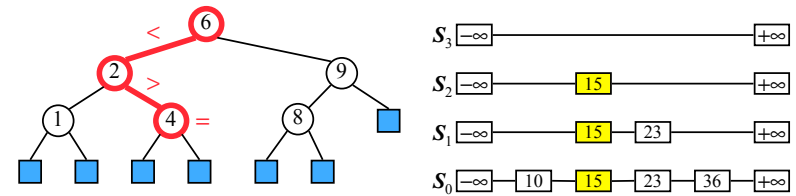


# Søgning



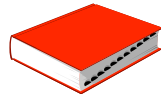
1

# Binære søgetræer og skiplister



2

# Ordbog ADT (Dictionary)



En **ordbog** lagrer en samling af emner, der kan søges i

Hvert emne er et par (nøgle, element)

De centrale operationer for en ordbog er søgning, indsættelse og fjernelse af emner

Anvendelser:

Adressebog

Kontokort-autorisering

Afbildning fra værtsnavne (f.eks.

akira.ruc.dk) til internetadresser

(f.eks. 130.225.220.8)

Metoder:

**findElement(k):**

Hvis ordbogen indeholder et emne med nøgle k, så returneres dets element. Ellers returneres det specielle element NO\_SUCH\_KEY

**insertItem(k, o):**

Indsætter emnet (k, o) i ordbogen

**removeElement(k):**

Hvis ordbogen har et emne med nøglen k, så fjernes det, og dets element returneres. Ellers returneres det specielle element NO\_SUCH\_KEY

**size(), isEmpty()**

**keys(), elements()**

3

# Ordrede ordbøger



Nøglerne antages at opfylde en total ordningsrelation

Nye operationer:

**closestKeyBefore(k)**

**closestElemBefore(k)**

**closestKeyAfter(k)**

**closestElemAfter(k)**

hvor k er en nøgle

4

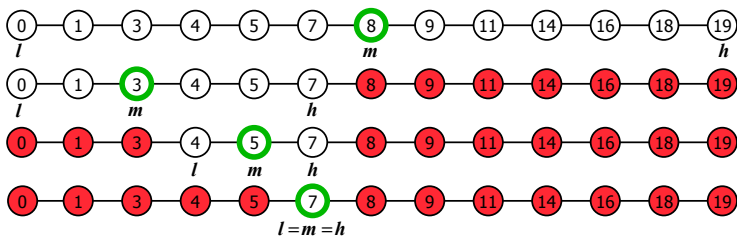
## Binær søgning



Binær søgning udfører operationen **findElement**( $k$ ) på en ordbog implementeret ved en array-baseret sekvens, der er **sorteret** i forhold til nøglerne

I hvert skridt halveres antallet af kandidat-emner  
Algoritmen terminerer efter  $O(\log n)$  skridt

Eksempel: **findElement**(7)



5

## Opslagstabel



En **opslagstabel** er en ordbog, der er implementeret ved hjælp af en sorteret sekvens. Emnerne lagres i en array-baseret sekvens, der holdes sorteret efter nøglerne. Der benyttes et comparator-objekt til sammenligning af nøgler.

Effektivitet:

**findElement** tager  $O(\log n)$  tid ved brug af binær søgning

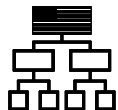
**insertItem** tager  $O(n)$  tid, da vi værste tilfælde skal parallelforskyde  $n-1$  emner for at gøre plads til det nye emne

**removeElement** tager  $O(n)$  tid, da vi i værste tilfælde skal parallelforskyde  $n-1$  emner for at placere tabellens emner kompakt i tabellen

En opslagstabel er kun effektiv for små ordbøger, eller for hvilke søgning er den mest almindelige operation, mens indsættelse og sletning er sjældne operationer (f.eks. autorisering af kontokort)

6

## Binært søgetræ



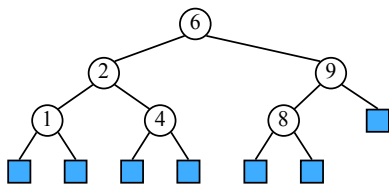
Et **binært søgetræ** er et binært træ, der indeholder nøgler (eller nøgle-element-par) og opfylder følgende betingelse:

En inorder-traversering af et binært søgetræ besøger knuderne i stigende orden

Lad  $u$ ,  $v$  og  $w$  være tre knuder, hvor  $u$  er i  $v$ 's venstre undertræ, og  $w$  er i  $v$ 's højre undertræ. Vi har, at

$$\text{key}(u) \leq \text{key}(v) \leq \text{key}(w)$$

De eksterne knuder indeholder ikke emner

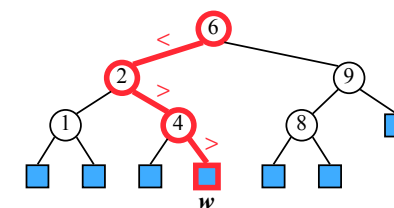


7

## Indsættelse

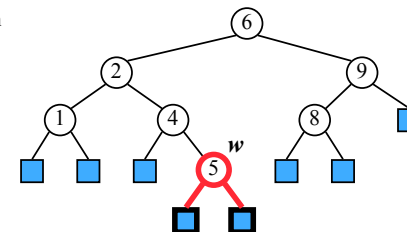
For at udføre operationen **insertItem**( $k, o$ ) søges efter nøglen  $k$

Antag at  $k$  ikke allerede findes i træet, og lad  $w$  være det blad, der nås ved søgningen



Så indsættes  $k$  i knuden  $w$ , og knuden omdannes til en intern knude

Eksempel: indsættelse af 5



8

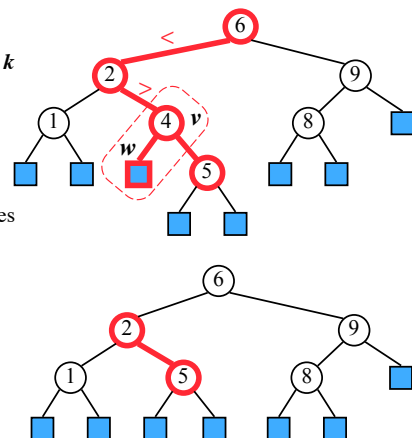
## Fjernelse

For at udføre operationen **removeElement(*k*)** søges efter nøglen *k*

Antag at *k* er i træet, og lad *v* være den knude, der indeholder *k*

Hvis *v* har et blad *w* som barn, så fjernes *v* og *w* fra træet

Eksempel: fjernelse af 4



9

## Fjernelse (fortsat)

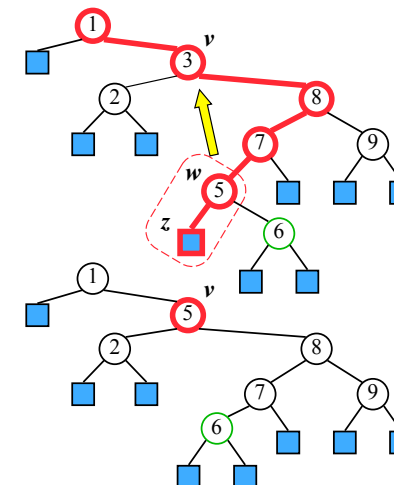
Betragt det tilfælde, hvor nøglen *k*, der skal fjernes findes i en knude *v*, hvis børn begge er interne knuder

Find den interne knude *w*, som besøges efter *v* in ved en in-order-traversering

Kopier *key(w)* til knuden *v*

Fjern *w* og dennes venstre barn, *z* (som må være et blad)

Eksempel: fjernelse af 3



10

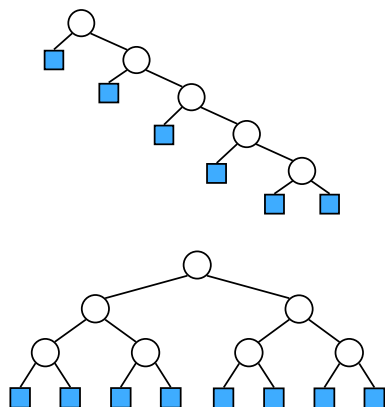
## Effektivitet

Betragt en ordbog med  $n$  emner, der er implementeret ved hjælp af et binært søgetræ med højden  $h$

Pladsforbruget er  $O(n)$

Metoderne **findElement**, **insertItem** og **removeElement** tager  $O(h)$  tid

Højden  $h$  er  $O(n)$  i værste tilfælde, og  $O(\log n)$  i bedste tilfælde



11

## Det gennemsnitlige tilfælde



Hvis  $n$  emner indsættes i tilfældig rækkefølge i et fra starten tomt binært søgetræ, vil længden af søgevejen i det frembragte træ i gennemsnit være  $1.39 \log_2 n$

