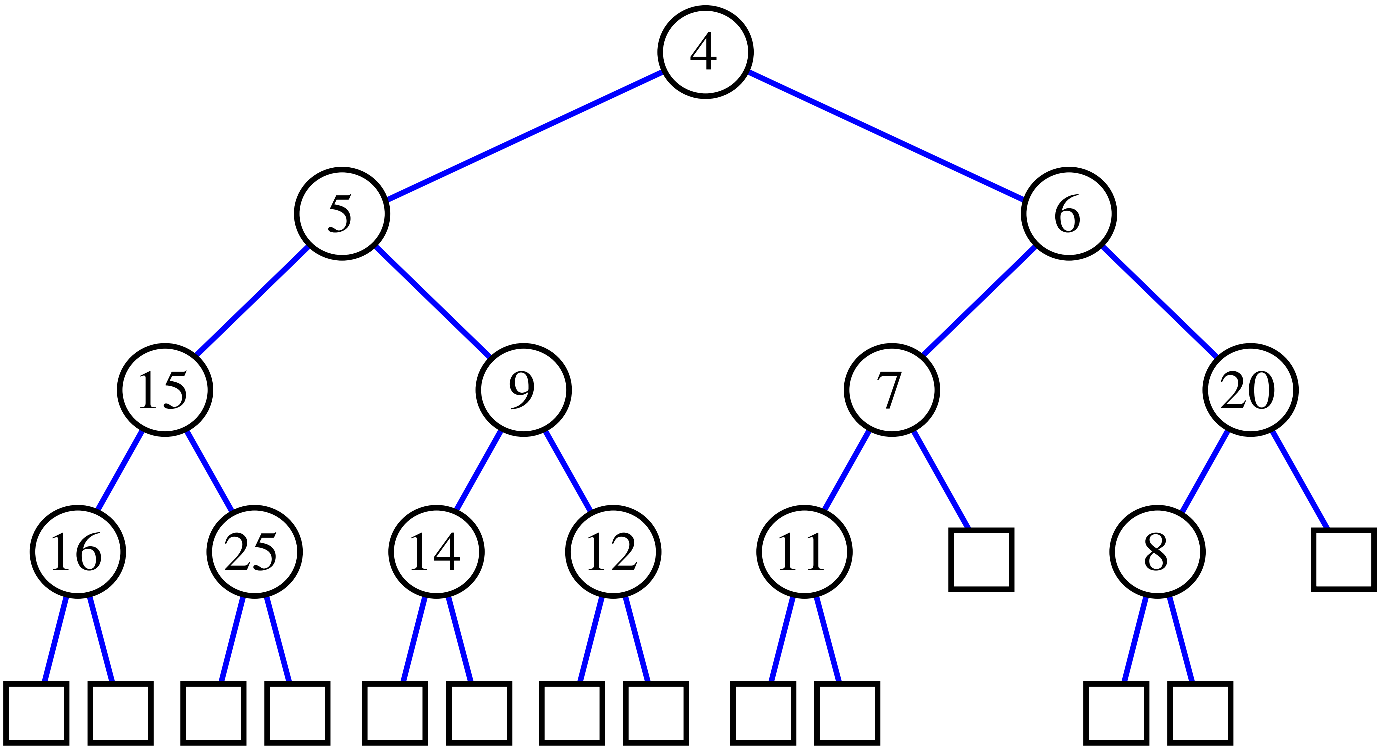
Heap

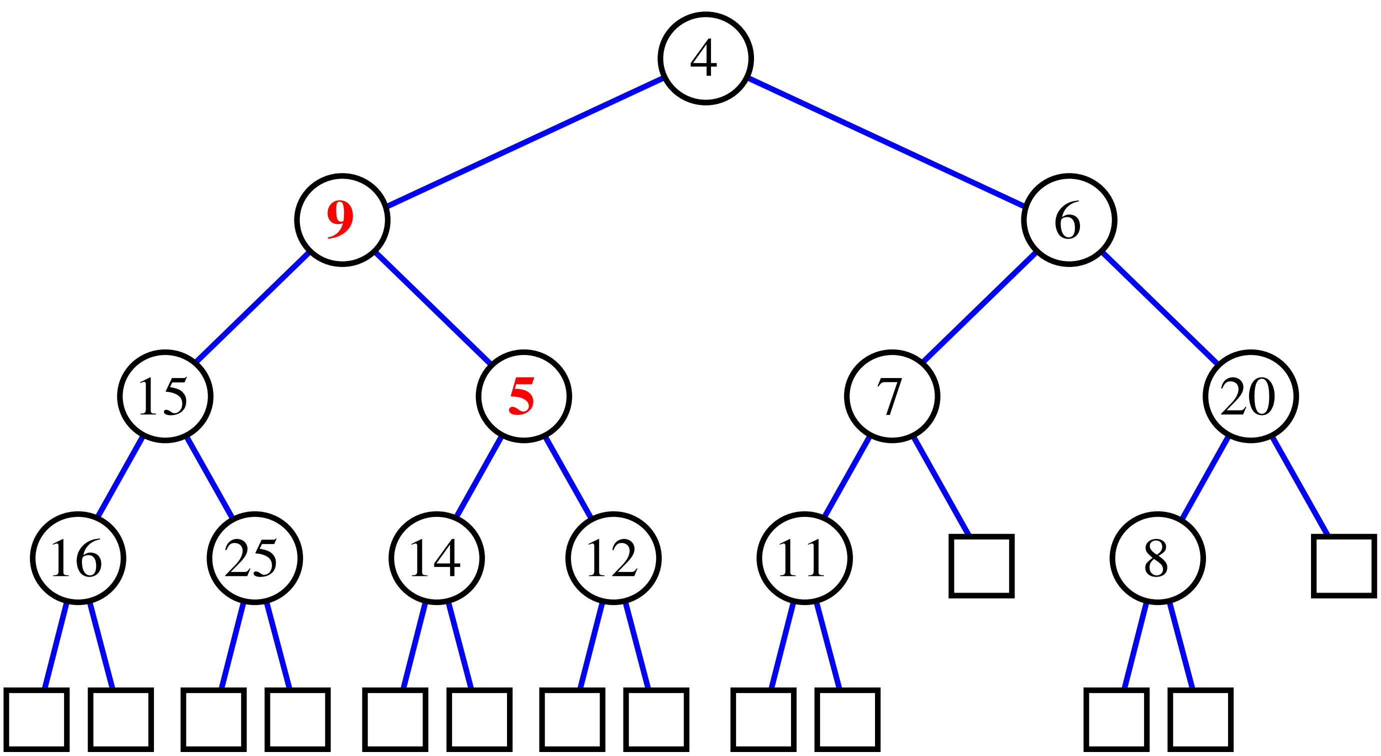
- En ***heap*** er et binært tre *T* som lagrer en samling nøkler (eng: key), eller nøkkel-element par, i sine interne noder og tilfredsstiller 2 krav (2 data-invarianter):

-
  - **HeapOrdning**: key(forelder(node)) ≤ key(node)
  - ***Komplett Binært tre***: alle nivå er fulle, unntatt det siste som er fyllt opp fra venstre mot høyre:



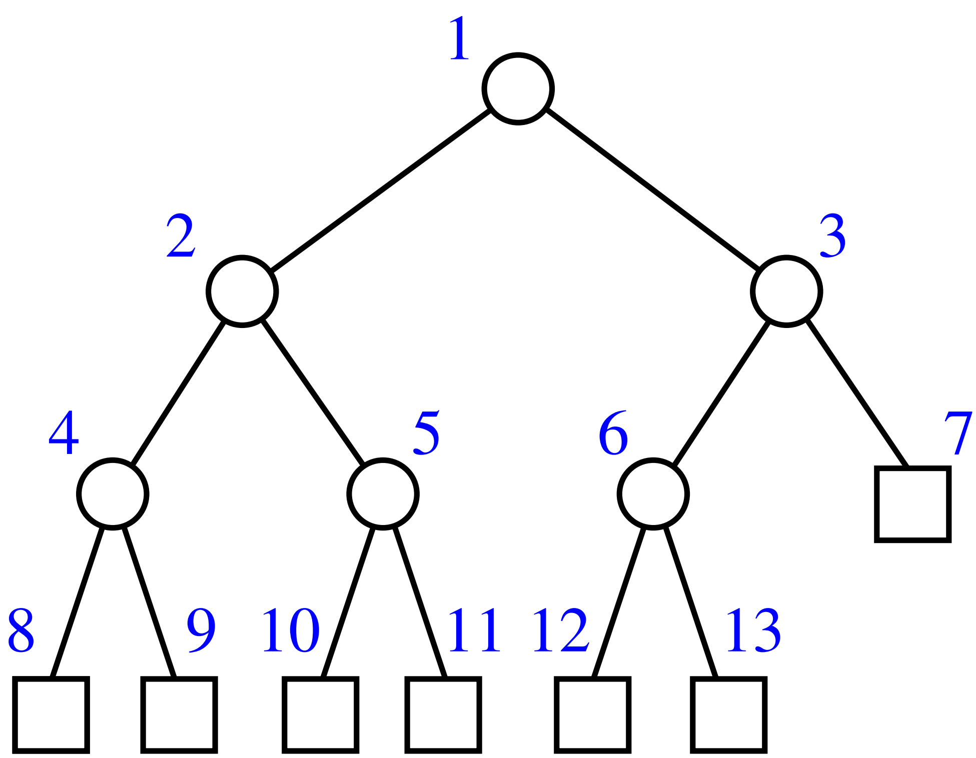
Er disse heap'er?

- 

- 

Hvordan implementere heap:

- NB: Heap er et Komplett Binært Tre.
- Derfor: bruk array direkte (eller mer generellt ADT Sequence) som i implementasjon av Binært Tre i denne ukes gruppeøvelse.

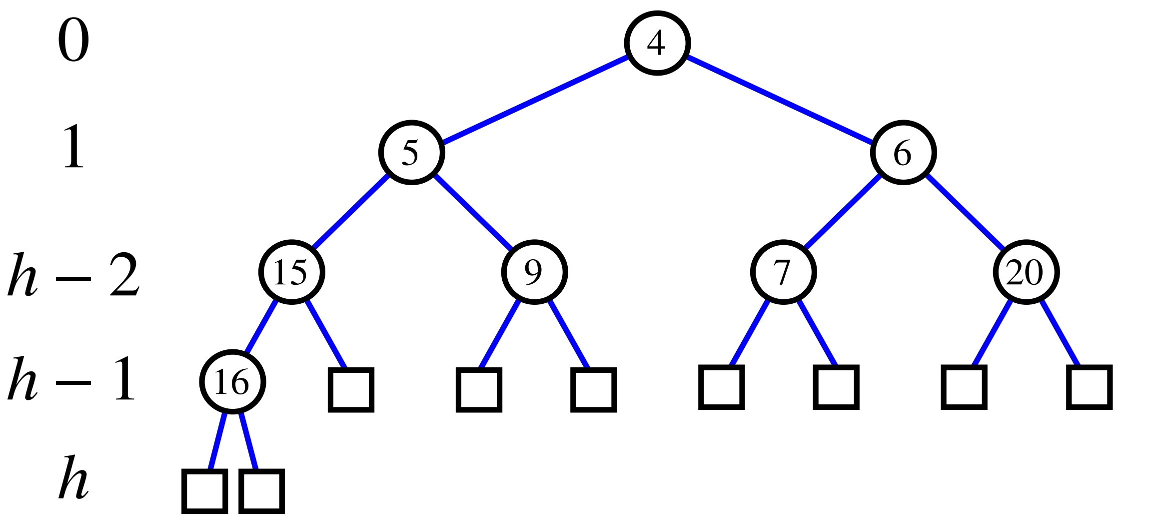


- Løvene kan være implisitte (dvs ikke som i figuren)
- Bruk doblingsvekststrategien for størrelse på array.
- Da er lagringsplass for n elementer O(n) og enhver traversering mellom barn og forelder tar tid O(1).

