17
Søgning og Søgetræer.

Lineær og binær søgning i lister.

Binære søgetræer.

Søgning efter knude i træ.

Indsættelse af knude i træ.

Søgning i og sortering af binært søgetræ.

Sletning af knude i binært søgetræ.

Balanceringskriterier.

AVL-træer.

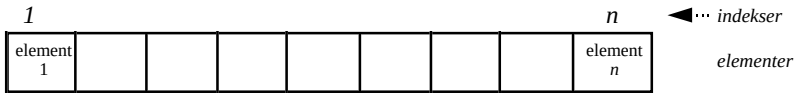
Indsættelse af knude i AVL-træer.

PS1 -- Søgning og søgetræer

Noter

Der er skrevet mange bøger, som behandler søgning. Det er dog værd at fremhæve én frem for alle andre: Knuth, *The art of Computer Programming (vol 3), Sorting and Searching*, Addison-Wesley. Selv om bogen er relativ gammel (fra 1973), indeholder den meget grundige diskussioner (intuitive såvel som tekniske), algoritmer (i pseudokode såvel som i assembler) og analyser.

Søgning i lister og arrays.



- **Lineær søgning i en liste.**
  - Elementerne i listen gennemsøges én for én (1, 2, ...), indtil listen er gennemløbet, eller søgeelementet er fundet.
  - Køretid: O(n).
- **Binær søgning i et *sorteret array* .**
  - Antag at elementerne i array-et er ordnet i stigende rækkefølge.

```
Binær-søgning(A: array,  e: element): indeks
    if A er tom  then returner ikke fundet
    elseif A[midterste indeks] = e
    then  returner midterste indeks
    elseif e er mindre end midterste element
    then  returner Binær-søgning(venstre-halvdel (A), e)
    else returner Binær-søgning(højre-halvdel (A), e)
```

  - Køretid: O(log n).

PS1 -- Søgning og søgetræer

Noter

Binær søgning er behandlet i afsnit 2.4.4 i Weiss. Den binære søgning i Weiss i afsnit 2.4.4 er programmeret iterativt, i modsætning til den rekursive formulering på denne slide.

Lister kan enten implementeres som kædede lister eller ved brug af et array. Dette har vi diskuteret i kapitlet om arrays og lister. Lineær søgning kan udføres næsten lige effektivt på kædede lister og array-baserede lister.

Det er ikke effektivt at lave binær søgning på kædede lister. Årsagen er, at vi i binær søgning skal opsøge det midterste element i listen; i en kædet liste er dette relativt kostbart, nemlig O(n). På denne slide har vi derfor formuleret binær søgning på arrays. Tidskompleksiteten log(n) for binær søgning fremkommer ved, at vi halverer søgeintervallet i hvert rekursivt kald.

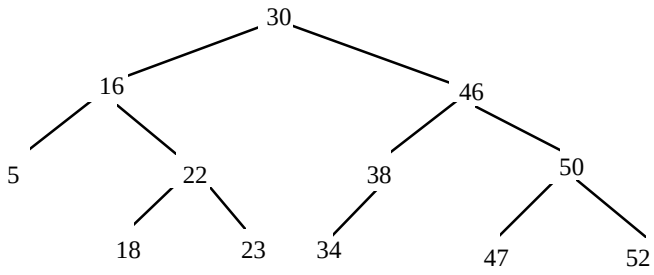
Algoritmer vil ofte i disse noter blive vist på skitseform, i en Pascal-agtig notation. Det er dermed vores mål at sætte fokus på den *algoritmiske ide* frem for detaljer. Der henvises til de benyttede lærebøger for en mere præcis og fyldestgørende beskrivelse af algoritmerne.

Binære søgetræer.

Lad < være en total ordning af knuderne i et binært træ.

Et binært træ T kaldes et søgetræ, hvis der for *alle* knuder N i T gælder at

- for *alle* knuder L i N's venstre undertræ: L < N.
- for *alle* knuder R i N's højre undertræ: N < R.