I praksis er udførelsestiden for de tre grundoperationer  $O(\log n)$  for tilfældigt input. Det er dog endnu ikke blevet påvist analytisk

12

## Balancerede søgetræer



**Balancering** er en teknik, der **garanterer**, at de værste tilfælde ved søgning ikke forekommer

Ideen er at omorganisere træet under indsættelse og sletning, så det bliver fuldt (eller næsten fuldt)

Et fuldt træ siges at være i **perfekt balance**. For enhver knude gælder nemlig, at antallet af knuder i dens venstre undertræ er lig med antallet af knuder i dens højre undertræ

I det følgende præsenteres en række datastrukturer, der **garanterer**  $O(\log n)$  kompleksitet for såvel søgning, indsættelse som sletning

13

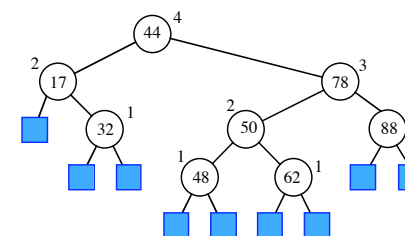
## AVL-træer

(Adelson-Velskii og Landis, 1962)

### AVL-træer er balancerede

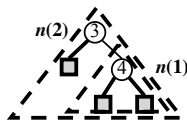
Et AVL-træ er et binært søgetræ, hvor det for enhver intern knude gælder, at højderne af dens børn højst afviger med 1

Eksempel på et AVL-træ  
Højderne er angivet ved siden af knuderne



14

## Højden af et AVL-træ



**Sætning:** Højden af et AVL-træ med  $n$  nøgler er  $O(\log n)$

### Bevis:

Lad  $n(h)$  betegne det minimale antal interne knuder i et AVL-træ med højden  $h$

Det er let at se, at  $n(1) = 1$  og  $n(2) = 2$

For  $h > 2$  indeholder et AVL-træ med højden  $h$  og med et minimalt antal interne knuder

- roden
- et AVL-træ med højden  $h-1$
- et AVL-træ med højden  $h-2$

Derfor gælder  $n(h) = 1 + n(h-1) + n(h-2)$

Da  $n(h-1) > n(h-2)$ , fås  $n(h) > 2n(h-2)$ , og derfor

$$n(h) > 4n(h-4), n(h) > 8n(h-6), \dots, n(h) > 2^i n(h-2i)$$

Ved løsning med basistilfældet,  $n(1) = 1$ , fås  $n(h) > 2^{h/2-1}$

Ved at tage logaritmen på begge sider af ulighedstegnet fås  $h < 2 \log n(h) + 2$

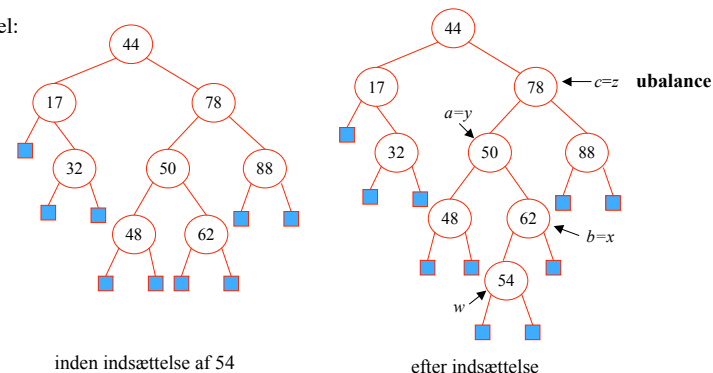
Derfor er højden af et AVL-træ  $O(\log n)$

15

## Indsættelse i et AVL-træ

Indsættelse sker i et binært søgetræ  
Sker altid ved at udvide en ekstern knude

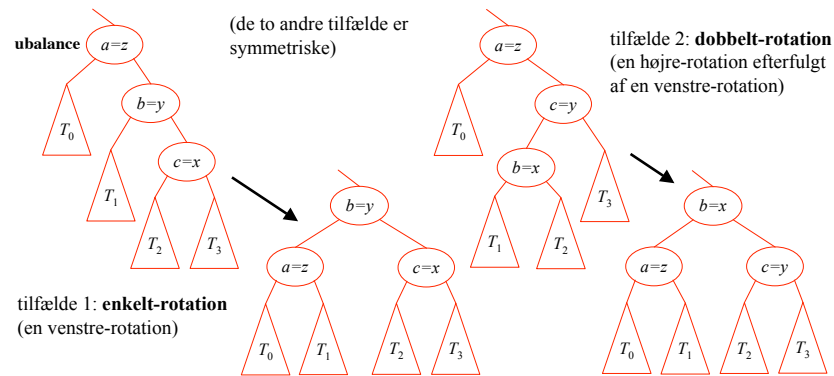
Eksempel:



16

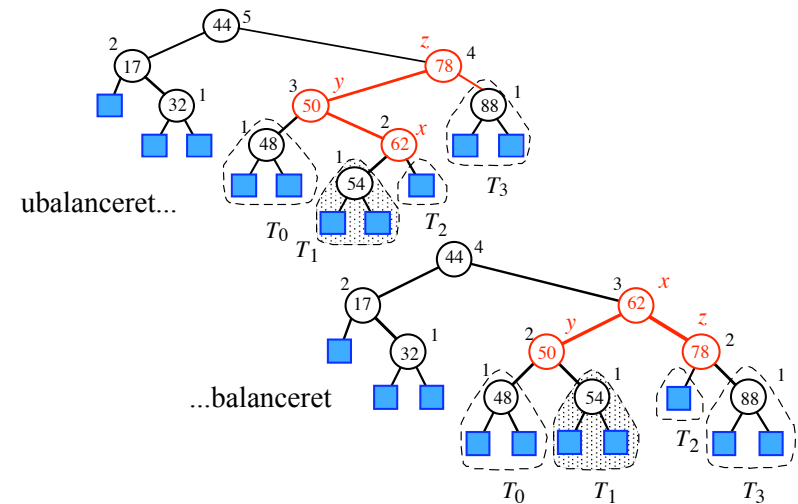
## Omstrukturering

Lad  $(a, b, c)$  være en inorder-opremsning af  $x, y, z$   
 Udfør de **rotationer**, der skal til for at gøre  $b$  til den øverste knude af de tre knuder



17

## Eksempel på indsættelse

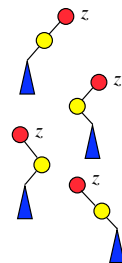


18

## Indsættelse i AVL-træ 4 tilfælde

Ved indsættelse i et af  $z$ 's undertræer er der 4 mulige tilfælde:

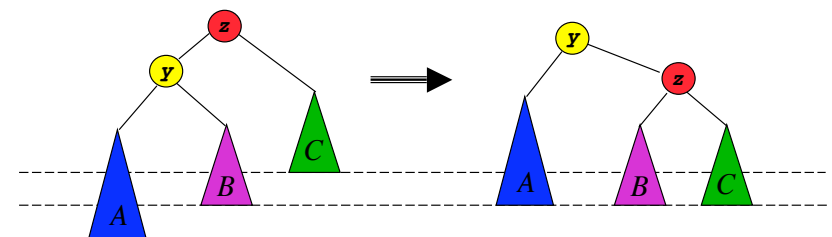
1. Indsættelsen sker i det *venstre* undertræ af  $z$ 's *venstre* søn.
2. Indsættelsen sker i det *højre* undertræ af  $z$ 's *venstre* søn.
3. Indsættelsen sker i det *venstre* undertræ af  $z$ 's *højre* søn.
4. Indsættelsen sker i det *højre* undertræ af  $z$ 's *højre* søn.



Tilfælde 1 og 4 er symmetriske.  
 Tilfælde 2 og 3 er symmetriske.