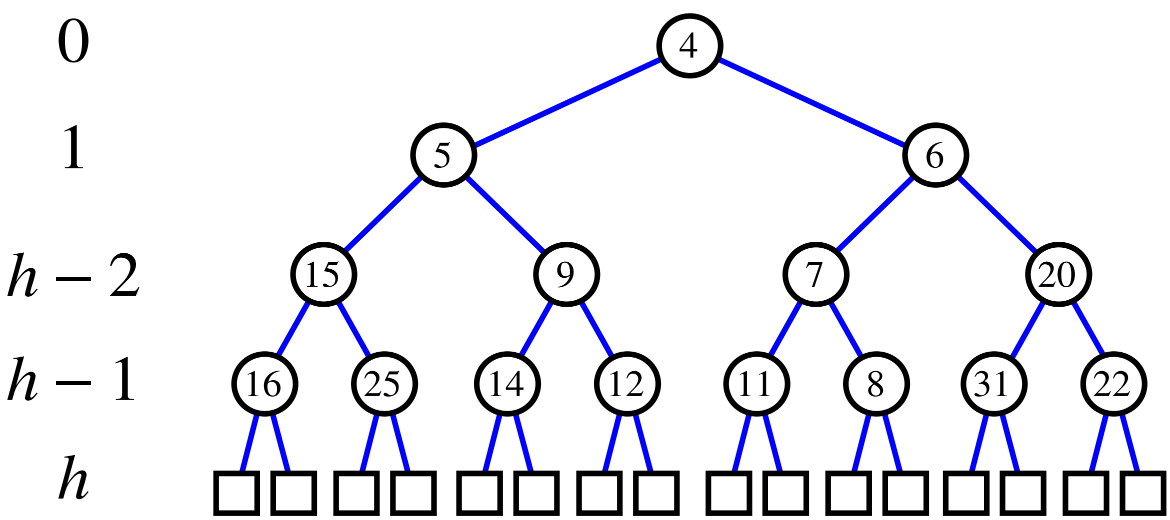
Høyden på en Heap

En heap T som lagrer n nøkler har høyde ....
$h = \lceil \log(n+1) \rceil$, dvs h=O(log n)

- $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1 = 2^{h-1} - 1 + 1 = 2^{h-1}$



- $n \leq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$

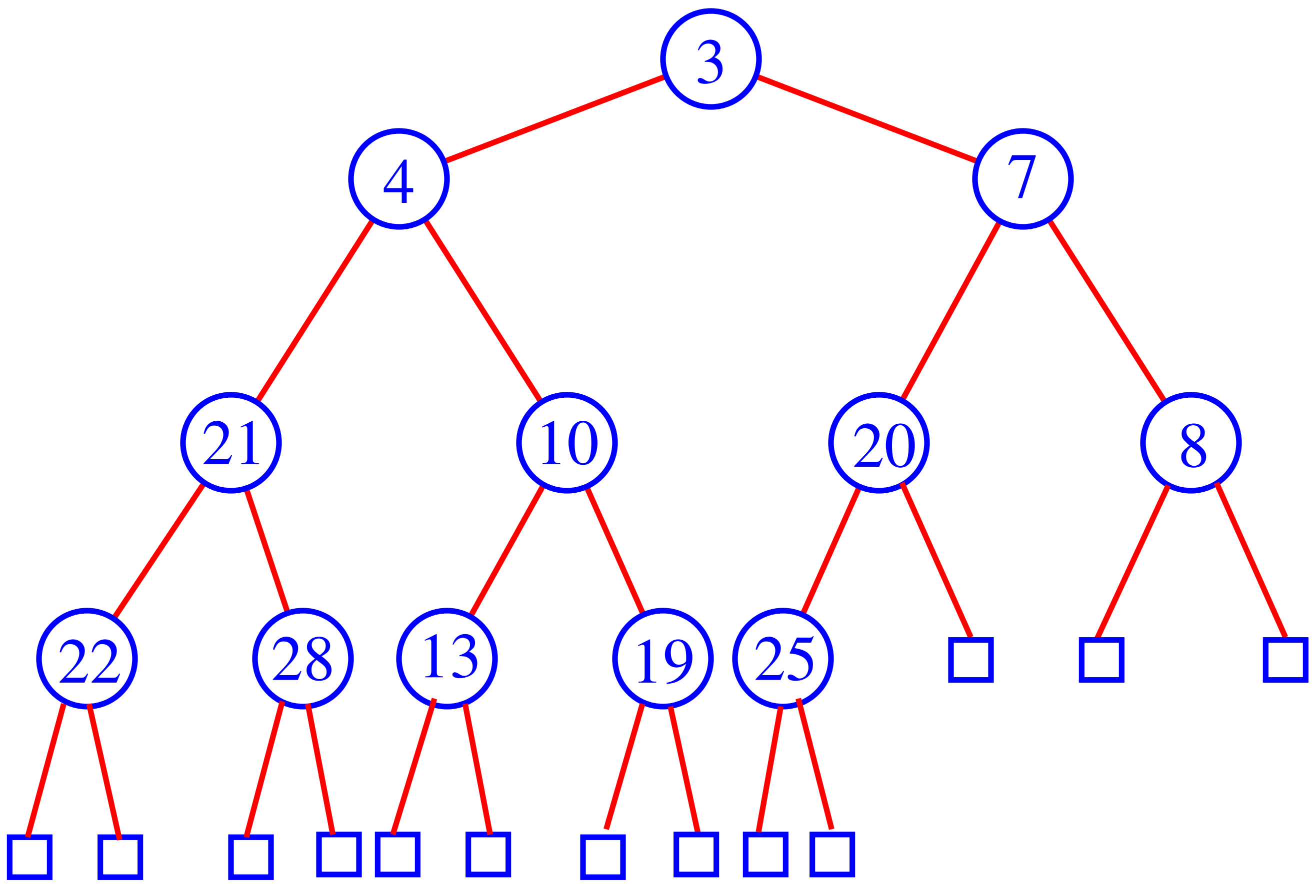


- Derfor har vi $2^{h-1} \leq n \leq 2^h - 1$
- Ta log på begge sider: h-1 ≤ log n ≤ h
- $\log(n+1) \leq h \leq \log n + 1$, gir $\boldsymbol{h = \lceil \log(n+1) \rceil}$

Heap Insertion

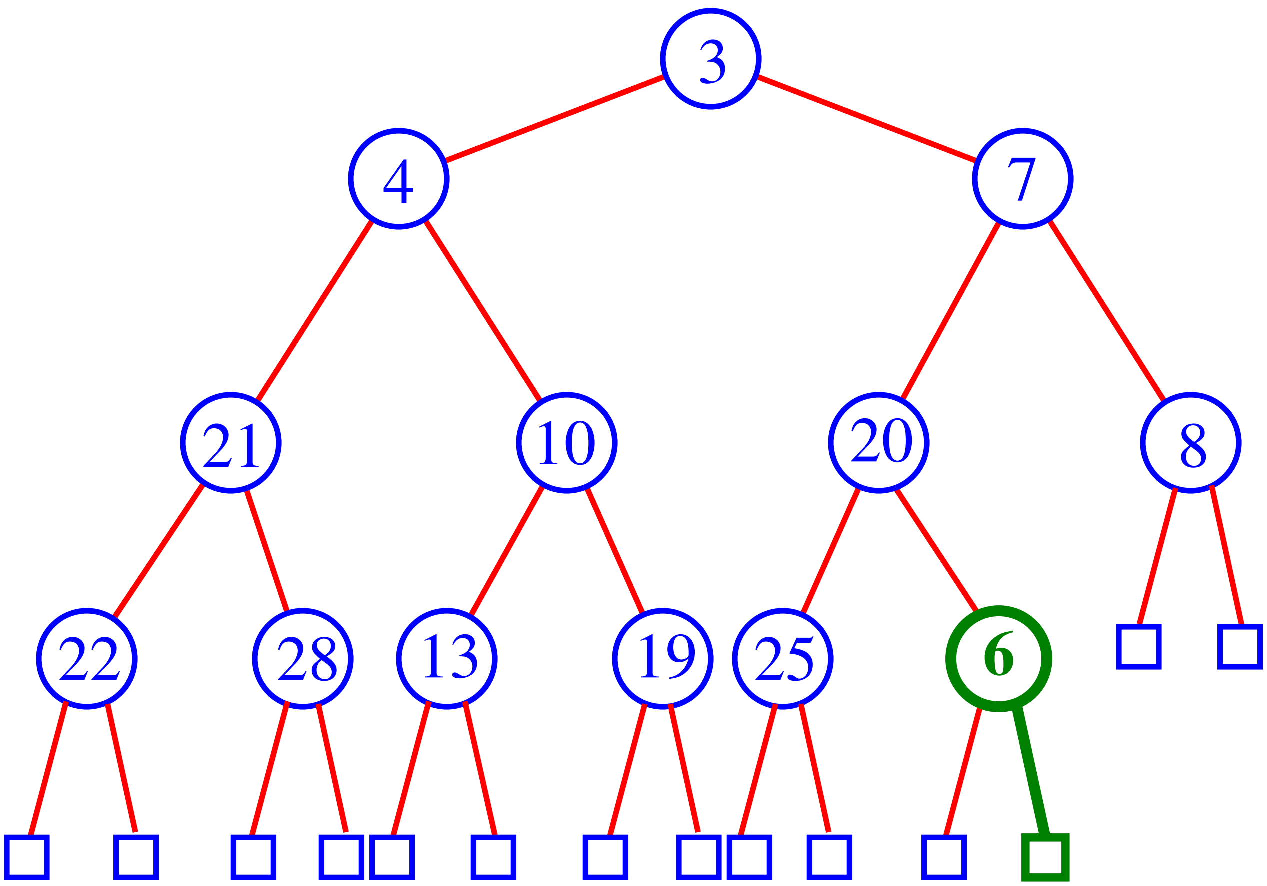
Anta vi har en heap og skal utføre

Insert(6)



Heap Insertion

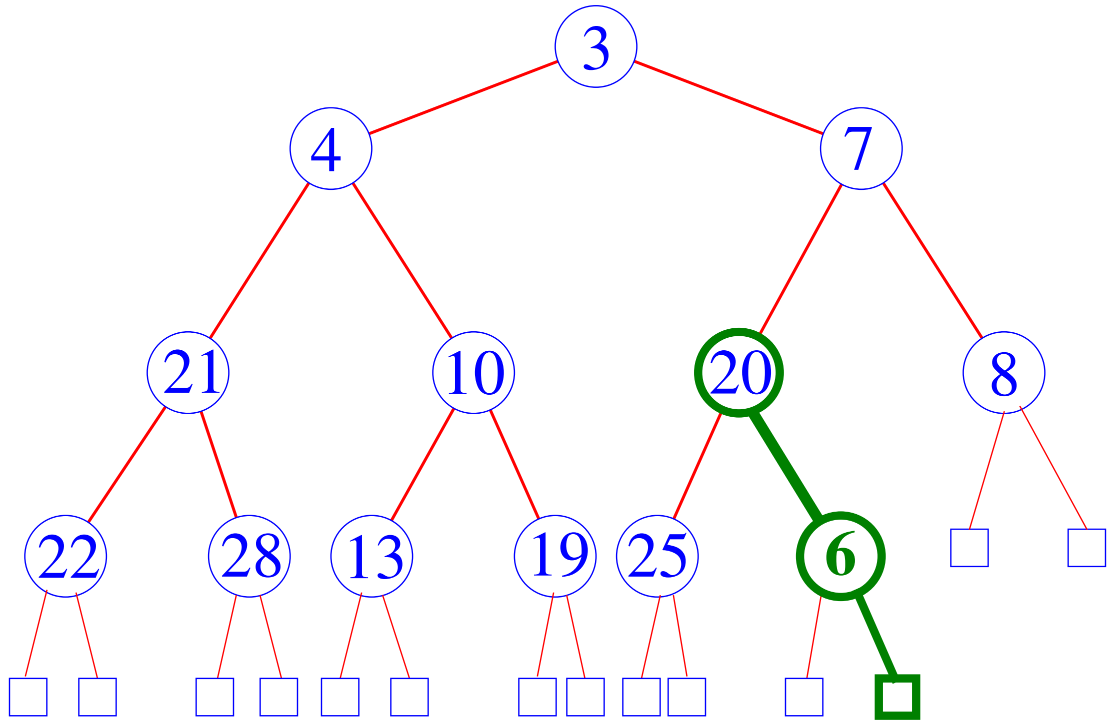
Legg den nye nøkkelen i ***neste ledige posisjon*** i heap'en.