PS1 -- Søgning og søgetræer

Noter

En total ordning af en mængde er en ordning < som involverer alle mængdens elementer, således at for to *forskellige* elementer a og b i mængden gælder enten at a < b eller b < a. Endvidere skal < være transitiv. (Se også under kapitlet Sortering 1, slide “Sorteringsproblemet (1)”, og sammenlign også med en partiel ordning, som vi omtalte i kapitlet Nedarvning II da vi studerende klassen PART_COMPAR).

I eksemplet på denne slide er der vist et binært søgetræ hvori knudernes **nøgler** er heltal. Ordningen mellem knuderne er bestemt af ordningen mellem nøglerne. I dette tilfælde er ordningen afledt af den sædvanlige ordning "<" på heltal.

I søgetræet vist ovenfor nøjes vi med at vise nøglerne i knuderne. I en realistisk situation vil der være forskellige andre data, som er knyttet til nøglen. Disse andre data vil vi kalde *satellit-information*. Det er som regel satellit-informationen vi søger efter, givet vi har kendskab til nøglen.

Bemærk, at der er nøgler og data i såvel indre knuder som i blade. Senere i kurset vil vi møde træer, hvor indre knuder udelukkende udgør et *indeks* til data, som er gemt i bladene.

Når vi taler om søgetræer er det vigtigt, at vi tager udgangspunkt i ordnede træer. Binære træer er dog, pr. definition, altid ordnede. (Husk på, at dette betyder, at undertræerne af en knude er ordnede, se forelæsningen om træer).

Hvis en knude i et binært søgetræ kun har ét undertræ, er det endvidere vigtigt, at vi kan udnævne dette undertræ til enten at være det venstre eller det højre undertræ.

Opgave. Det er vigtigt at man via eksemplet bliver fortroligt med definitionen af binære søgetræer. Besvar derfor følgende spørgsmål: (1) Hvilke tal kan indsættes i stedet for knuden 34 i ovenstående træ, uden at ændre træets form? (2) Samme spørgsmål for knuden 18.

Søgning i binært søgetræ i Eiffel

```
has (v: like item): BOOLEAN is
 -- Is there a node with item 'v' in the tree?
 -- Nodes are compared with shallow equality.
require
 item_exists: v /= Void
do
    if equal(v, item) then
      Result := true
    elseif v < item then
      Result := not left_child.Void and then left_child.has (v)
    else
      Result := not right_child.Void and then right_child.has (v)
    end
end
```


Noter

På denne slide er funktionen *has* fra klassen BINARY_SEARCH_TREE vist. *Has* returnerer hvorvidt dataelementet v findes i det binære søgetræ. Operationen er taget fra Eiffel biblioteket (version 2.3, og tilpasset til Eiffel 3).

Typen af v er erklæret ved association til item, som er en anden feature i klassen BINARY_SEARCH_TREE. Se afsnit 12.15 i "Eiffel the Language" angående sådanne typer. Bemærk også brugen af "and then" i funktionen *has.*

Det naturlige resultat af en søgning er enten true eller false, altså boolsk. Et mere nyttigt resultat er derimod void eller en reference til en knude i træet. (I nogle binærer søgetræes ADT-er kan dette dog opfattes som et brud på den 'information hiding' vi ønsker os, nemlig hvis knuder er et internt begreb, som vi ikke ønsker klienter af binære søgetræer skal kende til). Med det alternative resultat i hånden kan vi undersøge eller ændre træet på det sted, hvortil søgningen bragte os.

Indsættelse af et element i et binært søgetræ.

```
Indsæt(e: element, T: binært-søgetræ)

  if T ikke er tom
  then if e < T.key
       then Indsæt(e, venstre undertræ af T )
       elseif e > T.key
       then Indsæt(e, højere undertræ af T  )
       else Indsæt en dublet af e i roden af T
  else Opret en ny knude K, som indeholder e ;
       Indsæt knuden K på T's plads
```

Degenererede søgetræer:

Et søgetræ kan udvikle sig til en enkeltkædet liste, hvis elementerne indsættes i stigende eller faldende orden.

