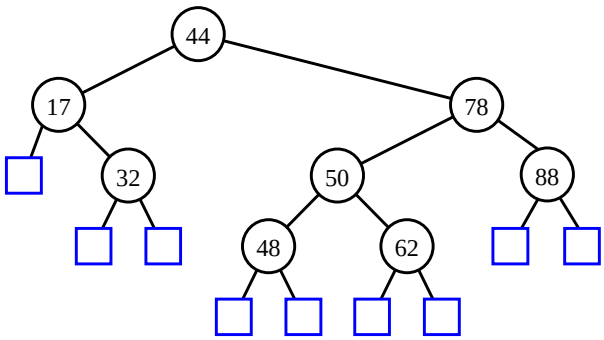
SØKING

- ADT Ordbok (eng: Dictionary)
- Binært Søk
- Binære SøkeTrær

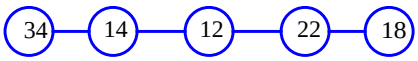


ADT Ordbok

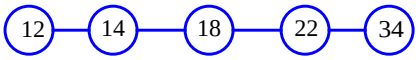
- en ordbok er en abstraksjon av en database
- som i en prioritetskø, så lagrer ordbok nøkkel-element par.
- hovedoperasjon er SØK vha NØKKEL
- container metoder:
  - size()
  - isEmpty()
  - elements()
- query metoder:
  - findElement(*k*)
  - findAllElements(*k*)
- oppdaterings- metoder:
  - insertItem(*k*, *e*)
  - removeElement(*k*)
  - removeAllElements(*k*)
- spesialelement
  - NO_SUCH_KEY, returnert av feilslått søk

Implementering av ordbok vha sekvens

- ***usortert sekvens***



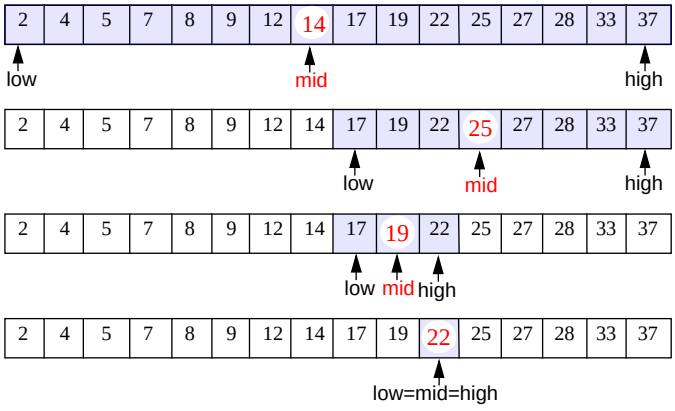
  - søking og fjerning tar O(*n*) tid
  - insetting tar O(1) tid
  - et godt valg for log-filer (mange innsettinger, få søk og fjerning)
- ***array-basert sortert sekvens*** (antar at nøklene har en totalordning)



  - søk tar O(log *n*) tid (vha ***binært søk***)
  - insetting og fjerning tar O(*n*) tid
  - et godt valg for oppslags-tabeller (mange søk, lite innsetting og fjerning)

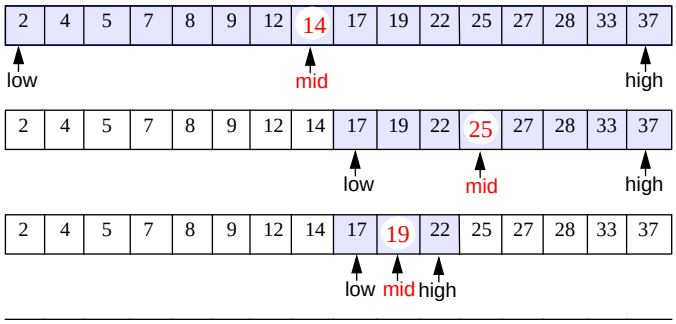
Binært Søk

- Søk etter gitt nøkkel i en sortert sekvens
- Reduser fra søk blant **n** til søk blant **n/2.** Hvordan?
- "høy-lav"
- findElement(22)



Pseudokode for Binært Søk

```
Algorithm BinarySearch(S, k, low, high)
if low > high then
  return NO_SUCH_KEY
else
  mid ← (low+high) / 2
  if k = key(mid) then
    return key(mid)
  else if k < key(mid) then
    return BinarySearch(S, k, low, mid–1)
  else
    return BinarySearch(S, k, mid+1, high)
```