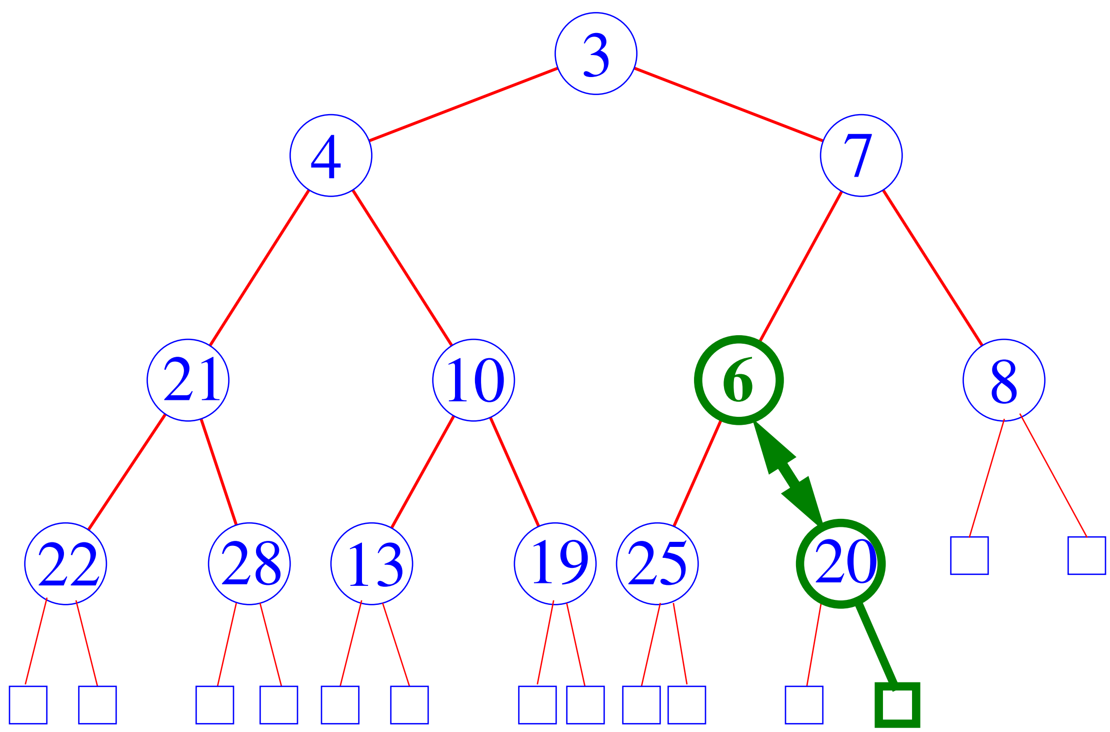


Så må vi gjenopprette Heapordning vha ***Upheap***.

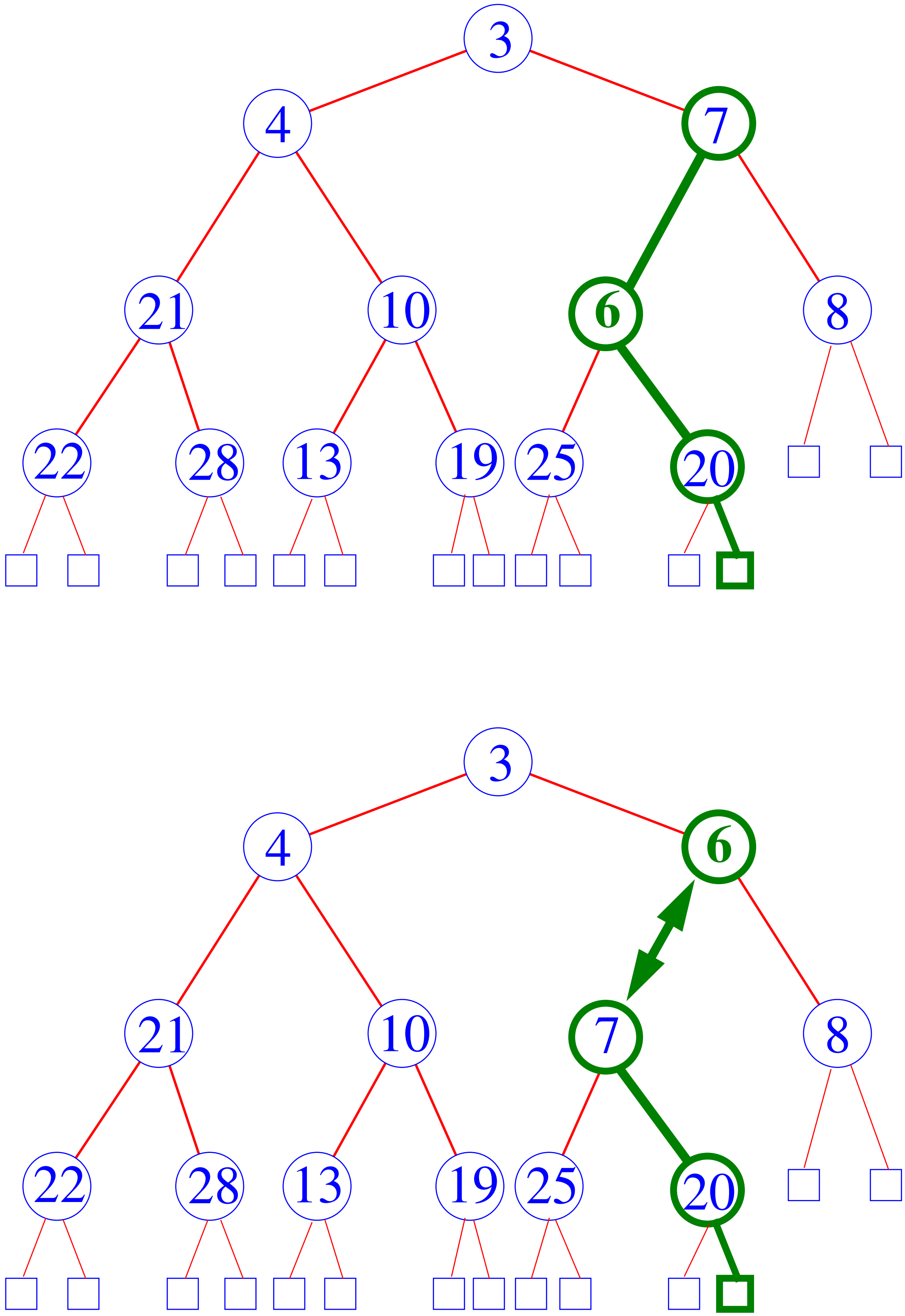
Upheap

- ***Swap alle feil forelder-barn nøkkelpar:***

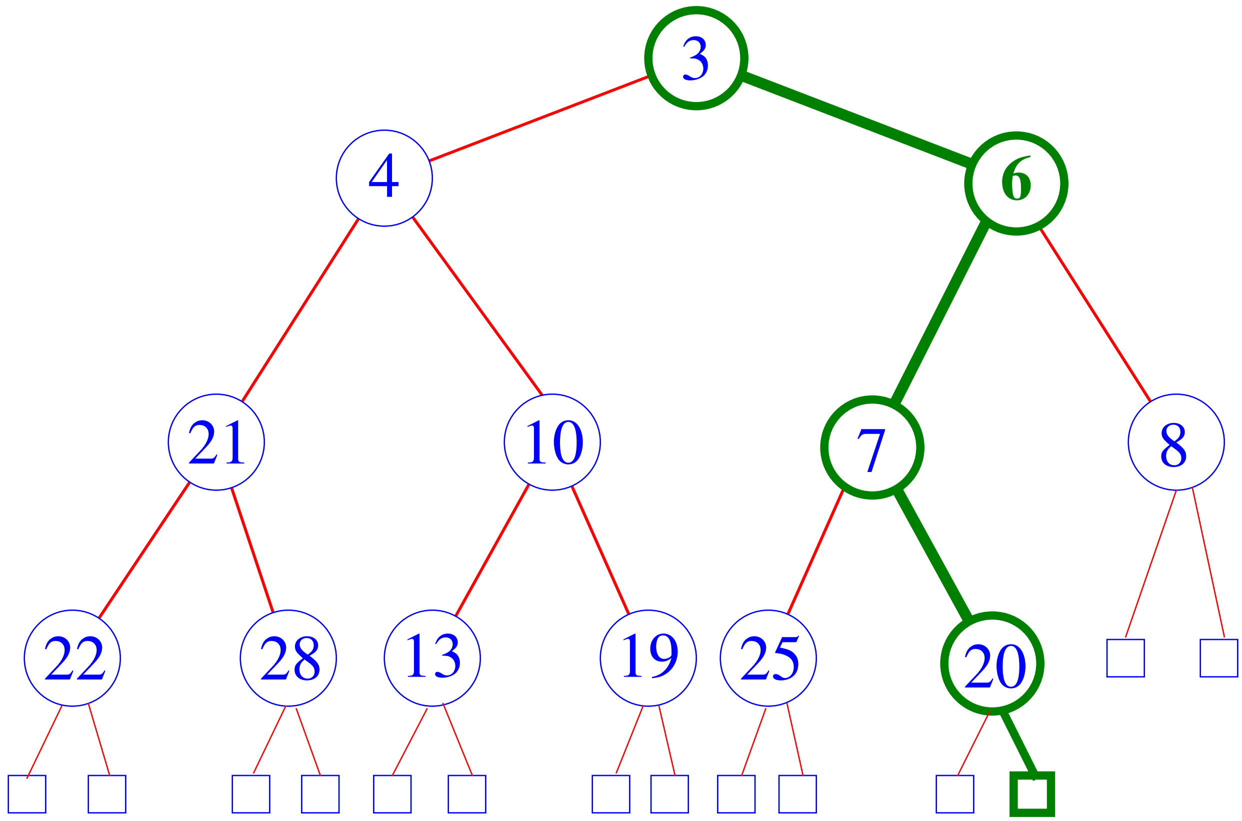




Upheap forts. langs sti til rot

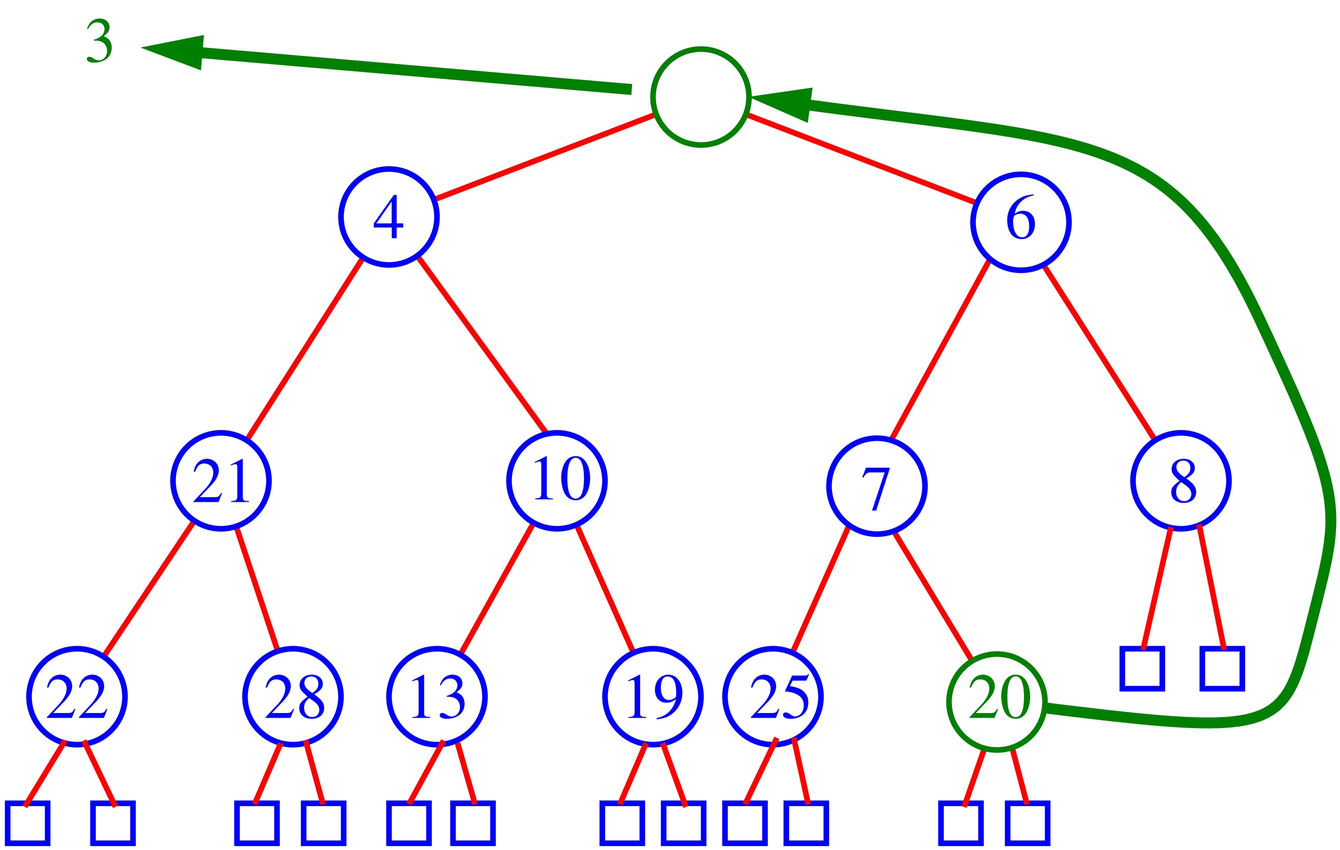


Slutt på Upheap



- ***Upheap*** slutter når den nye nøkkelen er større enn foreldrenøkkel eller når rot er nådd.
- Tidskompleksitet for Insert(key) blir da : O(total #swaps) $\leq$ O(*høyde*), dvs O(log *n*)