Noter

Søgning i et binært søgetræ foregår helt analog til binær søgning i et ordnet array. Hvis man derimod skal indsætte eller slette elementer i samlingen, viser det sig, at binære søgetræer er overlegne i forhold til binær søgning i et array.

Mere konkret kan vi observere, at køretiden for *indsættelse* af et nyt element i et ordnet array er O(n), hvor n er antallet af elementer i array-et. Vi får samme dårlige tidskompleksitet ved indsættelse i en kædet liste, idet vi skal søge efter det korrekte indsættelsessted.

Indsættelse af et nyt element i et balanceret binært søgetræ vil i køretid være O(log(n)). Dette ses ved at observere, at antallet af beregningstrin, der er nødvendig for indsættelse af elementet, svarer til dybden af knuden *K* i ovenstående algoritme. Den maksimale dybde af *K* er træets højde.

Vi vender tilbage til balanceringsproblematikken senere i denne forelæsning, hvor vi også vil gøre det præcist, hvad vi mener, når vi taler om et *balanceret* binært søgetræ.

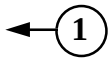
Det kan anbefales at studere Eiffel implementationen af ovenstående indsættelsesoperation. Der henvises til operationen *put* i klassen BINARY_SEARCH_TREE.

Sletning af et element i et binært søgetræ.

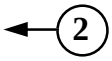
Slet(e: element, T: binært-søgetræ):

Find knuden K, som indeholder elementet e ;

If *K er et blad* **then**
Slet K umiddelbart



elseif *K har netop ét undertræ U* **then**
Lad roden af U indtage K´s plads i T



else *-- K har netop to undertræer:*
Slet det største element x i K's venstre undertræ ;
Erstat indholdet af K med x

Sletning af en knude i et binært søgetræ skal efterlade træet som et binært søgetræ.

PS1 -- Søgning og søgetræer

Noter

Lad os argumentere for, at ovenstående er korrekt, og at det efterlader træet som et binært søgetræ.

Tilfælde 1 er oplagt, og tilfælde 2 er næsten lige så let. Når den slettede knude K kun har ét undertræ U, vil alle knuderne i U opfylde ordnings-betingelserne i forhold til (en evt) far af K. I tilfælde 3 erstatter vi K med en nøje udvalgt knude, som vi her vil kalde X. X kan vælges som knuden, der indeholder det største element i K's venstre undertræ.

1. X kan slettes fra træet, idet det enten er et blad, eller det har netop et venstre undertræ. I modsat fald ville X ikke være det største element i K's venstre undertræ.

2. Måden hvorpå vi sletter X fra træet falder altså under tilfælde 1 eller 2 i ovenstående algoritme.

3. X erstatter nu K. Vi skal overbevise os om, at ordnings-relationerne mellem X og dens far hhv. børn er opfyldt:

4. Hvis vi antager, at K var rod i højre undertræ af sin far, ved vi at alle knuder i dette undertræ er større end K´s far. Dette gælder også for X. Altså opfylder X betingelser ift. sin nye far. Tilsvarende argumenteres, hvis K er rod i venstre undertræ. (Tænk over hvad der gælder, hvis K er rod i T).

5. X skal være større end alle knuder i sit nye venstre undertræ. Da X er udvalgt som det største element i dette undertræ, må alle andre elementer være mindre.

6. X skal endvidere være mindre end alle knuder i sit nye højre undertræ. Men da alle knuder, og specielt X, i K's oprindelige venstre undertræ, var mindre end knuderne i K´s højre undertræ, er dette krav også opfyldt.

X kunne alternativt være valgt som det *mindste* element i K's *højre* undertræ.

Køretiden af sletningen af et element er O(log n). Sletningen involverer (1) lokalisering af en knude samt (2) sletningen af knuden. Bidrag 1 koster O(log n). Bidrag 2 er i tilfældene 1 og 2 konstante, og i tilfælde 3 O(log n). Alt dette under forudsætning af, at træet er balanceret.