19

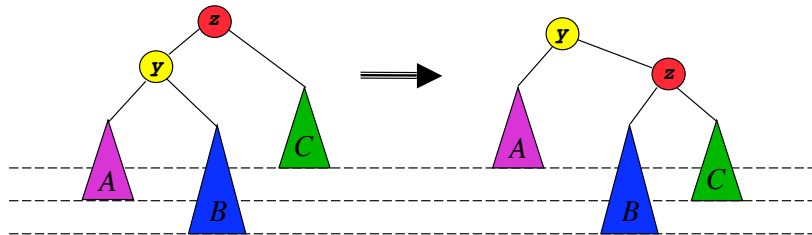
## Tilfælde 1



En højre-rotation genskaber balancen

20

## Tilfælde 2

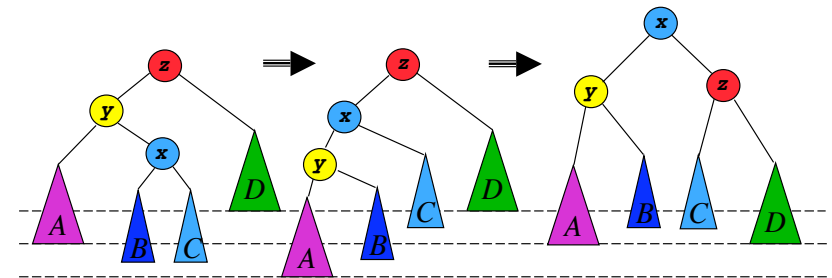


En højre-rotation genskaber **ikke** balancen

21

## Tilfælde 2

(fortsat)



En venstre-højre-dobbelrotation genskaber balancen

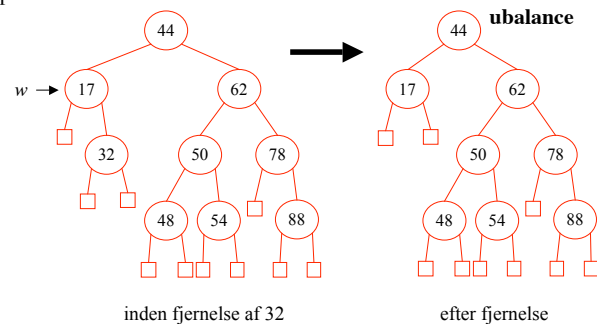
22

## Fjernelse fra et AVL-træ

Fjernelse starter som i et binært søgetræ

Forælderen  $w$  til en fjernet knude og dennes forfædre kan komme ud af balance

Eksempel:



inden fjernelse af 32

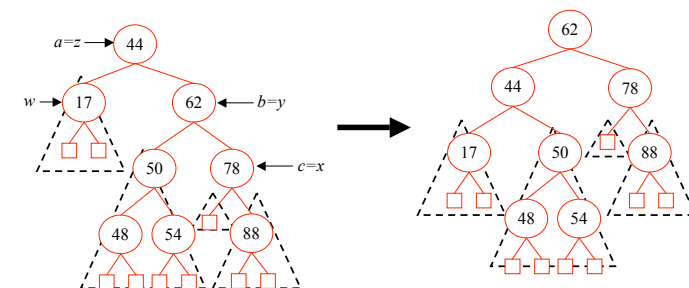
efter fjernelse

23

## Omstrukturering efter fjernelse

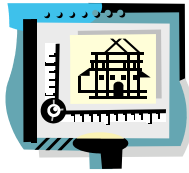
Lad  $z$  være den første ubalancerede knude, der mødes, når der gås op igennem træet fra  $w$ . Lad desuden  $y$  være det barn af  $z$ , der har størst højde, og lad  $x$  være det barn af  $y$ , der har størst højde

Balancen i  $z$  kan genskabes (med en enkelt- eller dobbelt-rotation)  
Men dermed kan balancen i en anden knude højere oppe i træet blive ødelagt  
Vi fortsætter derfor på samme måde, indtil roden nås



24

## Køretid for AVL-træer



Omstrukturering (rotation) tager  $O(1)$  ved brug af et hægtet binært træ

Søgning tager  $O(\log n)$   
højden af træet er  $O(\log n)$   
omstrukturering er ikke nødvendig

Indsættelse tager  $O(\log n)$   
søgningen tager  $O(\log n)$   
højest 1 omstrukturering, som tager  $O(1)$

Fjernelse tager  $O(\log n)$   
søgningen tager  $O(\log n)$   
omstrukturering op igennem træet tager  $O(\log n)$

25

## Flervejs-søgetræer

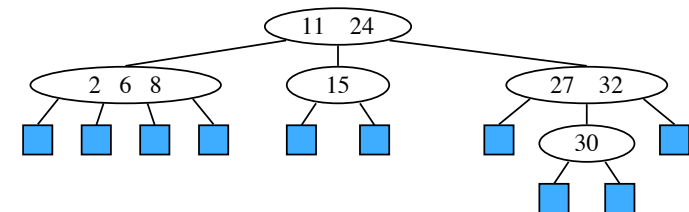
Et **flervejs-søgetræ** er et ordnet træ, hvor

Hver knude har mindst 2 børn og indeholder  $d-1$  nøgle-element-par  $(k_i, o_i)$ , hvor  $d$  er antallet af børn

For en knude med børn  $v_1 v_2 \dots v_d$  og nøglerne  $k_1 k_2 \dots k_{d-1}$  gælder

- nøglerne i undertræet med rod  $v_1$  er mindre end  $k_1$
- nøglerne i undertræet med rod  $v_i$  ligger imellem  $k_{i-1}$  og  $k_i$  ( $i = 2, \dots, d-1$ )
- nøglerne i undertræet med rod  $v_d$  er større end  $k_{d-1}$

Bladene indeholder ingen emner



26

## Flervejs-søgning

Svarer til søgning i et binært søgetræ

For hver interne knude med børnene  $v_1 v_2 \dots v_d$  og nøglerne  $k_1 k_2 \dots k_{d-1}$

$k = k_i$  ( $i = 1, \dots, d-1$ ): søgningen terminerer succesfuldt

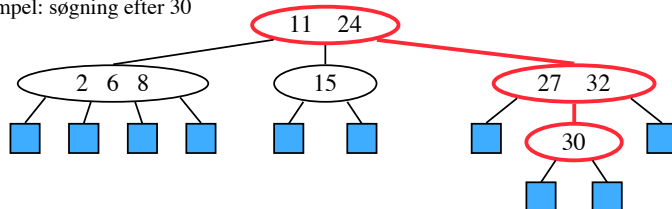
$k < k_1$ : søgning fortsætter med barnet  $v_1$

$k_{i-1} < k < k_i$  ( $i = 2, \dots, d-1$ ): søgningen fortsætte med barnet  $v_i$

$k > k_{d-1}$ : søgningen fortsætter med barnet  $v_d$

Hvis en ekstern knude nås, terminerer søgningen uden succes

Eksempel: søgning efter 30



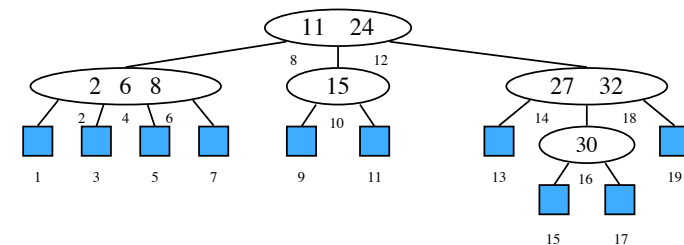
27

## Flervejs inorder-traversering

Vi kan udvide begrebet inorder-traversering fra binære træer til flervejs-søgetræer

Emnet  $(k_i, o_i)$  i en knude  $v$  besøges imellem de rekursive kald for de to undertræer, der har børnene  $v_i$  og  $v_{i+1}$  som rod

En inorder-traversering for et flervejs-søgetræ besøger nøglerne i stigende orden



28

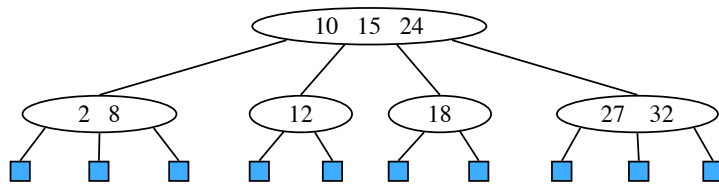
## (2,4)-træ

Et **(2,4)-træ** (også kaldet et **2-4-træ**, eller et **2-3-4-træ**) er et flervejs-træ med følgende egenskaber:

**Knude-størrelse-egenskaben:** enhver intern knude har højst 4 børn

**Dybde-egenskaben:** alle eksterne knuder har samme dybde

Afhængigt af antallet af børn, kaldes en intern knude for en **2-knude**, en **3-knude** eller en **4-knude**



29

## Højden af et (2,4)-træ

**Sætning:** Et (2,4)-træ med  $n$  emner har højden  $O(\log n)$

**Bevis:**

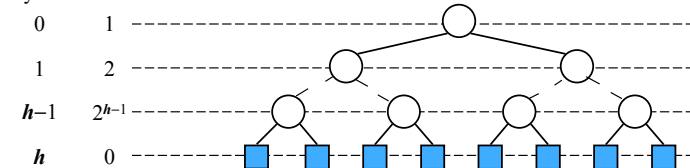
Lad  $h$  være højden af et (2,4)-træ med  $n$  emner

Da der er mindst  $2^i$  emner på dybden  $i = 0, \dots, h-1$  og ingen emner på dybden  $h$ , har vi