Fjerne minste nøkkel fra Heap
RemoveMin()

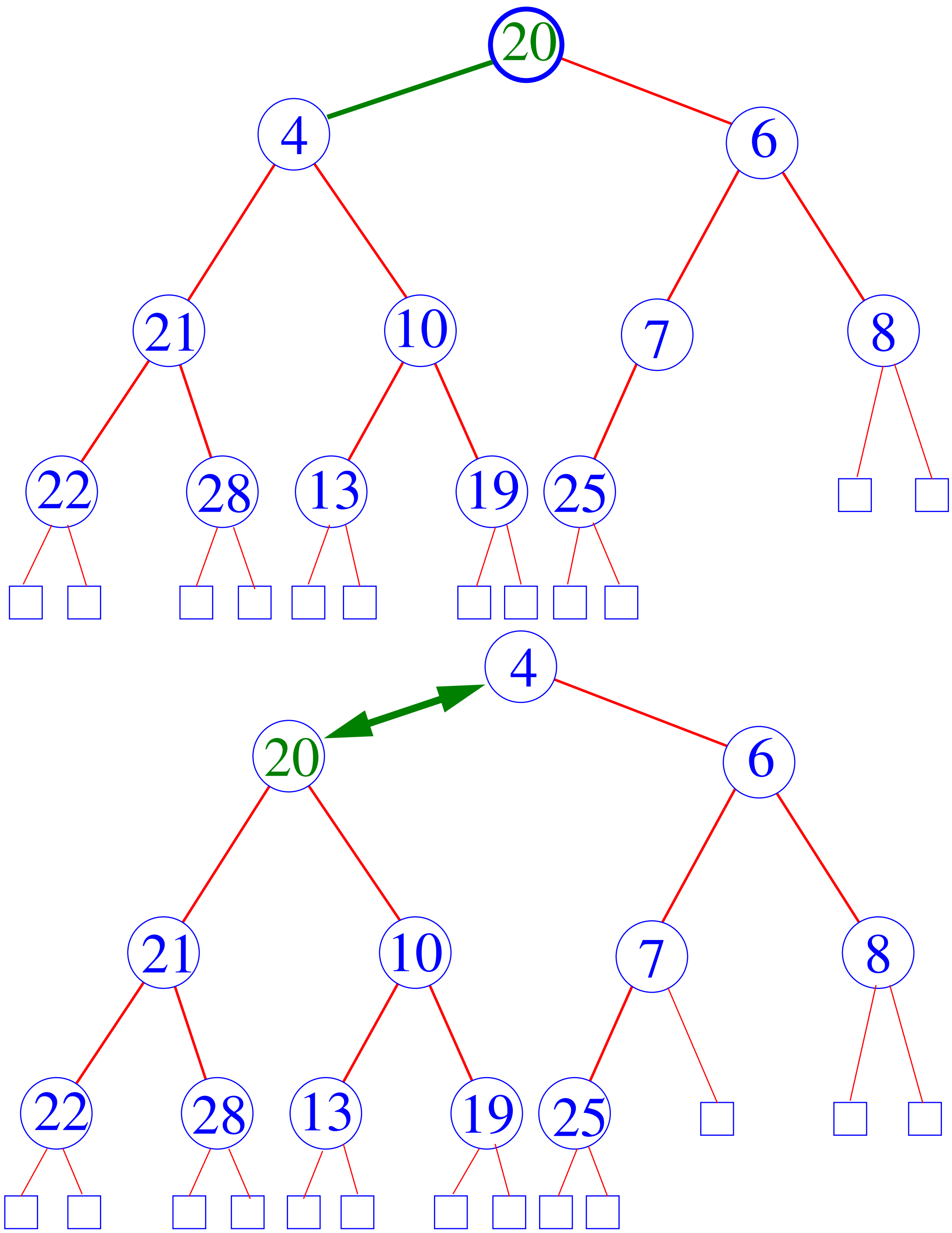


- Minste nøkkel er i rotnode (Heapordning)
- Fjerning av rotnode gir Komplett Binært Tre med 1 node mindre

- Løsning:

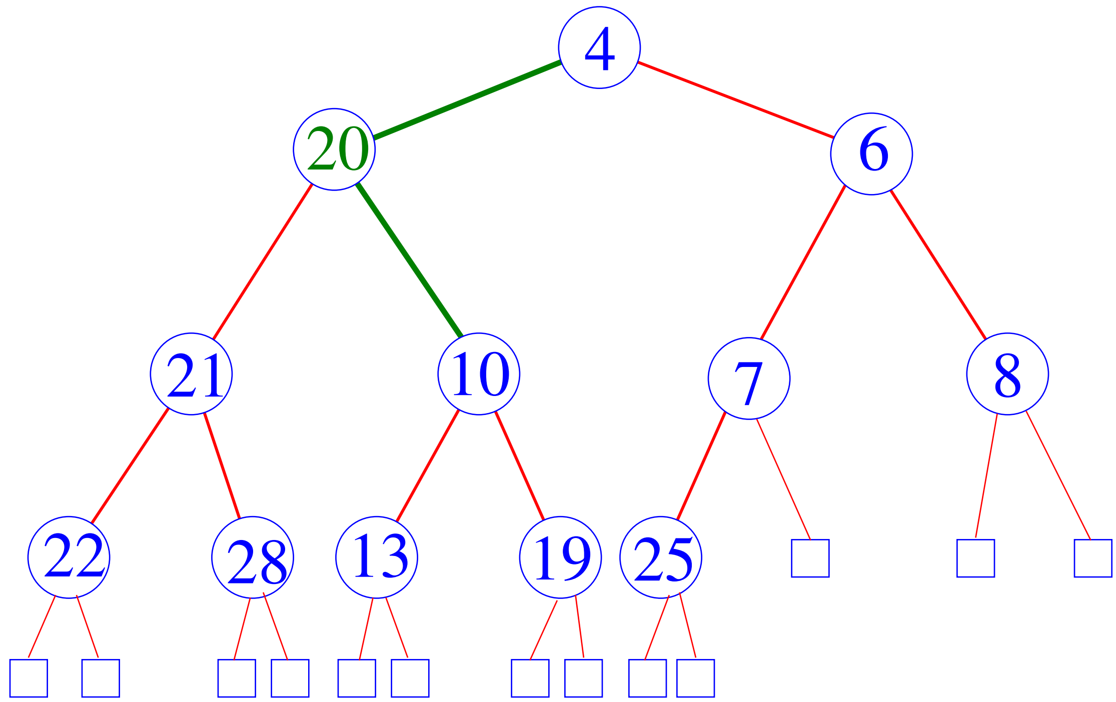
1) Siste node legges i rotnode
2) Gjenopprett Heapordning: ***Downheap***

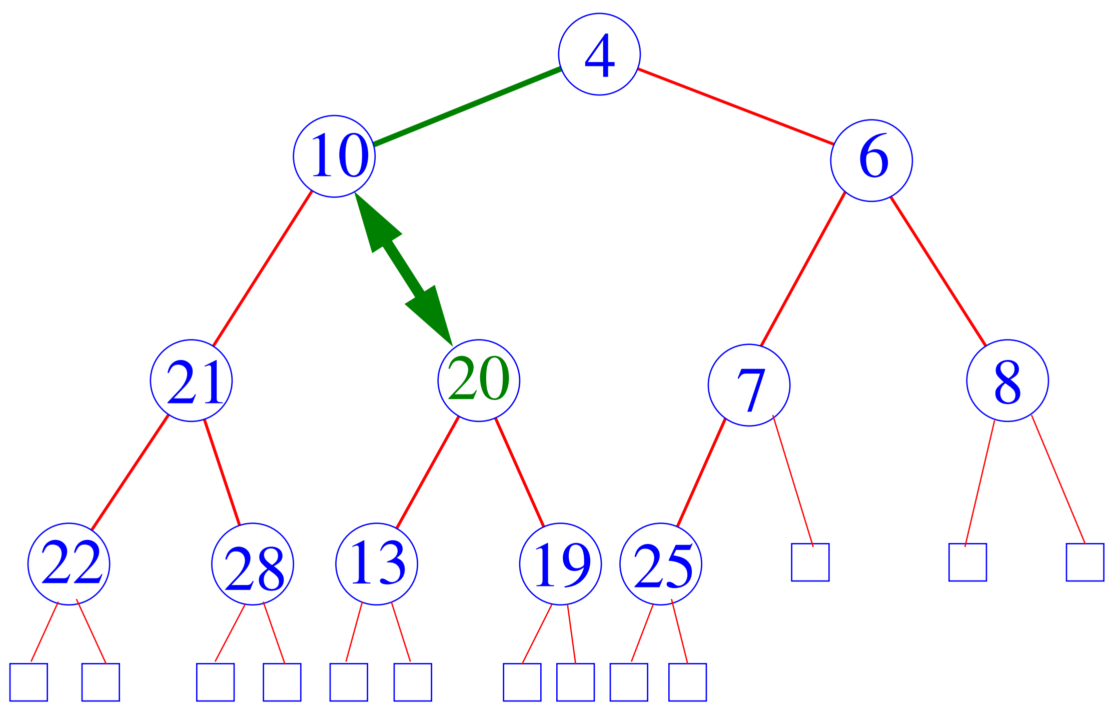
Downheap



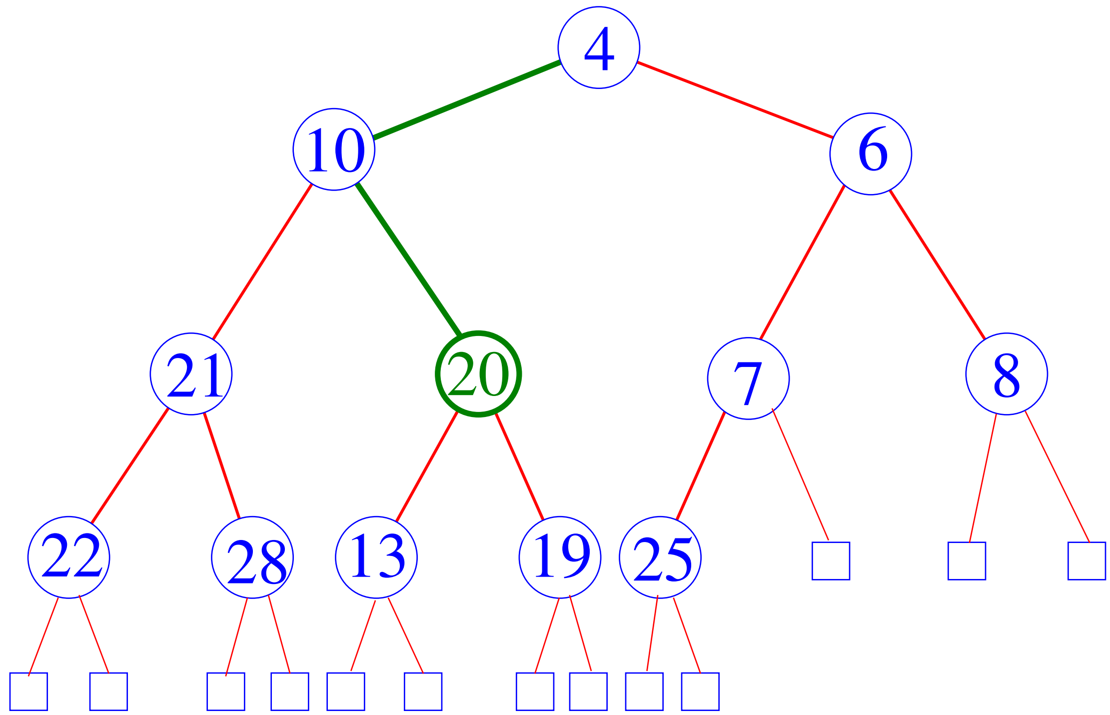
Downheap sammenlikner forelder med **minste** barn. Om mindre, utfør swap og fortsett.