Balanceringskriterier for binære træer.

Perfekt balance:

Et binært træ T er perfekt balanceret hvis der for *enhver* knude N i T gælder at

| antal-knuder(venstre-undertræ(N)) - antal-knuder(højre-undertræ(N)) | <= 1.

AVL balance:

Et binært træ T er balanceret efter AVL-kriteriet, hvis der for *enhver* knude N i T gælder, at

| højden(venstre-undertræ(N)) - højden(højre-undertræ(N)) | <= 1.

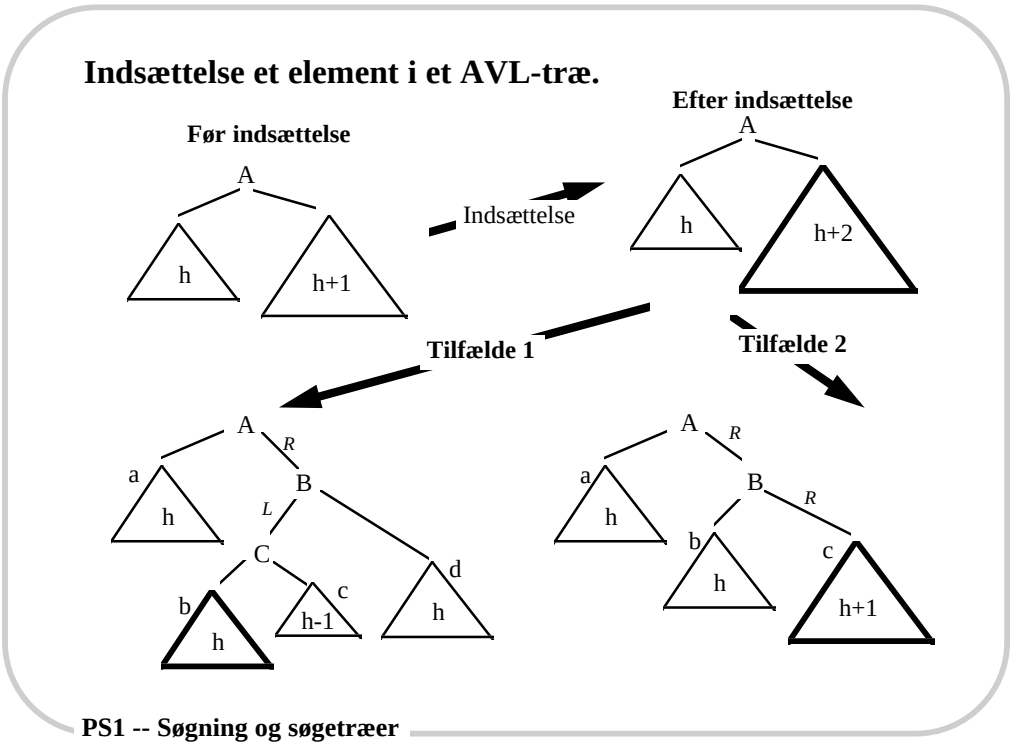
Noter

Vi har set, at den attraktive, logaritmiske tidskompleksitet af søgning, indsættelse og sletning opnåes, hvis træet er i en eller anden form for balance. Mere konkret kræves der, at træet ikke bliver for højt (se definitionen af højden af et træ i forelæsningen om træer). Både hvad angår søgning, indsættelse og sletning er tidskompleksiteten af operationerne begrænset af træets højde.

Vi har indset, at i visse uheldigesituationer kan træet degenerere til en kædet liste, i hvilket tilfælde træets højde bliver én mindre end antallet af knuder i træet.

Givet et (evt. degenereret) binært søgetræ kan man *transformere* det til et perfekt balanceret søgetræ. Det er imidlertid mere interessant og nyttigt at lade det binære søgetræ overholde en invariant, der sikrer at træet på ethvert tidspunkt (imellem træ-operationer) er balanceret.