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

Ved at tage logaritmen på begge sider af ulighedstegnet fås  $h \leq \log(n+1)$

dybde emner



30

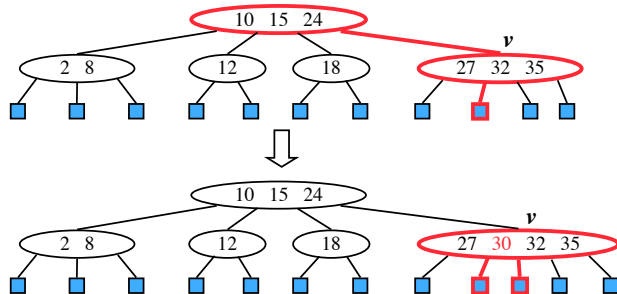
## Indsættelse

Et nyt emne  $(k, o)$  indsættes i forælderen  $v$  til det blad, der nås ved søgning efter  $k$

Dybde-egenskaben kan bevares, men

det kan resultere i **overløb** (knude  $v$  bliver til en 5-knude)

Eksempel: indsættelse af nøgle 30 giver overløb



31

## Overløb og splitting

Et overløb ved en 5-knude  $v$  håndteres med en splitting:

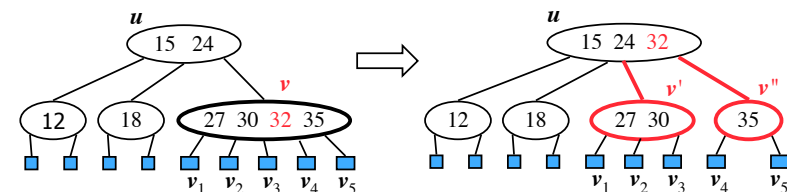
Lad  $v_1 \dots v_5$  være  $v$ 's børn og  $k_1 \dots k_4$  være  $v$ 's nøgler

Knude  $v$  erstattes med knuderne  $v'$  og  $v''$ , hvor

- $v'$  er en 3-knude med nøgler  $k_1, k_2$  og børn  $v_1, v_2, v_3$
- $v''$  er en 2-knude med nøgle  $k_4$  og børn  $v_4, v_5$

Nøglen  $k_3$  indsættes i forælderen  $u$  til  $v$  (en ny rod kan blive skabt)

Overløbet kan brede sig til forælderen  $u$



32



## Analyse af indsættelse

### Algorithm *insertItem(k, o)*

1. Vi søger efter nøglen  $k$  for at lokalisere indsættelsesknuden  $v$
2. Vi tilføjer det ny emne  $(k, o)$  til knuden  $v$
3. **while** *overflow*( $v$ ) **do**  
**if** *isRoot*( $v$ ) **then**  
     skab en ny tom rod over  $v$   
 $v \leftarrow$  *split*( $v$ )

Lad  $T$  være et (2,4)-træ med  $n$  emner

- $T$ 's højde er  $O(\log n)$
- Trin 1 tager  $O(\log n)$  tid, da vi besøger  $O(\log n)$  knuder
- Trin 2 tager  $O(1)$  tid
- Trin 3 tager  $O(\log n)$  tid, da hver splitning tager  $O(1)$  tid, og vi udfører  $O(\log n)$  splitninger

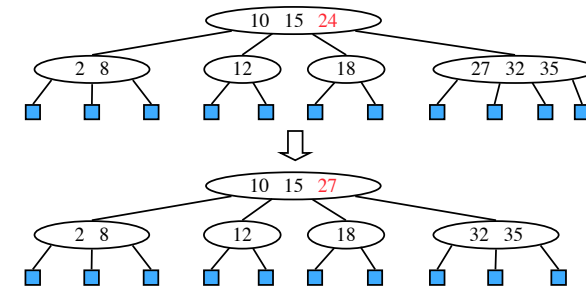
Derfor tager indsættelse i et (2,4)-træ  $O(\log n)$  tid

33

## Fjernelse

Fjernelse af et emne reduceres til det tilfælde, hvor emnet findes i en knude, der kun har blade som børn. I andre tilfælde erstattes emnet med dets inorder-efterfølger (eller med dets preorder-forgænger), og sidstnævnte emne fjernes

Eksempel: for at slette nøglen 24 erstattes den med 27 (dens inorder-efterfølger)



34

## Underløb og fusionering

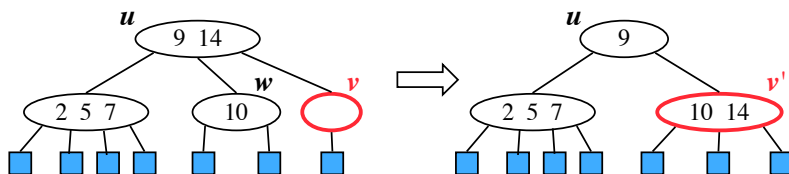
Fjernelse af et emne kan forårsage et **underløb**, hvor knuden  $v$  bliver til en 1-knude, d.v.s. en knude med et barn og ingen nøgler

For at behandle underløb betragtes to tilfælde:

**Tilfælde 1:** en tilstødende søskende er en 2-knude

**Fusionering:** vi sammensmelter  $v$  med en tilstødende søskende  $w$  og flytter et emne fra deres forælder  $u$  ned til den sammensmeltede knude  $v'$

Efter en fusionering kan underløbet brede sig til forælderen  $u$



35

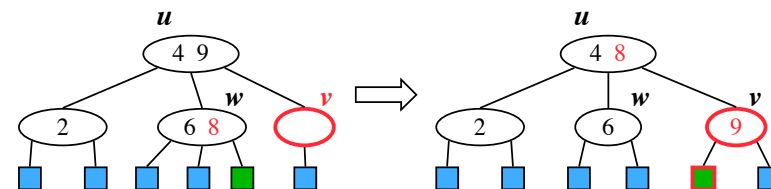
## Underløb og overførsel

**Tilfælde 2:** en tilstødende søskende,  $w$ , er en 3-knude eller en 4-knude

**Overførsel:**

1. Flyt et barn af  $w$  til  $v$
2. Flyt et emne fra  $w$  to  $u$
3. Flyt et emne fra  $u$  til  $v$

Efter en overførsel kan der ikke opstå underløb



36

## Analyse af fjernelse

Lad  $T$  være et (2,4)-træ med  $n$  emner  
 $T$ 's højde er  $O(\log n)$

Ved fjernelse

Besøges  $O(\log n)$  knuder for at lokalisere den knude,  
 hvorfra emnet skal fjernes

Vi behandler underløb ved en serie af  $O(\log n)$  fusioner,  
 efterfulgt af højst 1 overførsel

Hver fusion og overførsel tager  $O(1)$  tid

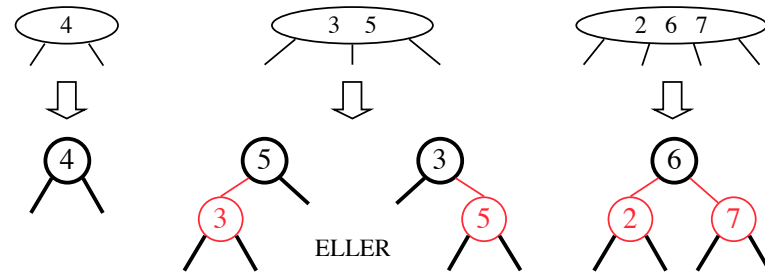
Fjernelse af et emne fra et (2,4)-træ tager  $O(\log n)$  tid

37

## Fra (2,4)- til rød-sort-træer

Et **rød-sort-træ** er en repræsentation af et (2,4)-træ ved hjælp af et binært træ, hvis knuder er farvet enten **rød** eller **sort**

Et rød-sort-træ har ligesom et (2,4)-træ logaritmisk effektivitet, men det er  
 simplet at implementere, da der kun er én knudetypen



38

## Rød-sort-træ

(Bayer, 1972)

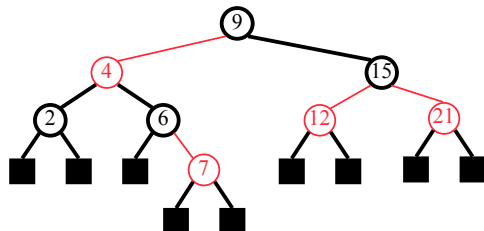
Et rød-sort-træ kan også defineres om et binært søgetræ, der har følgende fire egenskaber:

**Rod-egenskaben:** roden er sort

**Ekstern-egenskaben:** ethvert blad er sort

**Intern-egenskaben:** alle børn af en rød knude er sorte

**Dybde-egenskaben:** alle blade har samme sorte dybde



39

## Højden af rød-sort-træer



**Sætning:** Højden af et rød-sort-træ med  $n$  emner er  $O(\log n)$

**Bevis:**

Højden af et rød-sort-træ er højst 2 gange højden af sit tilsvarende (2,4)-træ,  
 som er  $O(\log n)$