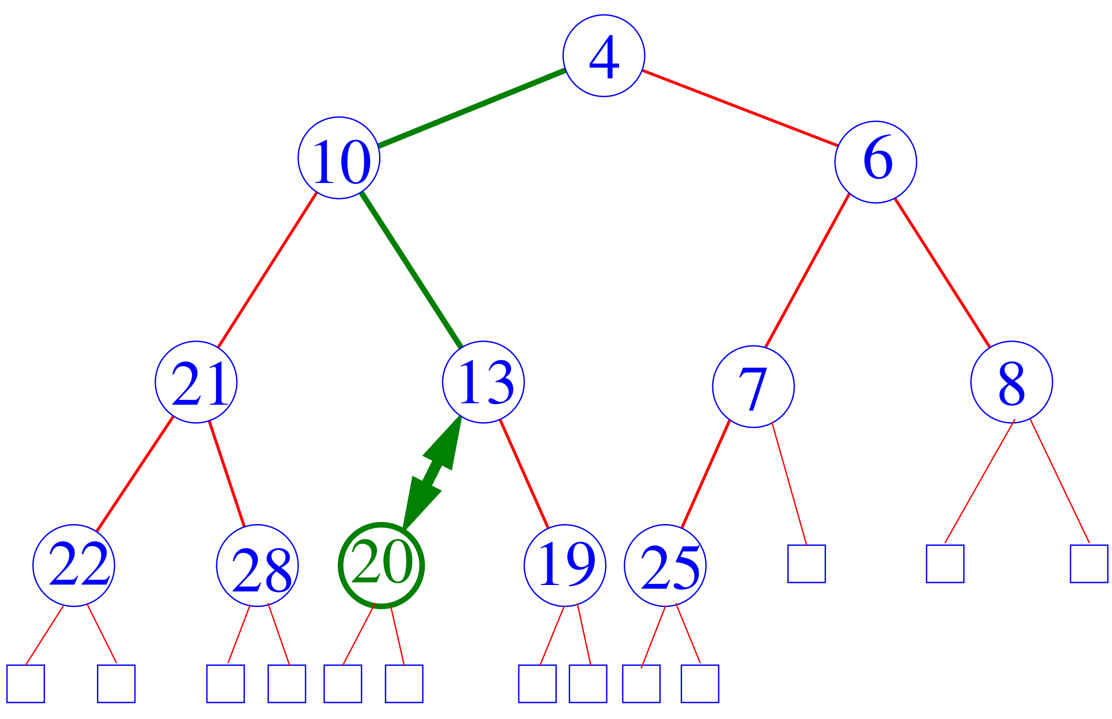
Downheap forts på ‘minste sti’



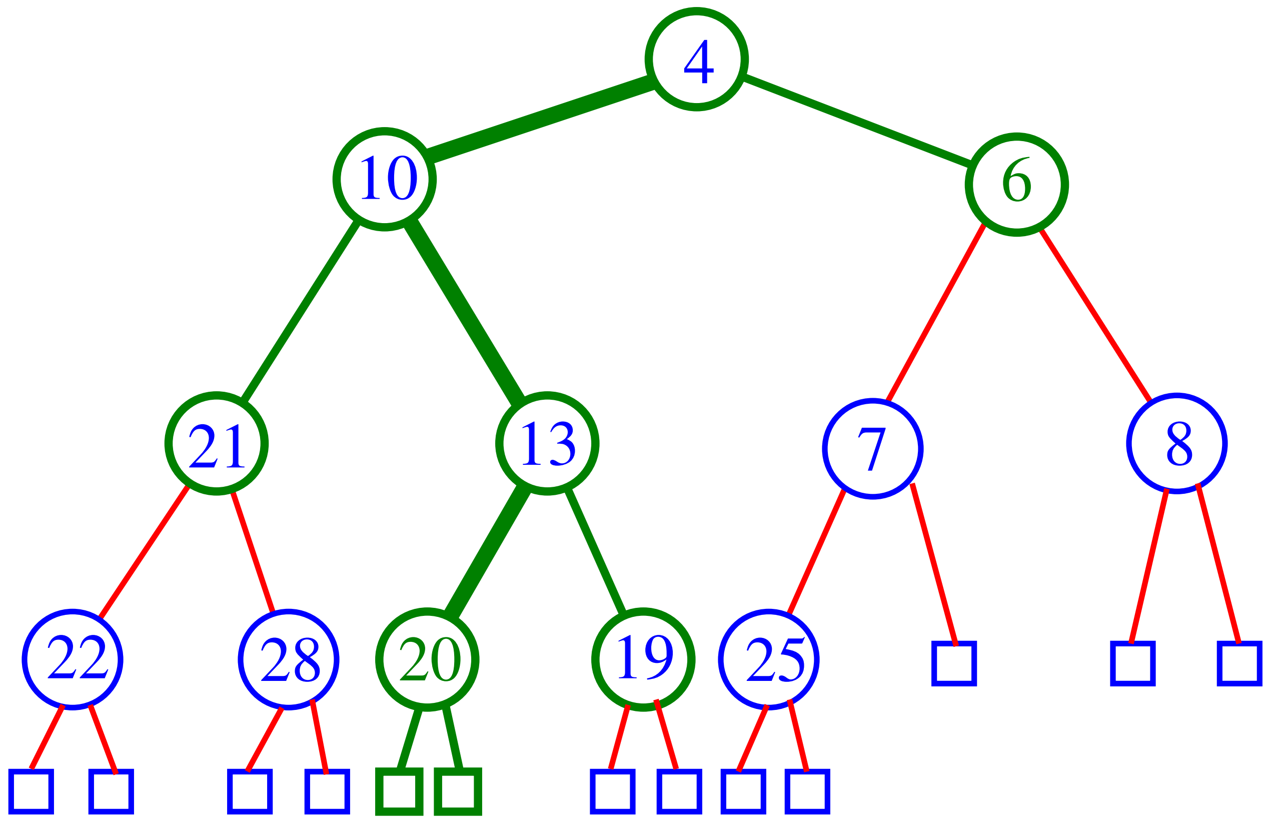


Downheap forts på 'minste sti'





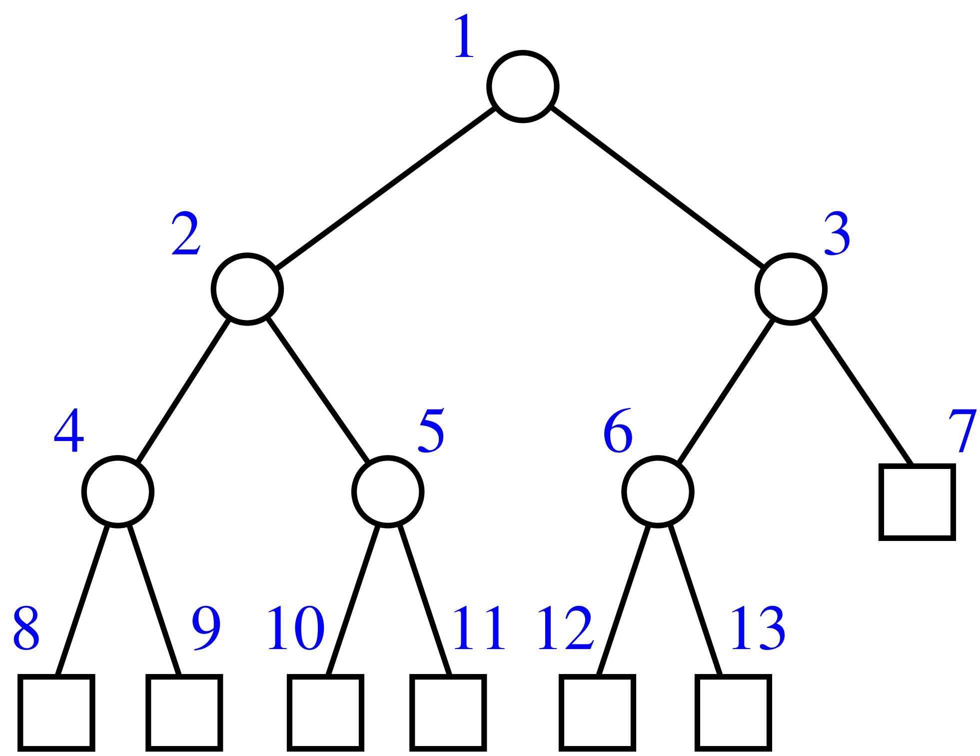
Slutt på Downheap



- ***Downheap*** slutter når nøkkelen er større enn nøkkelen i begge barn, eller når vi har nådd nederste nivå.
- Tidskompleksitet for RemoveMin() blir da: O(total #swaps) $\leq$ O(*høyde*), dvs O(log *n*)

Hvordan finne 'Neste Ledige Node?'

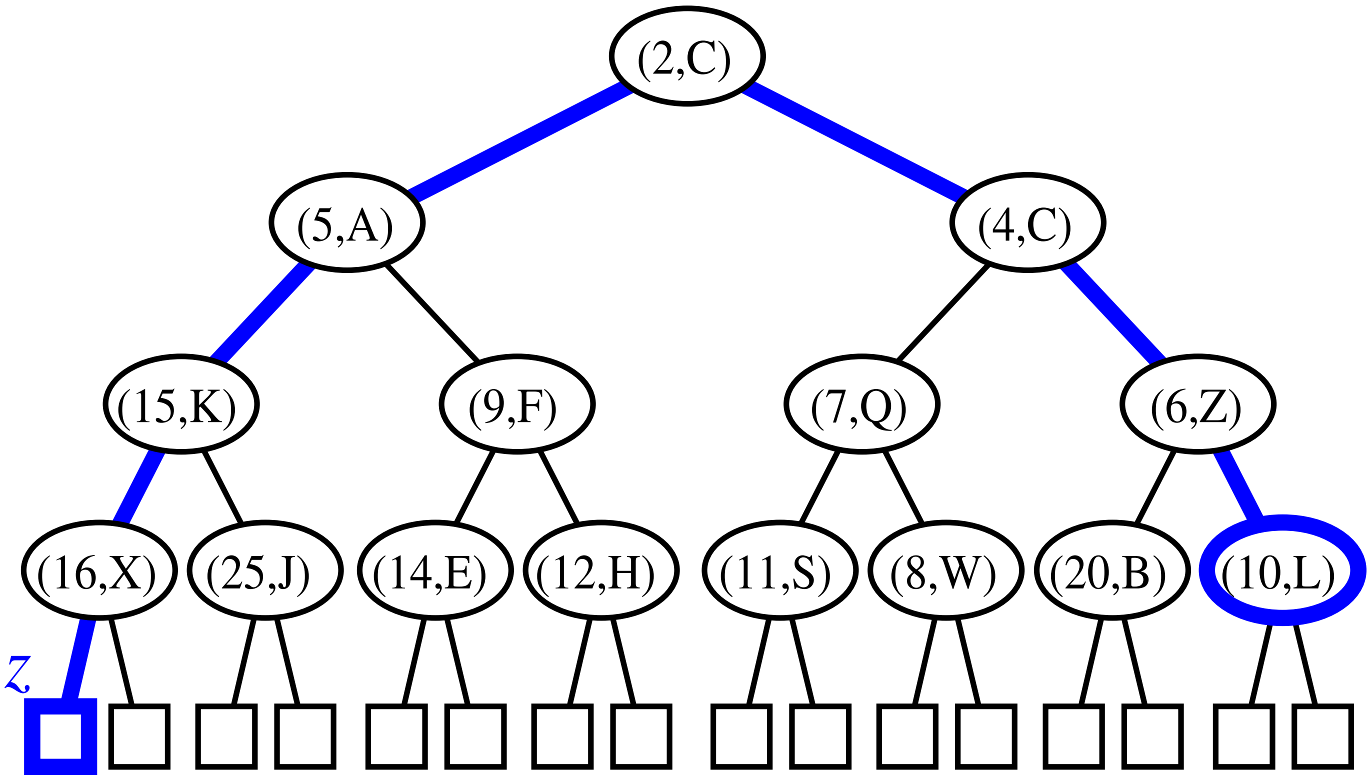
- Heap implementert vha array eller ADT Sekvens
- Node med indeks/rank i har venstre barn med indeks/rank 2j og høyre barn med indeks/rank 2j+1.