På de følgende slides vil vi se, hvordan vi kan udforme en indsættelsesoperation, der opretholder AVL balancen af det binære søgetræ.



PS1 -- Søgning og søgetræer

Noter

På denne og følgende slide illustreres indsættelse af et nyt element i et binært søgetræ, der overholder AVL-kriteriet. Et sådant træ vil vi kalde for et AVL træ.

Det højde-forøgende element, som indsættes, er på figurerne indsat i de træer, der er tegnet med fede linier. Navne på knuder er med store bogstaver, og navne på (under)træer er med små bogstaver. Inde i træerne er angivet højden af træerne, som en funktion af en variabel h.

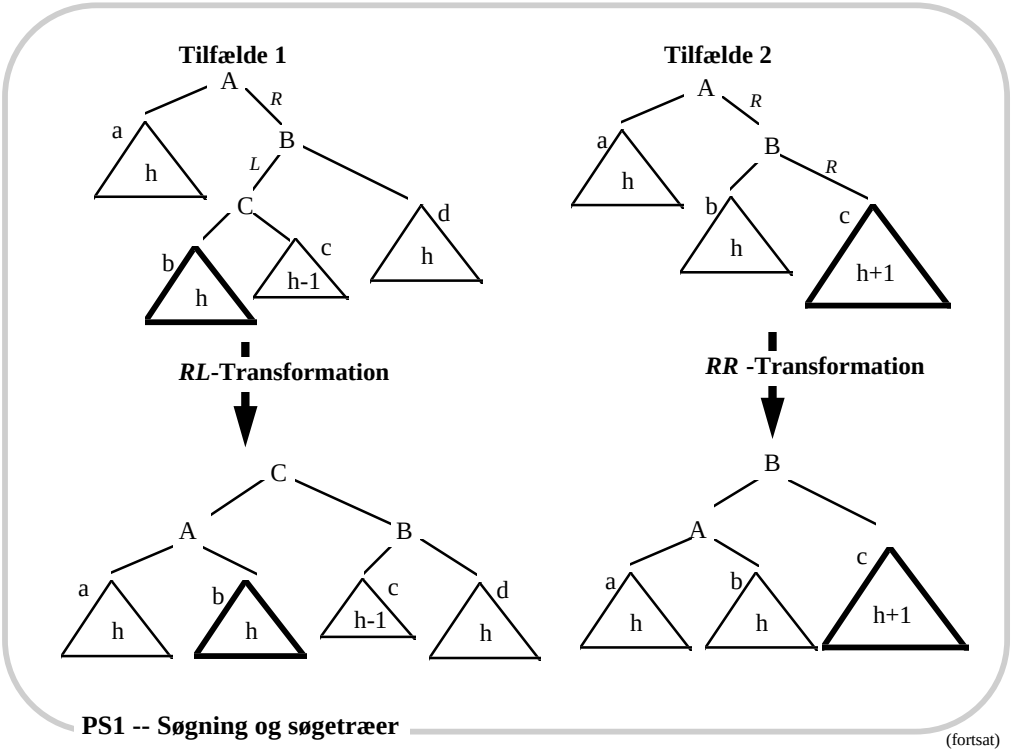
Vi kan uden tab af generalitet antage, at vi indsætter et højde-forøgende element i A's højre undertræ. Indsættelser i det venstre undertræ af A er, naturligvis, symmetriske.

Vi ønsker, at ubalancen skal forekomme i knude A. A er altså roden i det dybeste undertræ, hvori der forekommer AVL ubalance som følge af indsættelse af et element i træet.