Søgealgoritmen for et rød-sort-træ er den samme som for et binært søge-træ

Søgning i et rød-sort-træ tager derfor  $O(\log n)$  tid

40

## Indsættelse

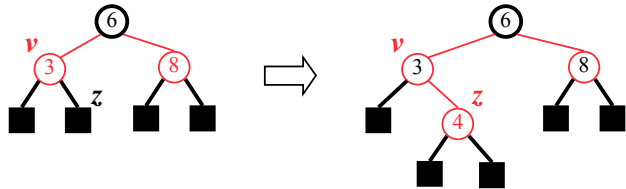
For at udføre operationen **insertItem**( $k, o$ ), udføres en indsættelse som for et binært søgetræ, og den indsatte knude farves **rød**, med mindre den er roden

Vi opretholder dermed rod-, eksternt- og dybde-egenskaben, men ikke nødvendigvis intern-egenskaben

Hvis forælderen,  $v$ , til  $z$  er sort, opretholdes også intern-egenskaben, og vi er færdige

Ellers ( $v$  er rød) har vi **dobbelt-rødt**, hvilket kræver en omstrukturering af træet

Eksempel, hvor indsættelsen af 4 giver dobbelt-rødt:



41

## Afhjælpning af dobbelt-rødt

Betragt en dobbelt-rød-situation med barn  $z$  og forælder  $v$ , og lad  $w$  være søskende til  $v$

Tilfælde 1:  $w$  er sort

Dobbelt-rødt svarer til en ukorrekt repræsentation af en 4-knude

Tilfælde 2:  $w$  er rød

Dobbelt-rødt svarer til overløb

**Omstrukturering:** vi ændrer 4-knude-repræsentationen

**Omfarvning:** Vi udfører en operation svarende til en splitning



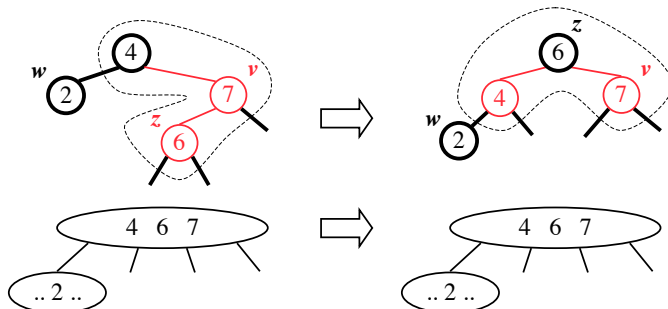
42

## Omstrukturering

En omstrukturering afhjælper barn-forældre-dobbelt-rødt, når forældreknuden har en sort søskende

Det svarer til at skabe en korrekt repræsentation af en 4-knude

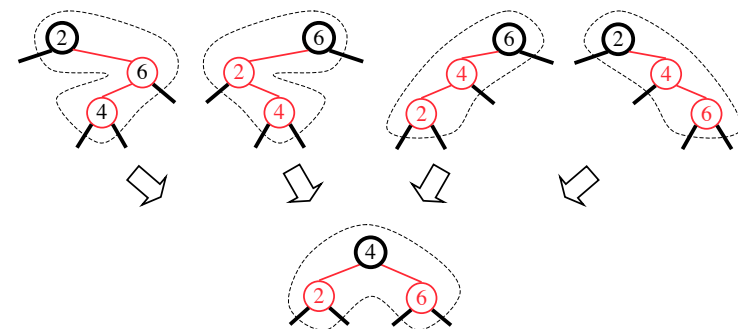
Intern-egenskaben genskabes, og de øvrige egenskaber bevares



43

## Omstrukturering (fortsat)

Der er fire muligheder for omstrukturering afhængigt af, om de dobbelt-røde knuder er venstre eller højre-børn



44

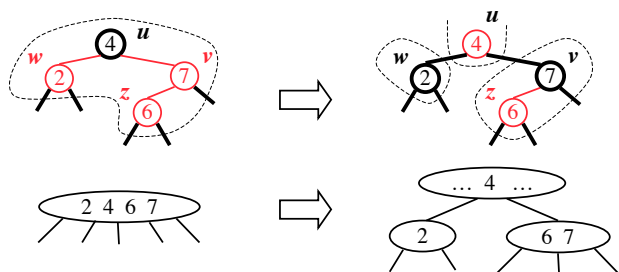
## Omfarvning

En omfarvning afhjælper børn-forælder-dobbelt-rødt, når forælder-knuden har en rød søskende

Forælderen  $v$  og dens søskende  $w$  farves sort, og bedsteforælderen farves rød, medmindre den er roden

Det svarer til at udføre en splitning af en 5-knude

Dobbelt-rødt kan brede sig til bedsteforælderen  $u$



45

## Analyse af indsættelse

Algorithm *insertItem*( $k, o$ )

1. Vi søger efter nøglen  $k$  for at lokalisere indsættelsesknuden  $z$
2. Vi tilføjer det ny emne ( $k, o$ ) til knuden  $z$  og farver knuden rød
3. **while** *doubleRed*( $z$ ) **do**  
     **if** *isBlack*(*sibling*(*parent*( $z$ ))) **then**  
          $z \leftarrow \text{restructure}(z)$   
     **else**  
          $z \leftarrow \text{recolor}(z)$

Lad  $T$  være et (2,4)-træ med  $n$  emner

- $T$ 's højde er  $O(\log n)$
- Trin 1 tager  $O(\log n)$  tid, da vi besøger  $O(\log n)$  knuder
- Trin 2 tager  $O(1)$  tid
- Trin 3 tager  $O(\log n)$  tid, da vi udfører  $O(\log n)$  omfarvninger, som hver tager  $O(1)$  tid, samt højst en omstrukturering, som tager  $O(1)$  tid

Derfor tager indsættelse i et rød-sort-træ  $O(\log n)$  tid

46

## Fjernelse

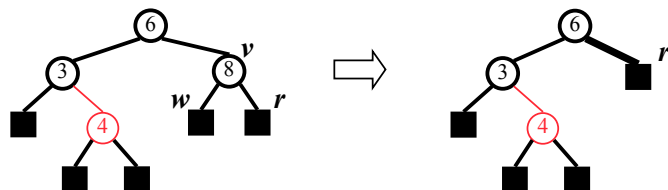
For at udføre operationen *remove*( $k$ ), udføres en fjernelse som for et binært søgetræ

Lad  $v$  være den fjernede interne knude,  $w$  den fjernede eksterne knude, og lad  $r$  være  $w$ 's søskende

Hvis enten  $v$  eller  $r$  var røde, farver vi  $r$  sort, og vi er færdige

Ellers (både  $v$  og  $r$  var sorte) farver vi  $r$  **dobbelt-sort**, hvilket ødelægger internegenskaben og nødvendiggør en omstrukturering

Eksempel: Fjernelse af 8 giver dobbelt-sort



47

## Afhjælpning af dobbelt-sort

Algoritmen til afhjælpning af dobbelt-sort knude  $r$  med søskende  $y$  behandler tre tilfælde:

**Tilfælde 1:**  $y$  er sort og har et rødt barn

Der udføres en omstrukturering (svarende til en overførsel), og vi er færdige

**Tilfælde 2:**  $y$  er sort, og begge dens børn er sorte

Der udføres en omfarvning (svarende til fusionering), hvilket kan give dobbelt-sort opad i træet

**Tilfælde 3:**  $y$  er rød

Der udføres en justering (svarende til valg af anden repræsentation for en 3-knude), hvorved tilfælde 1 eller 2 opstår

Fjernelse fra et rød-sort-træer tager  $O(\log n)$  tid

48

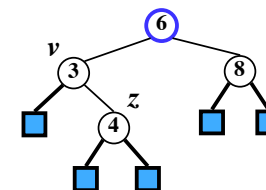


## AVL-træer contra Rød-Sort-træer

Et rød-sort-træ er sædvanligvis mere effektivt

Indsættelse i et AVL-træ kræver (i værste tilfælde) 2 gennemløb af en vej (ned til et blad og tilbage til roden), mens indsættelse i et rød-sort-træ kan klares med 1 gennemløb