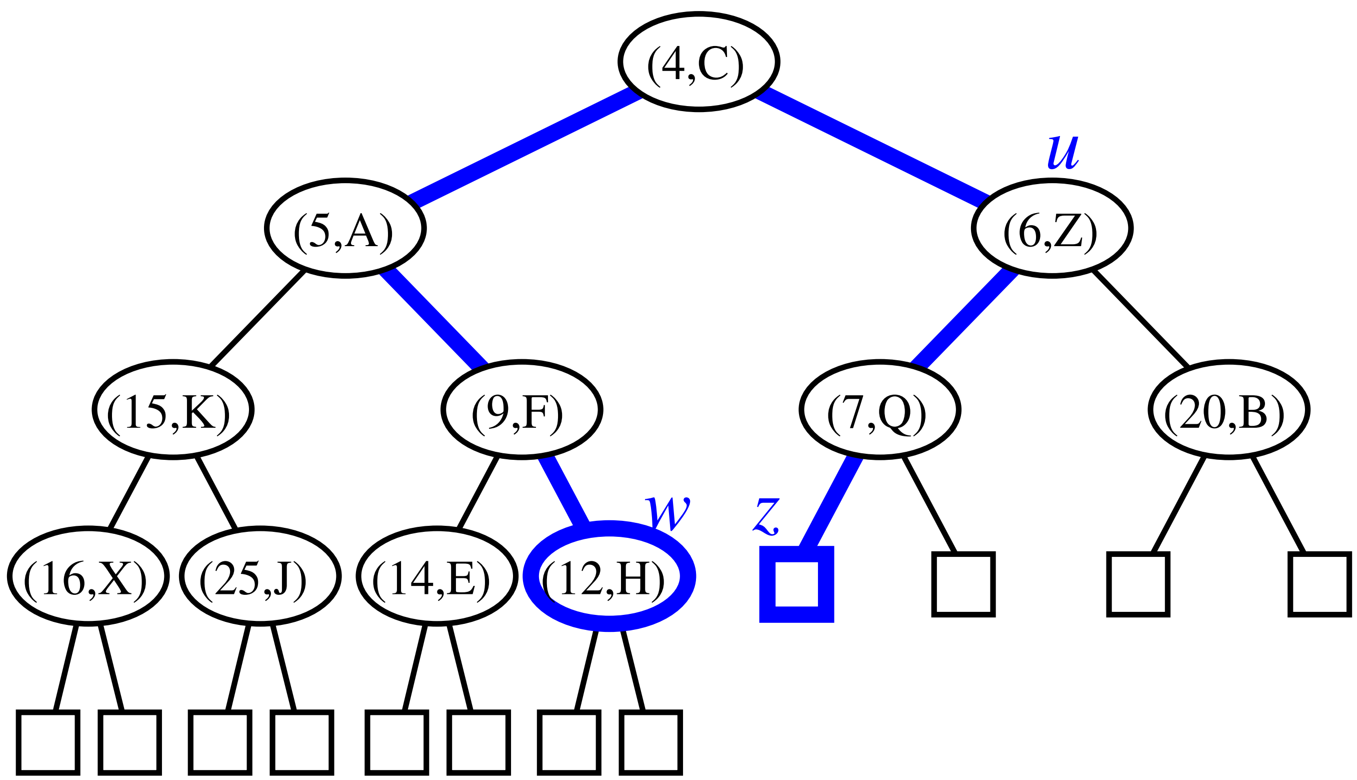


- Neste ledige node er i 'neste ledige indeks/rank'.
- Insetting og fjerning anvender metoder insertLast og removeLast for sekvensen, da Last alltid vil være neste ledige node.

Hvor er 'Neste ledige Node'?

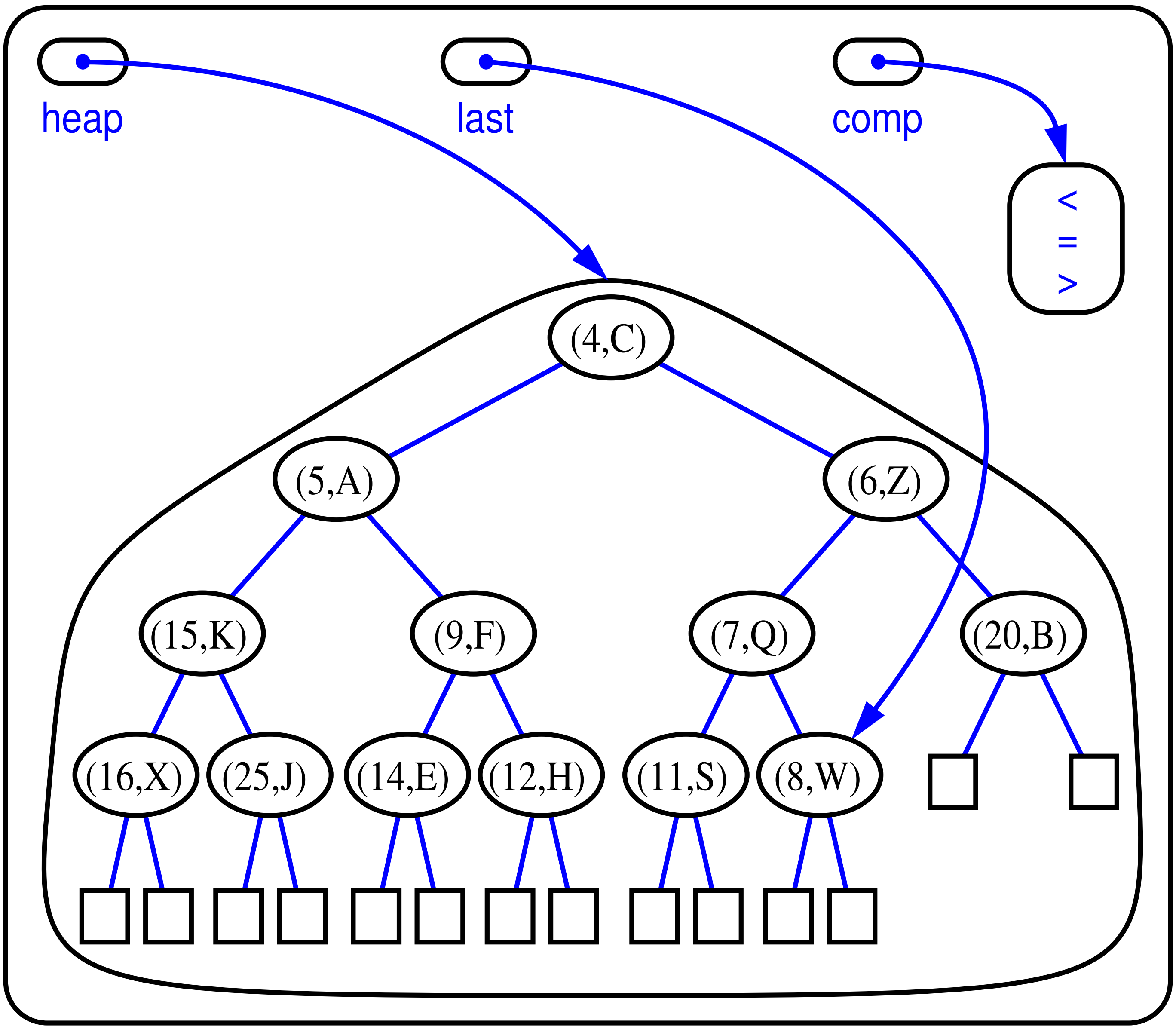
- Heap impl. vha ADT Binært Tre. 2 spesialtilfeller:





PQ implementert vha Heap

```
public class HeapPriorityQueue implements
PriorityQueue {
   BinaryTree T;
   Position last;
   Comparator comparator;
   ...
}
```



PQ vha Heap

- Med Prioritetskø implementert som heap, vil insertItem og removeMin hver ta tid O(log k), hvor k er antall elementer i PQ på det aktuelle tidspunkt.
- Om vi har max n elementer i heap, så er høyden på heap alltid $\boldsymbol{h} \mathbf{<=} \lceil \mathbf{log}(\boldsymbol{n}\mathbf{+1}) \rceil$ og verste-fall tidskompleksitet for disse metodene O(log n).

Heap-Sortering

```
1:for i=1 to n {Heap.insertItem(key,element)}
2:for i=1 to n {Heap.removeMin()}
```

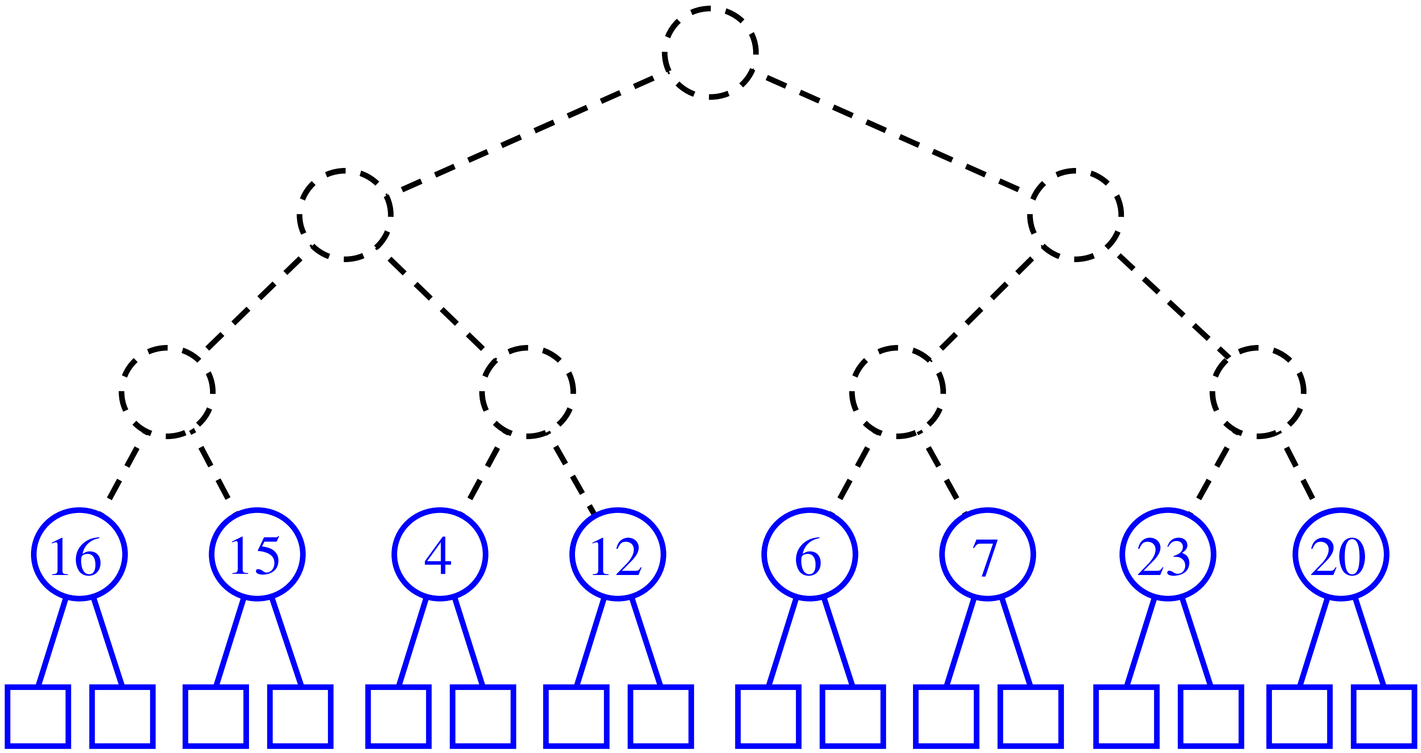
- Både fase 1 og 2 tar O(n log n) tid, og dermed blir ***heap-sortering*** en O(n log n) algoritme.
- O(n log n) kjøretid er mye bedre enn O(n^2) som for selection-sort og insertion-sort.