Kjøretid for binært søk

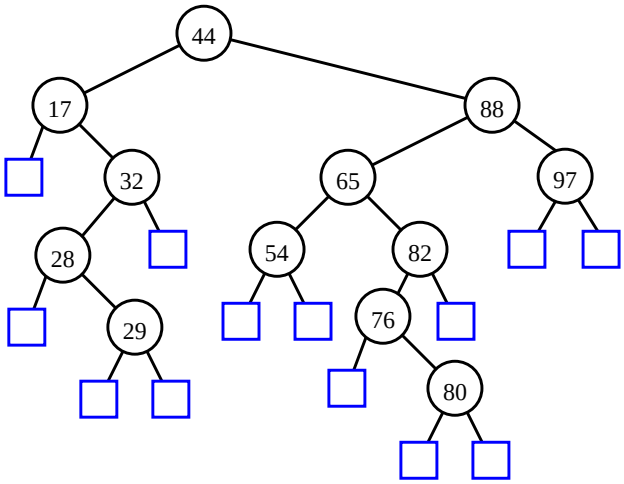
- Kandidatmengden ***halveres for hvert steg***

steg	|søkemengde|
0	n
1	$n/2$
2	$n/4$
...	...
i	$n/2^i$
$\log_2 n$	1

- I en array-basert implementasjon av sekvens tar rank-aksess tid O(1), dermed blir binærsøking en **O(log *n*)** operasjon

Binære SøkeTrær

- Et binært søketre er et binært tre hvor
  - hver intern-node lagrer et (nøkkel, element)-par fra ordboken.
  - for node som lagrer (k, e) vil alle nøkler i venstre deltre ha nøkkel < k og i høyre ha nøkkel >= k.



Søk

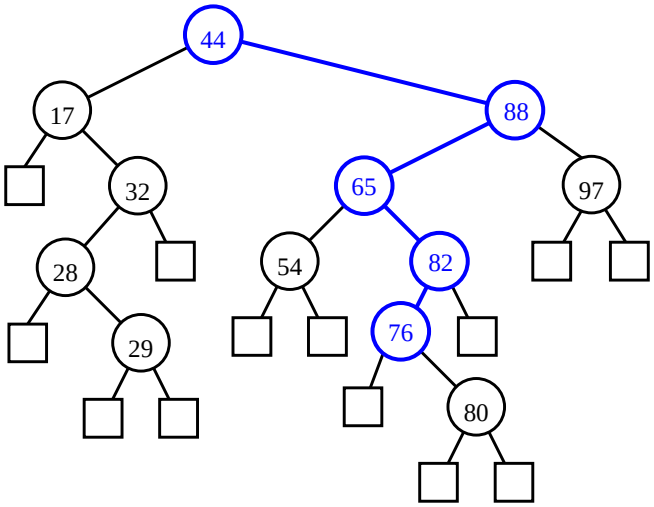
- Et binært søketre *T* er et ***decision tree***, hvor spørsmålet som stilles for en intern-node *v* er om nøkkelen *k* er mindre enn, større enn, eller lik *v*.
- Pseudokode:

Algorithm TreeSearch(*k*, *v*):
Input: Søkenøkkel *k* og en node *v* i et binært søketre *T*.
Output: Node w i deltreet *T*(*v*) med rot i *v*,
s.a. enten er w en intern node som lagrer nøkkel *k* eller så er w en ekstern node som angir plassen der k ville vært lagret om den befant seg i treet.

```
if v is an external node then
  return v
if k = key(v) then
  return v
else if k < key(v) then
  return TreeSearch(k, T.leftChild(v))
else
  { k > key(v) }
  return TreeSearch(k, T.rightChild(v))
```

Søk-eksempel I

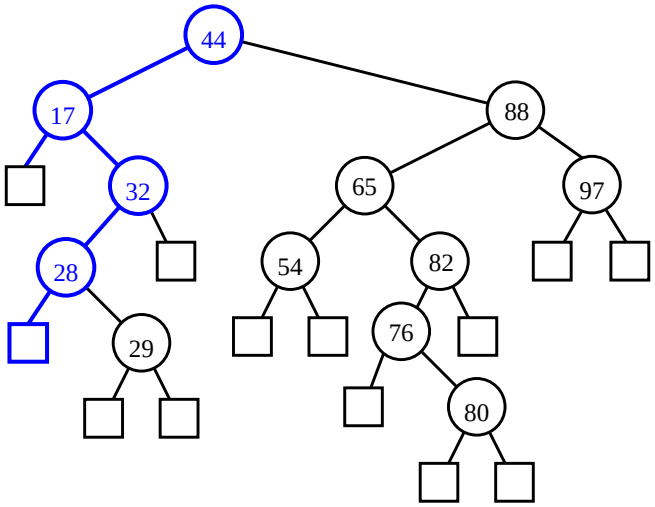
- findElement(76)



- Et søk med treff starter i rotnode og ender i en internnode.
- Hva med findAllelements(*k*)? Hva gir Innorden traversering?