## Splay-træer



## I splay-træer foretages rotation efter enhver operation (selv efter søgning)

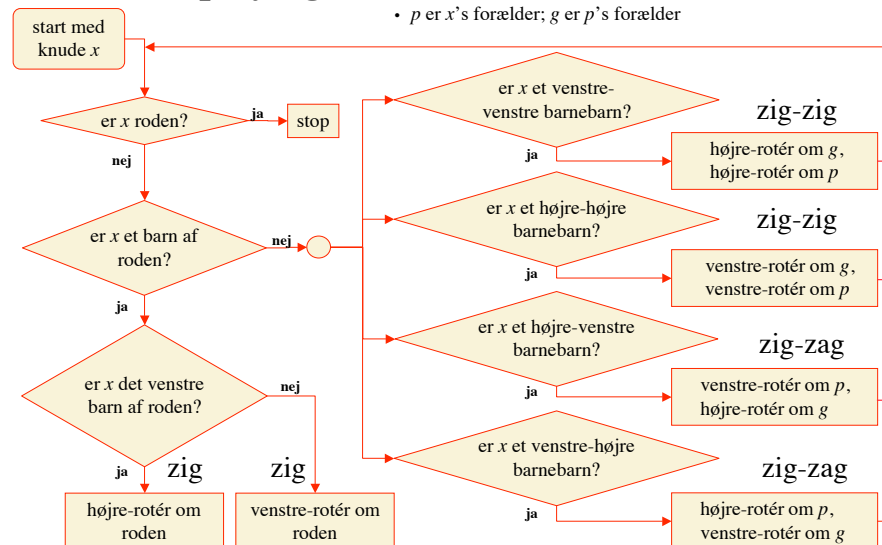


Ny operation: *splay*  
flytter en knude op som rod ved hjælp af rotationer

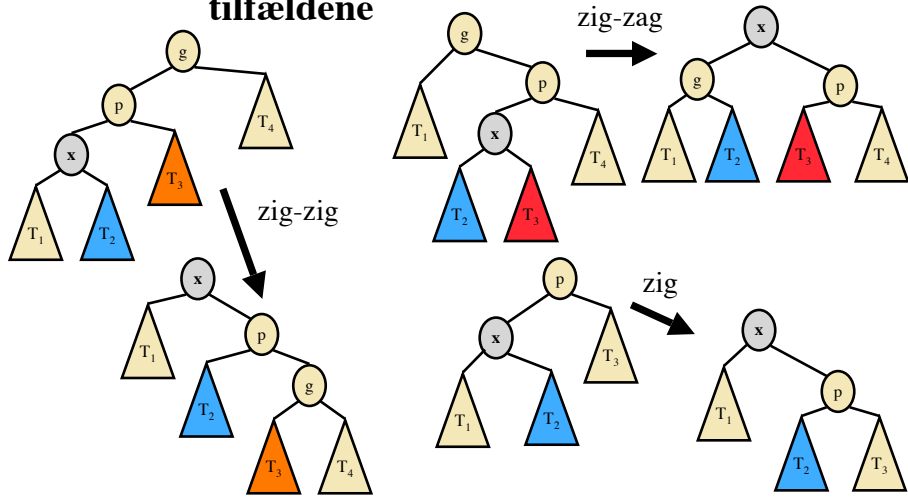
to splay  $\approx$  at forvride

## Splaying:

- at "x er et højre-venstre barnebarn" betyder, at x er et højre barn af sin forælder, som selv er et venstre barn af sin forælder
- p er x's forælder; g er p's forælder

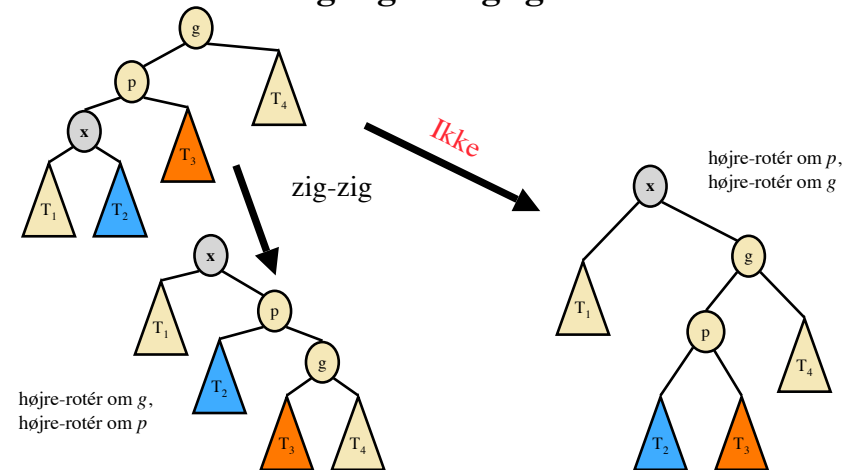


## Visualisering af tilfældene



53

## Rækkefølgen af rotationer for zig-zig er vigtig

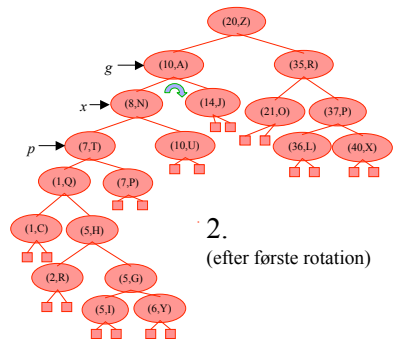


54

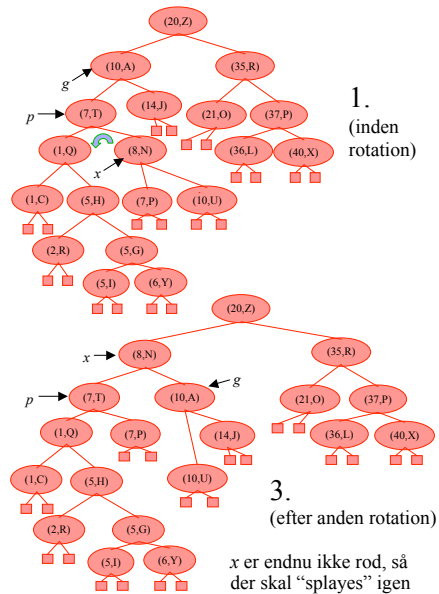
## Eksempel

Lad  $x = (8, N)$

$x$  er det højre barn af sin forælder, der er det venstre barn af bedsteforælderen venstre-rotér om  $p$  og højre-rotér om  $g$  (zig-zag)



2. (efter første rotation)



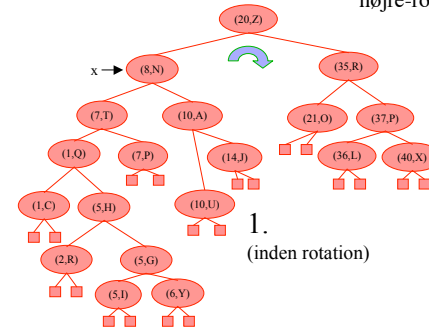
3. (efter anden rotation)

$x$  er endnu ikke rod, så der skal "splayes" igen

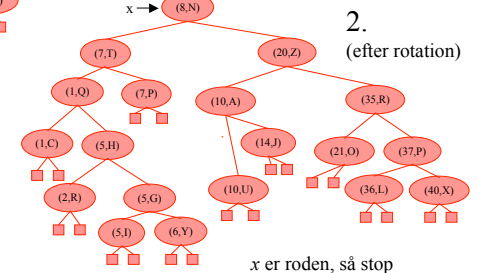
55

## Eksempel (fortsat)

nu er  $x$  venstre barn af roden højre-rotér om roden (zig)



1. (inden rotation)



2. (efter rotation)

$x$  er roden, så stop

56

## Definition af splay-træ



Et **splay tree** er et binært søgetræ, hvor hver knude “splayes”, efter at den er tilgået (efter en søgning eller en opdatering)

Den dybeste interne knude, der er tilgået, bliver splayet

Splaying koster  $O(h)$ , hvor  $h$  er højden af træet - i værste tilfælde  $O(n)$  [  $O(h)$  rotationer, som hver koster  $O(1)$  ]

57

## Splay-knuder



Hvilken knude splayes efter en operation?

operation	splay-knude
findElement	hvis nøglen findes i en knude, da denne knude hvis nøglen ikke findes, så forælderen til den eksterne knude, hvor søgningen stoppede
insertElement	den nye knude, der indeholder det indsatte element
removeElement	forælderen til den interne knude, der blev fjernet

58

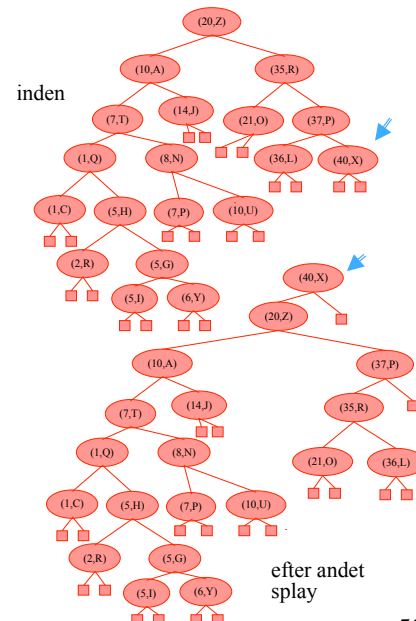
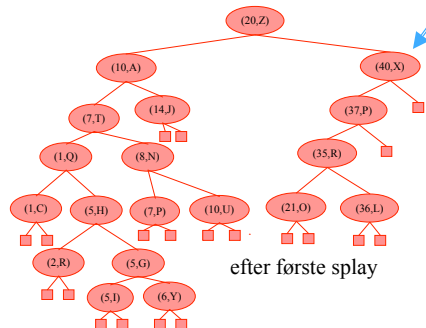
## Eksempel på resultatet af splaying