In-Place Heap-Sort

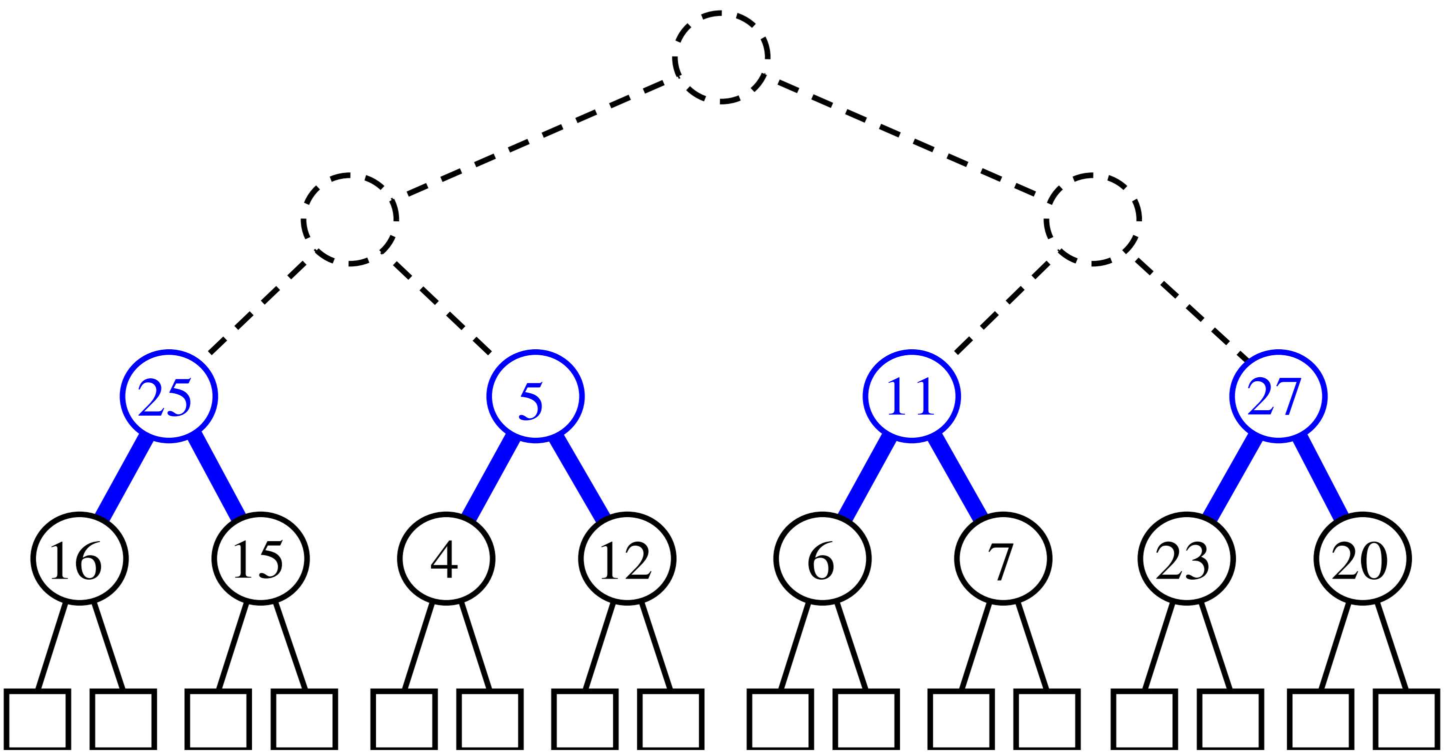
- Gitt sekvens som skal sorteres, så bygg en heap ut av **samme** sekvensen. Ikke bruk ekstern heap.
- Hvordan bygge heap ut av en sekvens?