Søke-eksempel II

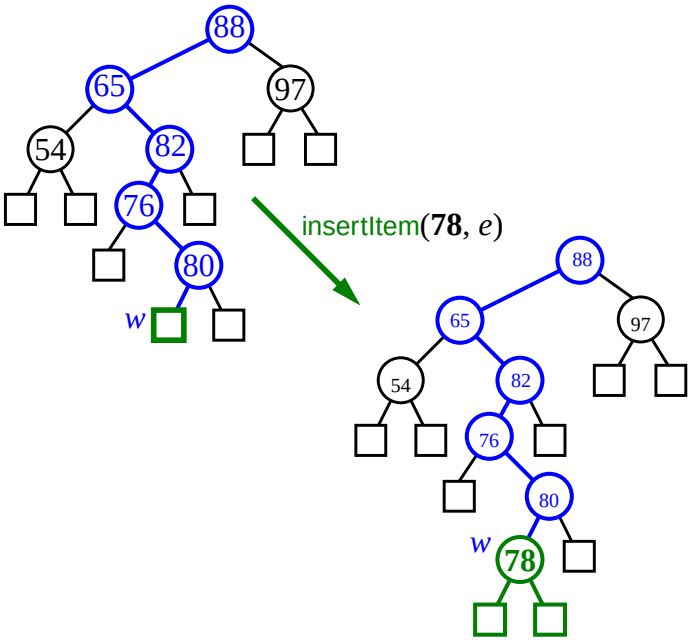
- findElement(25)



- Et søk etter en nøkkel som ikke finnes i treet starter i rotnode og ender i en løv-node.

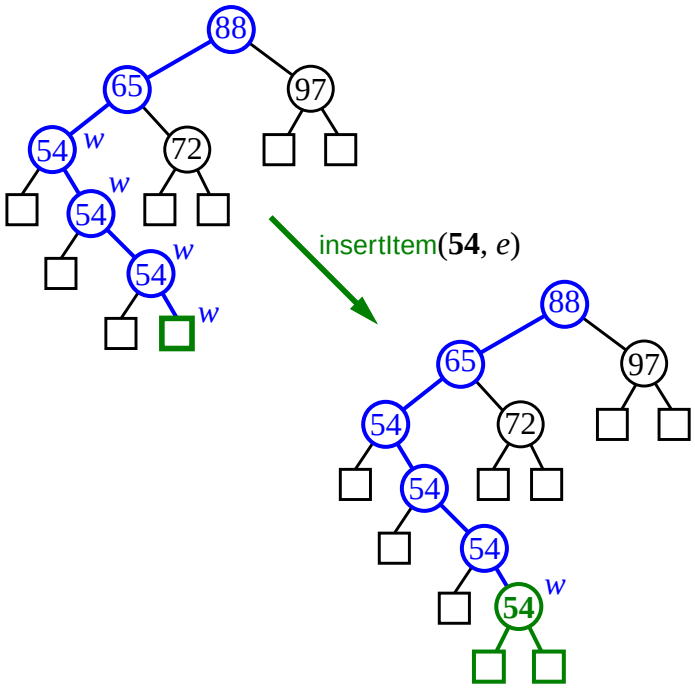
Insetting

- For insertItem(*k*, *e*), lar vi *w* være noden returnert av TreeSearch(*k*, *T*.root())
- Om *w* er ekstern, vet vi at *k* ikke er lagret i *T*. Vi kaller expandExternal(*w*) på *T* og lagrer (*k*, *e*) i *w*