Træet behøver ikke at blive mere balanceret

Eksempel: splay (40,X)

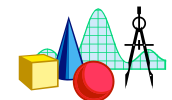
før er dybden af det højeste blad 3,  
og dybden af det dybeste er 7

efter er dybden af det højeste blad 1,  
og dybden af det dybeste er 8



59

## Amortiseret analyse af splay-træer



Køretiden for enhver operation er proportional med den tid, der bruges på splaying

Omkostninger: zig = 1 krone, zig-zig = 2 kroner, zig-zag = 2 kroner

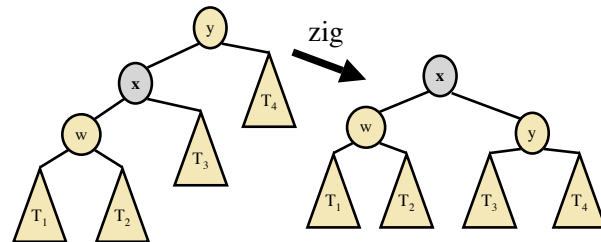
Dermed bliver omkostningen for at splaye en knude, der har dybde  $d$ , lig med  $d$  kroner

Definer  $\text{rank}(v)$  som logaritmen til antallet af knuder i undertræet med rod  $v$

Antag, at vi lagrer  $\text{rank}(v)$  kroner i enhver knude  $v$  i splay-træet

60

## Omkostning per zig

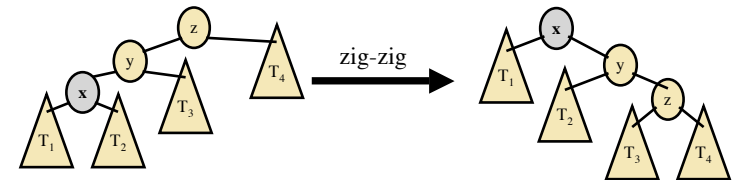


Udførelse af en zig koster højst  $\text{rank}'(x) - \text{rank}(x)$ :  
 $\text{cost} = \text{rank}'(x) + \text{rank}'(y) - \text{rank}(y) - \text{rank}(x)$   
 $\leq \text{rank}'(x) - \text{rank}(x)$

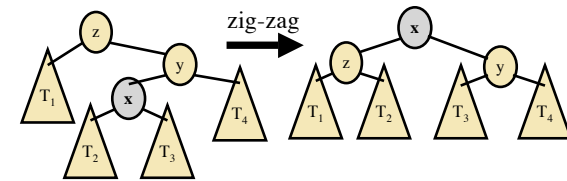
[  $\text{rank}'(y) \leq \text{rank}(y)$  ]

61

## Omkostning per zig-zig og zig-zag



Udførelse af zig-zig eller zig-zag koster højst  $3(\text{rank}'(x) - \text{rank}(x)) - 2$   
 Bevis: side 192 i lærebogen



62

## Omkostning for splaying



Omkostningen for at splaye en knude  $x$  på dybden  $d$  i et træ med rod  $r$  er højst  $3(\text{rank}(r) - \text{rank}(x)) - d + 2$

**Bevis:** Splaying af  $x$  bruger  $\lceil d/2 \rceil$  deltrin:

$$\begin{aligned} \text{cost} &= \sum_{i=1}^{\lceil d/2 \rceil} \text{cost}_i \\ &\leq \sum_{i=1}^{\lceil d/2 \rceil} (3(\text{rank}_i(x) - \text{rank}_{i-1}(x)) - 2) + 2 \\ &= 3(\text{rank}(r) - \text{rank}_0(x)) - 2(\lceil d/2 \rceil) + 2 \\ &\leq 3(\text{rank}(r) - \text{rank}(x)) - d + 2 \end{aligned}$$

63

## Effektiviteten af splay-træer



Husk, at rangen af en knude er logaritmen til dets undertræs størrelse  
 Derfor er den amortiserede omkostning for enhver splay-operation  $O(\log n)$ .

Splay-træer kan faktisk tilpasse sig, så der udføres søgning på ofte forespurgte emner hurtigere end  $O(\log n)$

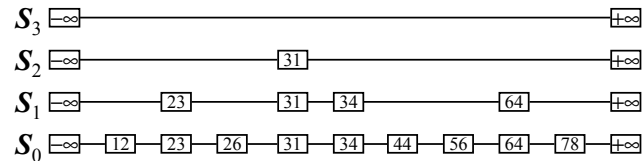
64



## Hvad er en skipliste?

En **skipliste** er en mængde  $S$  af forskellige (nøgle, element)-par i en række af lister  $S_0, S_1, \dots, S_h$ , hvor

- Hver liste  $S_i$  indeholder de specielle nøgler  $-\infty$  and  $+\infty$
- Listen  $S_0$  indeholder alle nøgler i  $S$  i ikke-aftagende orden
- Hver liste er en delsekvens af den forrige, d.v.s.  $S_0 \supseteq S_1 \supseteq \dots \supseteq S_h$
- Listen  $S_h$  indeholder kun de to specielle nøgler



65

## Søgning

Vi søger efter en nøgle  $x$  i en skipliste således:

Start i første position i den øverste liste

I den aktuelle position,  $p$ , sammenlignes  $x$  med  $y = \text{key}(\text{after}(p))$

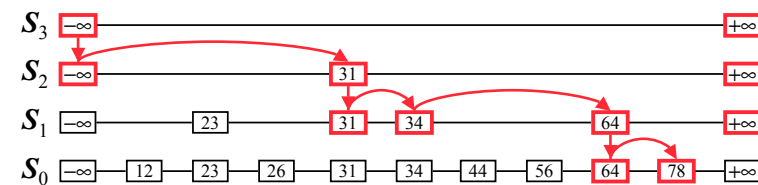
Hvis  $x = y$ , returneres  $\text{element}(\text{after}(p))$

Hvis  $x > y$ , søges videre "fremad"

Hvis  $x < y$ , søges videre "nedad"

Hvis der forsøges at gå nedad fra den nederste liste, returneres  $NO\_SUCH\_KEY$

Eksempel: søgning efter 78



66

## Randomiserede algoritmer

En **randomiseret algoritme** udfører møntkast (d.v.s. bruger tilfældige bit) til at styre sin udførelse

Den forventede køretid af en randomiseret algoritme analyseres ud fra følgende forudsætninger:

Den indeholder sætninger af typen

Mønten er "upartisk"  
Møntkastene er uafhængige

```

b ← random()
if b = 0 then
  do A ...
else { b = 1 }
  do B ...
    
```

Køretiden for det værste tilfælde for en randomiseret algoritme er ofte stor, men sandsynligheden for, at det indtræffer, er meget lille (det indtræffer, når alle møntkast giver samme udfald)

Køretiden afhænger af møntkastenes udfald

67

## Indsættelse

For at indsætte et emne  $(x, o)$  i en skipliste benyttes en randomiseret algoritme:

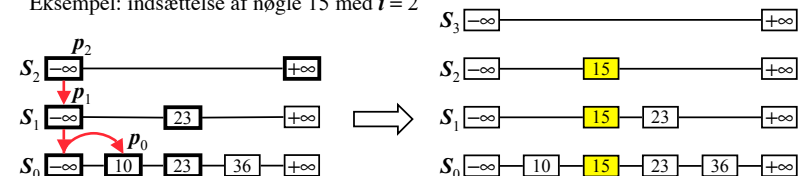
Vi kaster gentagne gange en mønt, indtil vi får "plat". Vi betegner med  $i$  det antal gange, vi har fået "krone"

Hvis  $i \geq h$ , tilføjes de nye lister  $S_{h+1}, \dots, S_{i+1}$ , som hver kun indeholder de to specielle nøgler

Herefter søges efter  $x$  i skiplisten, og vi bestemmer positionerne  $p_0, p_1, \dots, p_i$  på de emner, der har størst nøgle mindre end  $x$  i hver af listerne  $S_0, S_1, \dots, S_i$

For  $j \leftarrow 0, \dots, i$  indsættes  $(x, o)$  i listen  $S_j$  efter position  $p_j$