Bottom-Up Heap Konstruksjon

- bygg $(n + 1)/2$ trivielle ett-element heap'er

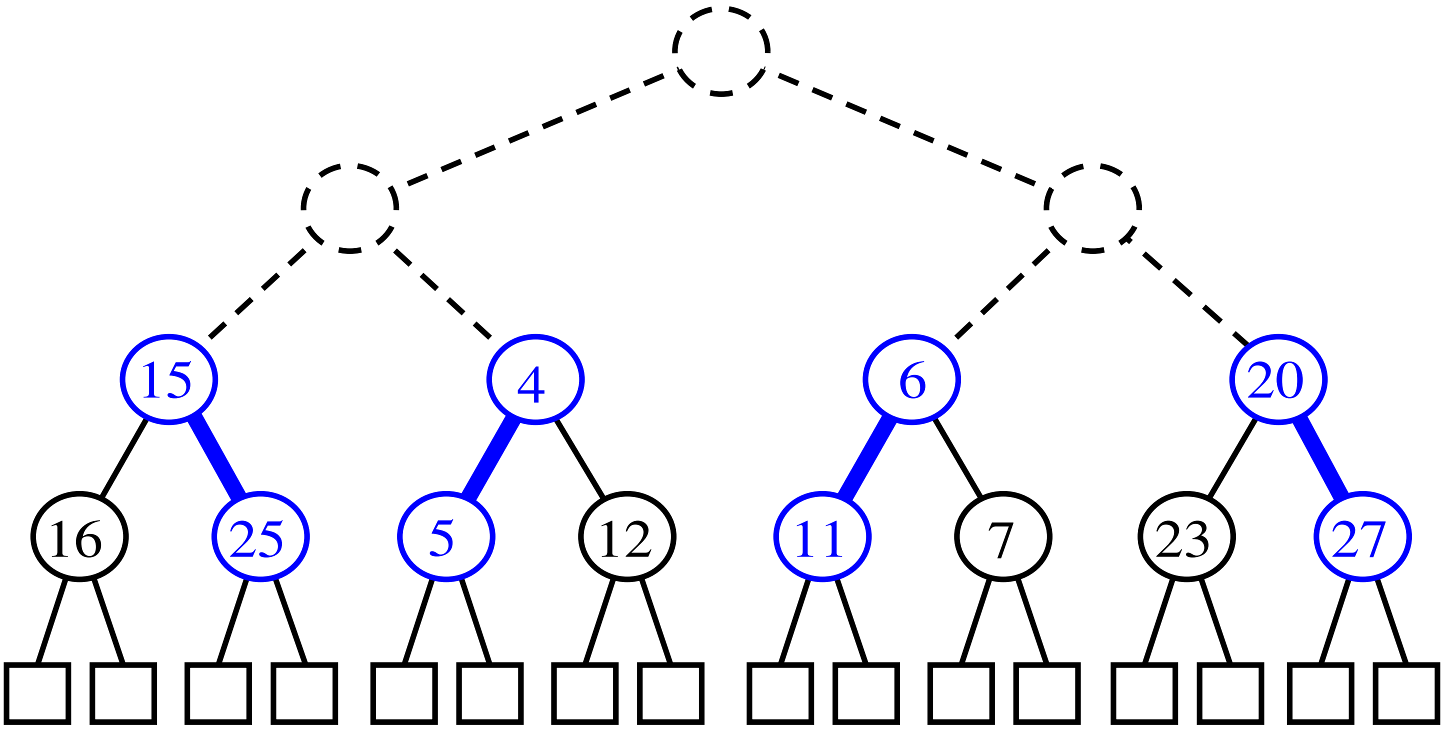


- bygg så 3-element heap'er oppå disse

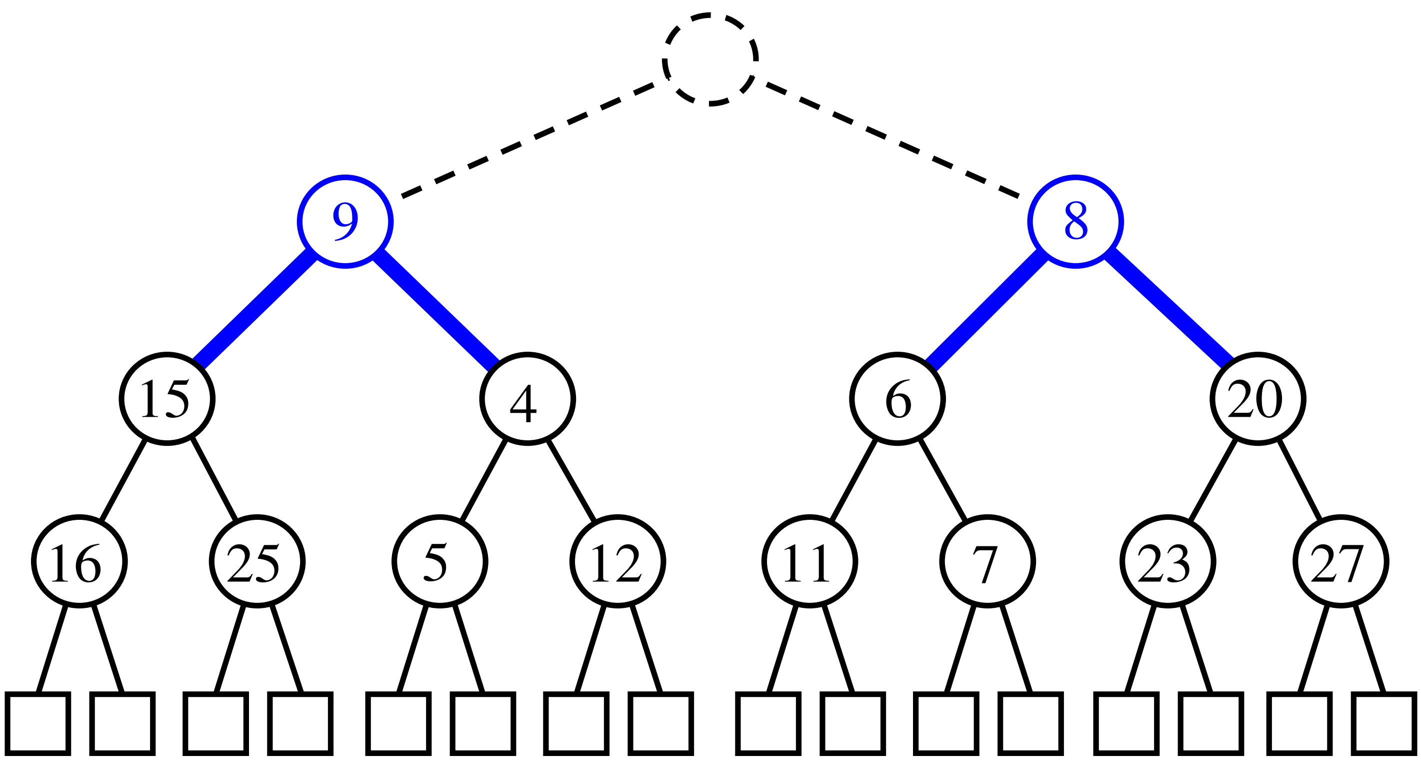


Bottom-Up Heap Konstruksjon

- ***downheap*** for a bevare heap-ordning

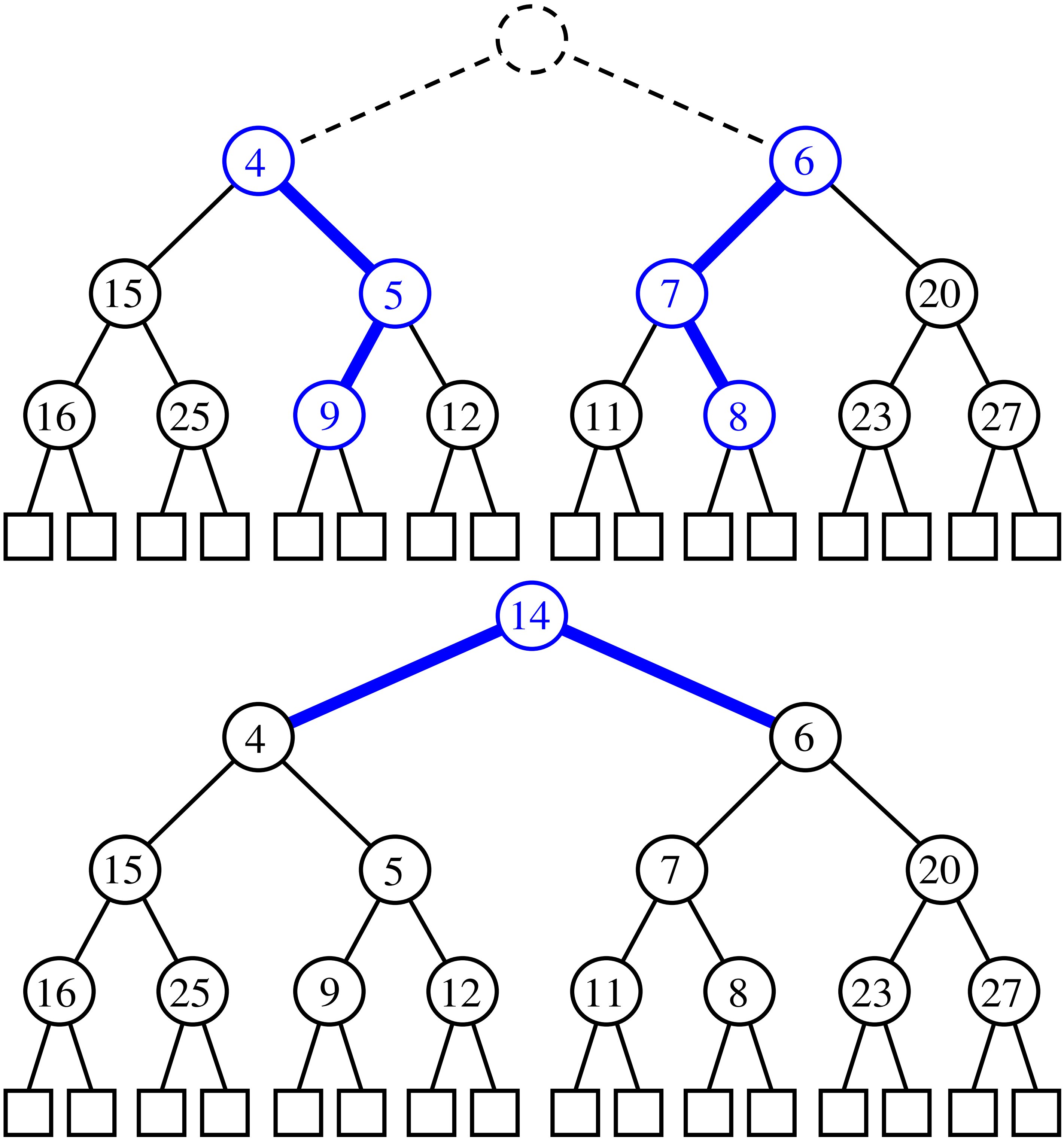


- bygg så 7-element heap'er oppå disse

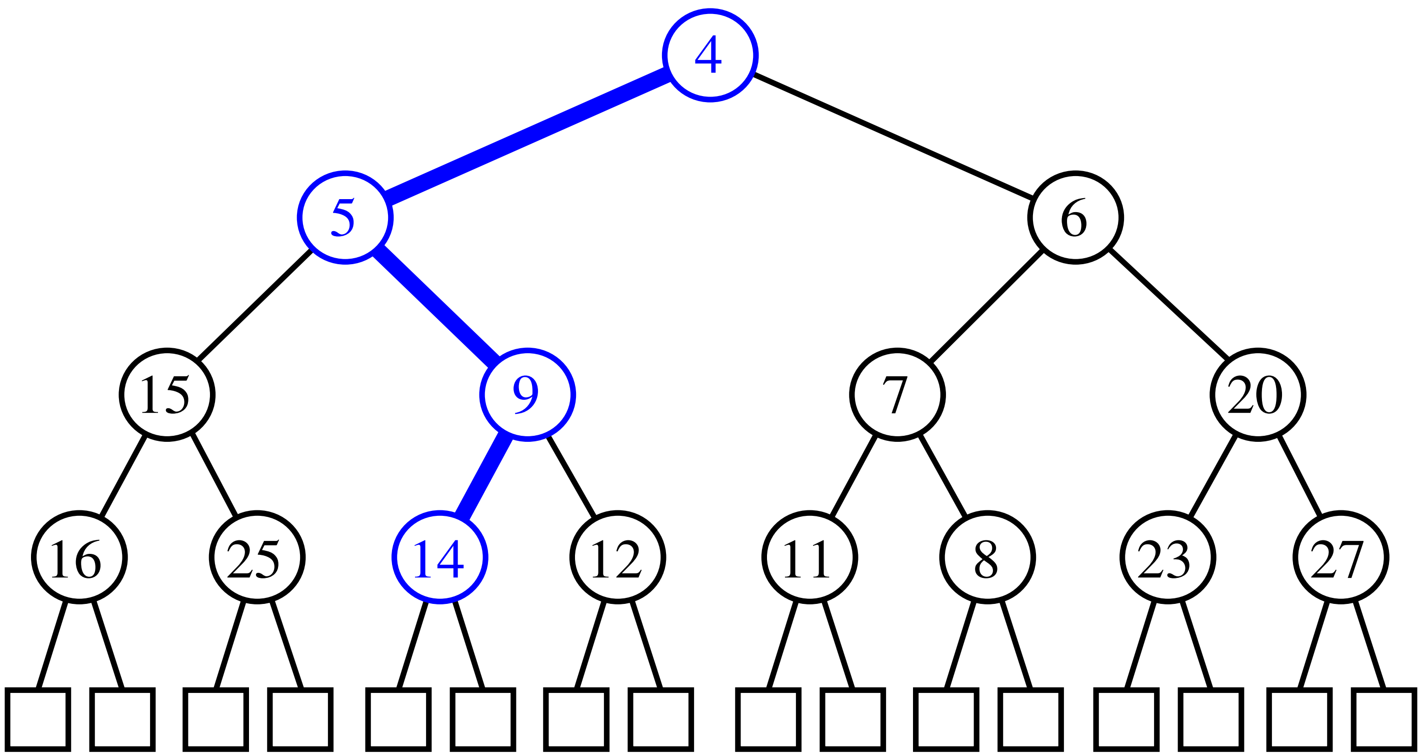


Bottom-Up Heap Konstruksjon

- Utfør downheap på disse 7-element heap'er.



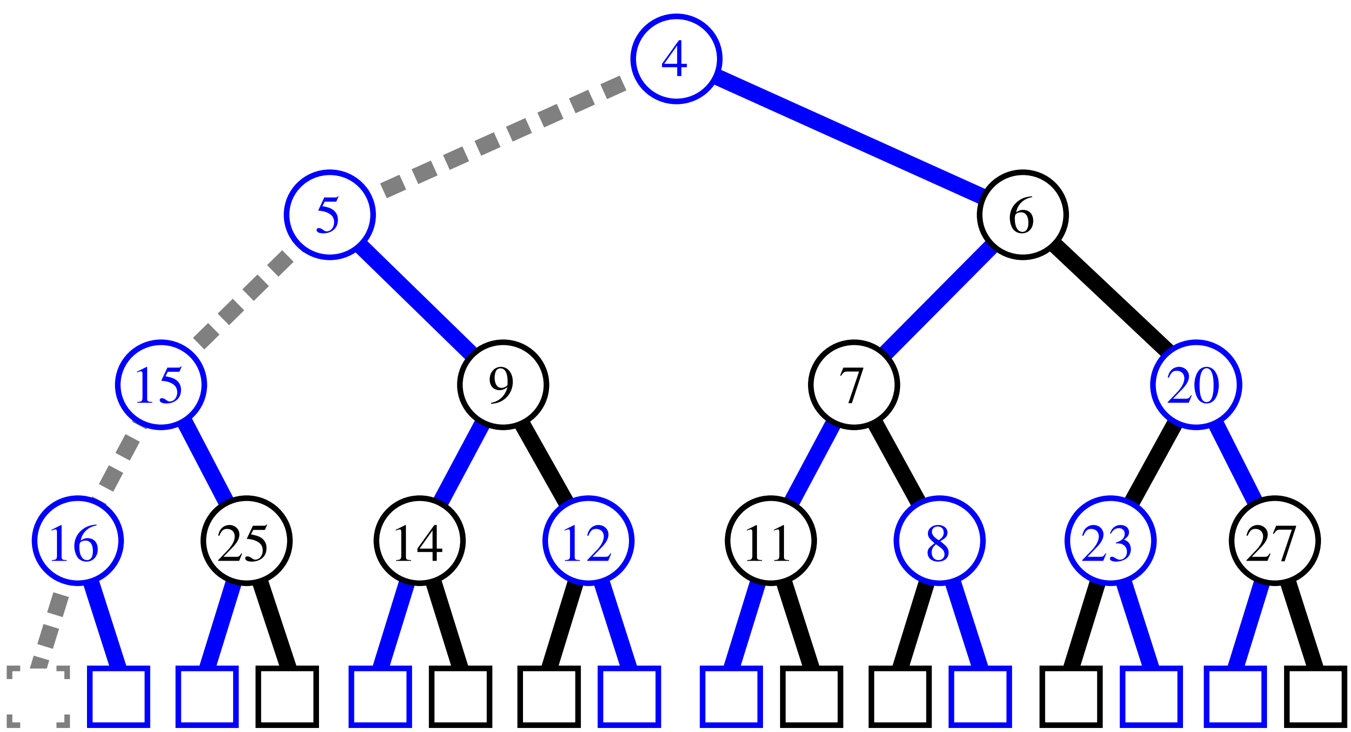
Bottom-Up Heap konstruksjon (forts.)



Presto! En heap.

Analyse av Bottom-Up Heap Konstruksjon

- Påstand: Bottom-up heap konstruksjon med n nøkler tar $\boldsymbol{O}(n)$ tid, selv om vi også setter inn i sekvensen.
  - Sett inn $(n + 1)/2$ noder
  - Sett inn $(n + 1)/4$ noder og utfør downheap
  - Sett inn $(n + 1)/8$ noder og utfør downheap
  - ...
  - visuell analyse:



- n innsettinger, $n/2$ upheaps med totalt $\boldsymbol{O}(n)$ kjøretid
- Vi kan altså **bygge** en heap i O(n) tid.

Locator

- En Locator er knyttet til et objekt og brukes til å finne igjen objektet selv om det har blitt flyttet til en ny posisjon i en samling.
- ADT locator har følgende metoder:
  - element(): return the element of the item associated with the locator.
  - key(): return the key of the item assocated with the locator.
- Vha locator kan vi legge inn følgende metoder i ADT Prioirtetskø:
  - insert(k,e): insert (k,e) into P and return its locator
  - min(): return the locator of an element witih smallest key
  - remove(l): remove the element with locator l
- I applikasjonen for aksjehandel er dette nyttig: returner locator når en ordre settes inn, slik at ved kansellering av samme ordren kan den lett finnes vha locator, selv om ordrens posisjon er blitt endret.

Position og Locator

- Hva er forskjellen på posisjon og locator?
- Deres metoder likner, men bruken er forskjellig
- Posisjon abstraherer en spesifikk implementasjon av aksess til elementer (f.eks. indeks eller noder).
- Posisjoner er definert relativt til hverandre (e.g., previous-next, parent-child)
- Locator'er holder tritt med hvor elementer er lagret.
- Locators assossierer elementer med nøkler.