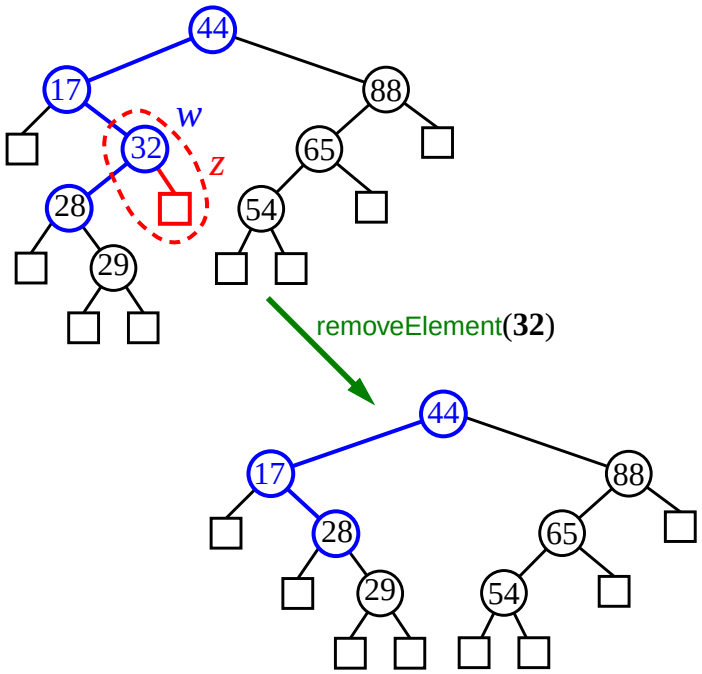
Insetting II

- Om *w* er intern, betyr dette at et annet element med nøkkel *k* er lagret i *w*. Vi kaller innsetting rekursivt på *T*.rightChild(*w*). insertItem(**54**, *e*):



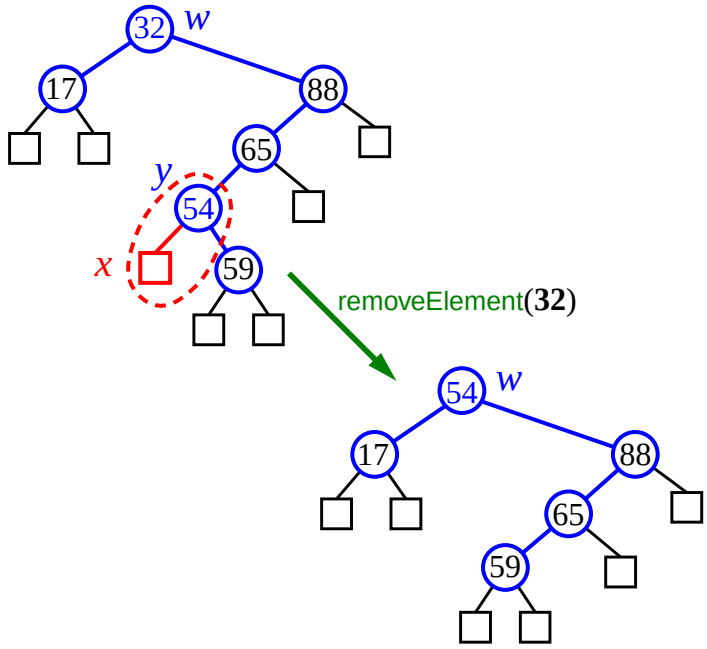
Fjerning I

- Finn node *w* hvor nøkkelen er lagret vha TreeSearch
- Om *w* ***har et eksternt barn*** *z*, fjerner vi *w* og *z* med removeAboveExternal(*z*)



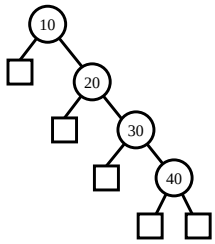
Fjerning II

- Om *w* ***ikke har eksterne barn***:
  - finn intern-node *y* som følger etter *w* i innorden
  - flytt (nøkkel,element) par fra *y* til *w*
  - utfør removeAboveExternal(*x*), hvor *x* er venstre barn av *y* (denne må være eksternl)



Kjøretid

- Et søk, en innsetting eller en fjerning traverserer alle en ***rot-mot-løv sti***, og besøker kanskje også ***søsken*** til noder på denne stien
- Det brukes O(1) tid for hver traverserte node
- Kjøretiden for hver operasjon blir O(*h*), hvor *h* er høyden på treet
- Høyden på det binære treet er i verste fall lik antall lagrede nøkler, dvs *n*, og dette skjer når treet blir til en sti.



- For å oppnå en god kjøretid må treet holdes ***balansert***, dvs vi vil ha O(log *n*) høyde
- Mer sofistikerte metoder for Innsetting og Fjerning vil garantere balanserte trær. Mer om dette i kurset I234.