Eksempel: indsættelse af nøgle 15 med  $i = 2$

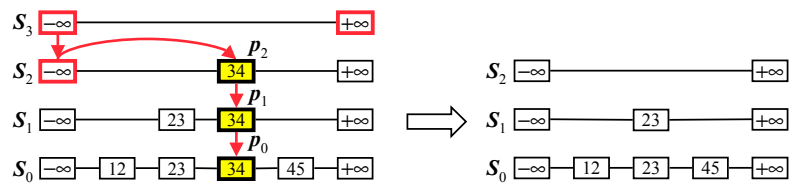


68

## Fjernelse

For at fjerne et emne med nøgle  $x$  fra en skipliste benyttes følgende fremgangsmåde:  
 Søg efter  $x$  i skiplisten og find positionerne  $p_0, p_1, \dots, p_i$  for de emner med nøgle  $x$ , hvor position  $p_j$  er i listen  $S_j$   
 Fjern positionerne  $p_0, p_1, \dots, p_i$  fra listerne  $S_0, S_1, \dots, S_i$   
 Med undtagelse af en, fjernes alle lister, der nu kun indeholder de to specielle nøgler

Eksempel: fjernelse af nøglen 34

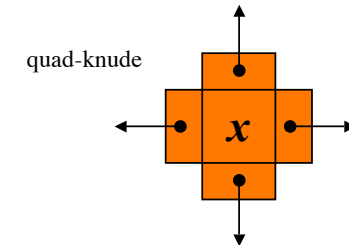


69

## Implementering

En skipliste kan implementeres ved hjælp af **quad-knuder**

En quad-knude indeholder:  
 emnet  
 hægte til knuden før  
 hægte til knuden efter  
 hægte til knuden under  
 hægte til knuden over



Desuden defineres de specielle nøgler MINUS\_INF og PLUS\_INF, og comparator-objektet ændres, så det kan håndtere disse nøgler

70

## Pladsforbrug

Pladsen, der bruges af en skipliste afhænger af de tilfældige bits, der bruges til indsættelsesalgoritmen.

Følgende sandsynlighedsteoretiske fakta benyttes:

**Faktum 1:** Sandsynligheden for at få  $i$  kroner i træk ved kast med en mønt er  $1/2^i$

**Faktum 2:** Hvis ethvert af  $n$  emner findes i en mængde med sandsynlighed  $p$ , vil det forventede antal elementer i mængden være  $np$

Betragt en skipliste med  $n$  emner

Ud fra faktum 1 indsættes et emne i listen  $S_i$  med sandsynligheden  $1/2^i$

Ud fra faktum 2 vil den forventede størrelse af listen  $S_i$  være  $n/2^i$

Det forventede antal knuder i en skipliste er

$$\sum_{i=0}^h \frac{n}{2^i} = n \sum_{i=0}^h \frac{1}{2^i} < 2n$$

Derfor er det forventede pladsforbrug for en skipliste med  $n$  emner  $O(n)$

71

## Højde

Køretiden for søgning og indsættelse i en skipliste afhænger af højden  $h$  af skiplisten

Vi kan vise, at en skipliste med  $n$  elementer har højde  $O(\log n)$  med høj sandsynlighed

Hertil benyttes endnu et sandsynlighedsteoretisk faktum:

**Faktum 3:** Hvis enhver af  $n$  hændelser har sandsynlighed  $p$ , vil sandsynligheden for at mindst én hændelse indtræffer være højst  $np$  (summen af deres sandsynligheder)

Betragt en skipliste med  $n$  emner

Ud fra faktum 1 indsættes et emne i listen  $S_i$  med sandsynligheden  $1/2^i$

Ud fra faktum 3 vil sandsynligheden for at listen  $S_i$  har mindst ét emne være højst  $n/2^i$

Ved at vælge  $i = 3 \log n$  fås, at sandsynligheden for at  $S_{3 \log n}$  har mindst ét element er højst

$$n/2^{3 \log n} = n/n^3 = 1/n^2$$

En skipliste med  $n$  emner har derfor en højde, der højst er  $3 \log n$  med en sandsynlighed, der mindst er  $1 - 1/n^2$

72

## Effektivitet af søgning

Søgetiden for en skipliste er proportional med antallet af *nedadgående* skridt, plus antallet af *fremadgående* skridt

Når der søges fremad i en liste, vil enhver af de nøgler, der undersøges, ikke tilhøre en højere liggende liste

De nedadgående skridt er begrænset af højden af skiplisten og er således  $O(\log n)$  med høj sandsynlighed

Et fremadgående skridt er knyttet til et tidligere møntkast

Ud fra faktum 4 er det forventede antal fremadgående skridt i hver liste lig med 2

For at analysere de fremadgående skridt benyttes endnu et sandsynlighedsteoretisk faktum:

Derfor er det forventede antal fremadgående skridt  $O(\log n)$

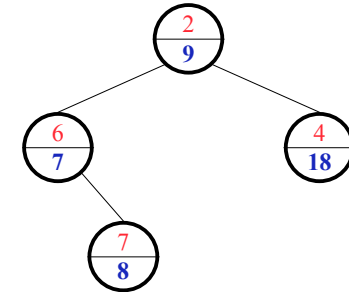
**Faktum 4:** Det forventede antal møntkast, der kræves for at slå plat, er 2

Vi kan derfor konkludere, at søgning forventes at tage  $O(\log n)$  tid

Analyse af indsættelse og fjernelse giver tilsvarende resultater

73

## Treaps



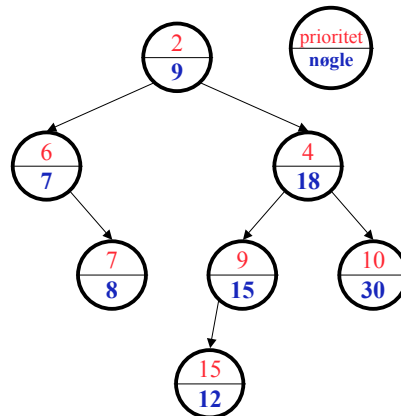
74

## Treap

En **treap** er et binært søgetræ, der samtidig opfylder hobbetingelsen

Prioriteterne vælges tilfældigt

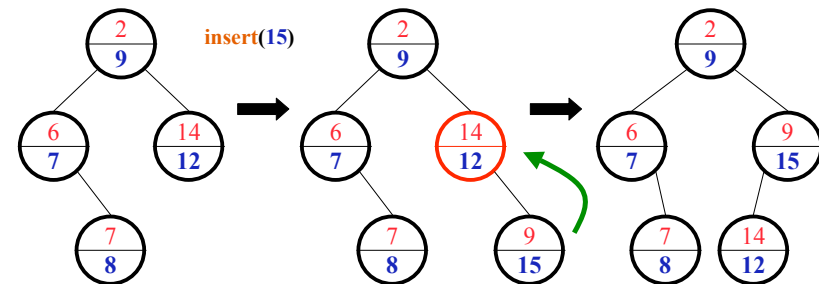
Indsættelse, søgning og fjernelse forventes at tage  $O(\log n)$  tid



75

## Indsættelse

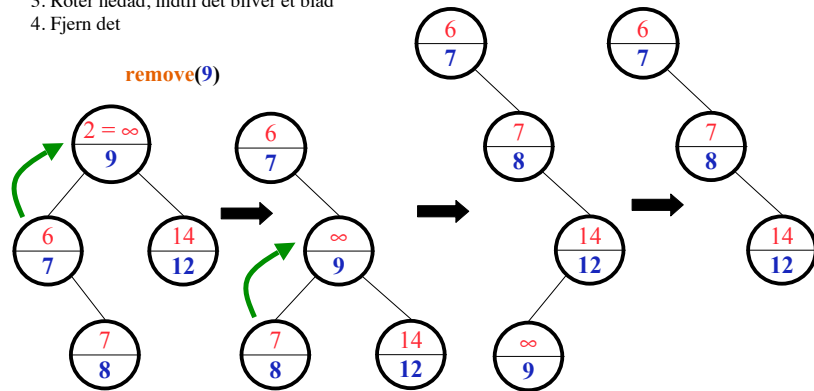
1. Vælg en tilfældig prioritet
2. Indsæt i det normale binære søgetræ
3. Roter opad, indtil hobbetingelsen er opfyldt



76

## Fjernelse

1. Find elementet, der skal fjernes
2. Sæt dets prioritet til  $\infty$
3. Roter nedad, indtil det bliver et blad
4. Fjern det



77

## Implementering af en ordbog

Sammenligning af effektive implementationer

	Søg	Indsæt	Fjern	Noter
Hashtabel	1 forventet	1 forventet	1 forventet	<ul style="list-style-type: none"> <li>• ingen ordning</li> <li>• simpel at implementere</li> </ul>
Skipliste Treap	$\log n$ høj sands.	$\log n$ høj sands.	$\log n$ høj sands.	<ul style="list-style-type: none"> <li>• randomiseret indsættelse</li> <li>• simpel at implementere</li> </ul>
(2,4)-træ Rød-sort træ	$\log n$ værste tilfælde	$\log n$ værste tilfælde	$\log n$ værste tilfælde	<ul style="list-style-type: none"> <li>• svær at implementere</li> </ul>

78