Tilfælde 1: Vi indsætter et højdeforøgende element i B's venstre undertræ. Det venstre undertræ af B vil være af højde h+1. Vi vil inkludere roden af B's venstre undertræ, og kalde denne knude for C. Ubalancen opstår således enten ved, at der indsættes et højdeforøgende element i det venstre undertræ (b) eller i det højre undtræ (c) af C. Lad os her antage at højdeforøgelsen sker i b. (Det andet tilfælde er symmetrisk). Under denne antagelse har c højden h-1. Højden af træet d er h idet vi jo har antaget, at højdeforøgelsen sker i B's venstre undertræ, samt at ubalancen forekommer i knuden A, og ikke i B.

Tilfælde 2: Vi indsætter et højde-forøgende element i B's højre undertræ, som vi kalder c. Vi kunne naturligvis nuancere tegningen af c til at omfatte en knude (C) og to undertræer, men da den efterfølgende transformation ikke ændrer dette træ, er denne detalje i figuren unødvendig.

Tilfælde 1 identificeres som RL-tilfældet, idet ubalancen set i forhold til A forekommer ved først at følge referencen fra A til højre (Rigth) undertræ, og dernæst følge referencen til venstre (Left) undertræ. Af samme årsag identificeres tilfælde 2 som RR-tilfældet. De to tilfælde, vi ikke har vist på denne slide, kaldes hhv. LR og LL tilfældene.



PS1 -- Søgning og søgetræer

(fortsat)

Noter

Denne slide viser resultatet af transformationerne, ét for hvert af de to tilfælde vi identificerede på den forrige slide.

Øverst er vist de to træer, vi identificerede nederst på forrige slide. Husk igen på, at knuden A er den dybeste knude i træet, hvor vi har identificeret et brud af AVL balancekriteriet.

Bemærk, at undertræerne af de tegnede knuder ikke ændres overhovedet i transformationerne. Vi er således istand til at realisere transformationerne gennem nogle få manipulationer af referencer mellem knuder.

Der er to ting vi skal sikre os, når vi betragter resultatet af en transformation:

1. Er ordningen af knuder og deltræer uforandret (med andre ord: er det transformerede træ et binært søgetræ med de samme knuder som det oprindelige træ).

2. Er balanceringskriteriet opfyldt for det transformerede træ (med andre ord: er det transformerede træ et AVL-træ).

I tilfælde 1 er in-order ordningen a A b C c B d, såvel før som efter transformationen.

I tilfælde 2 er in-order ordningen a A b B c, såvel før som efter transformationen.

Det er let at overbevise sig om, at de resulterende træer er AVL-træer.

I en praktisk udformning af disse transformationer (ved programmeringen af operationer i træ-ADTen) vil vi antage, at vi direkte kan aflæse balanceringen af et træ. Dette kan naturligvis gøres ved at lagre højden af ethvert undertræ i træet; det er dog smartere at lagre den *relative balance* af et træ, hvilket fortæller, om træet hælder én til venstre, én til højre, eller om det er i balance.

Hvis vi sammenligner de transformerede træer med træet, som det så ud før indsættelsen, kan vi se, at disse træer alle er af højde h+2. Det medfører, at der ikke er flere knuder mellem det transformerede træ og roden, hvori der forekommer AVL ubalance. Med andre ord, er det kun nødvendigt at foretage én transformation for at genoprette balancen for hele træet.

Sortering af objekterne i et binært søgetræ.

Antag at der er behov for både effektiv

- søgning, indsættelse og sletning samt
- sortering af

af en samling af data med total ordning.

Samlingen af data repræsenteres som et balanceret binært søgetræ.

Operation	*Køretid*
• Søgning, indsættelse og sletning	*O(log n).*
• Sortering: in-order gennemløb af træet.	*O(n).*

Noter

Den bemærkelsesværdige tidskompleksitet af sorteringen beror naturligvis på, at data i et binært træ allerede er ordnede. Hvert element er omhyggeligt indsat på sin rette plads i træet for en pris, der er i størrelsesordenen O(log n). Hvis vi derfor ønsker at sortere en samling af data ved (1) at opbygge et binært søgetræ, og (2) at gennemløbe træet, er tidskompleksiteten ikke bedre end kendte, gode sorteringsalgoritmers tidskompleksitet.