16
Træer.

Definition af et træ.

Definitioner i tilknytning til træer.

Repræsentation af træer.

Binære træer.

Den abstrakte datatype.

Gennemløb af binære træer.

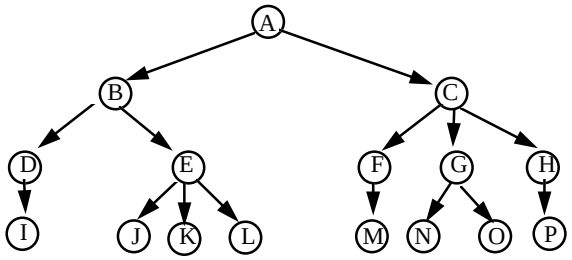
Træer i Eiffel.

Noter

Definition af et træ.

Et **træ** T af knudetype S er en samling af knuder. T skal enten være

1. en tom samling, eller
2. bestå af en **knude** af typen S, som kaldes **roden** af T, med et endeligt antal, disjunkte, tilknyttede træer (**undertræer**) af knudetype S.



PS1 -- Træer

Noter

Bemærk, at definitionen af et træ er rekursiv.

Når man tegner et træ, lader man som regel orienteringen af kanterne være implicit givne (oppe fra og ned), og man undlader at tegne "pilehovederne". Denne konvention vil vi følge i resten af disse noter.

Definitioner i tilknytning til træer (I).

Kant	Forbindelsen ("pilen") mellem en knude K og roden R i et undertræ af K. Kanten betegnes (K,R).
Sti fra N1 til Ni	En sekvens af kanter på formen: (N1,N2), (N2,N3), ... (Ni-1,Ni), i • 2
Efterkommer af knude K	En knude E, hvor der er en sti fra K til E.
Forgænger af knude K	En knude F, hvor der en sti fra F til K.
Blad	En knude, der ikke har nogen efterkommere.
Indre knude	En knude, som ikke er et blad.
Et ordnet træ.	Et træ, hvor mængden af undertræer af hver knude er ordnet.

Noter

I tilknytning til definitionen af træer kan man definere en mængde funktioner og prædikater på træer og knuder.

På denne og næste slide definerer vi en mængde sådanne funktioner. Man skal være opmærksom på, at der kan herske nogen terminologiforvirring mellem forskellige forfattere, og at det samme funktionsnavn undertiden bliver anvendt med forskellig betydning.

I definitionen af en **sti** betegner tuplen (N, M) to knuder, hvor M er rod i et undertræ af N. Med andre ord, og lidt mindre præcist, der er en kant (en "pil") fra N til M. Vi vil tale om en sti (af længde 0) fra en knude til sig selv.

Efterkommere og forgængere hedder på engelsk hhv. *decendants* og *ancestors*. Relativ til den måde, vi plejer at tegne træer på, er efterkommere af en knude k tegnet under k, og forgængere er tegnet over k. Vi taler også om **direkte efterkommere** og **direkte forgængere** i de tilfælde, hvor efterkommere og forgængere er forbundet til k med en sti af længde 1.

Et **blad** kaldes på engelsk for *leaf* eller *terminal.*

Hvis man lader sig inspirere af familieterminologi, er det naturligt at tale om **børn, forældre** og **søskende** af en knude.

Definitioner i tilknytning til træer (II).

Længden af en sti	Antallet af kanter i stien.
Dybden af en knude K i et træ T	Længden af stien fra roden af T til K.
Højden af en knude K i et træ T	Længden af den længste sti fra K til et blad i T.
Højden af et træ T	Højden af roden af T.
Graden af en knude K	Antallet af ikke-tomme undertræer af træet med rod K.
Graden af et ikke-tomt træ	Den maksimale grad af nogen knude i træet.

Noter

Denne slide definerer nogle kvantitative størrelser på stier, knuder og træer.

Ifølge ovenstående definitioner er dybden af roden 0

Ligeledes er højden af et blad 0.

Højden af et tomt træ sætter vi pr. definition til -1.

Dybden af en knude kaldes undertiden for **niveauet** (*level*) af knuden. Nogle forfattere definerer roden til at være på niveau 1. Vi holder fast ved, at dybden af roden er 0.

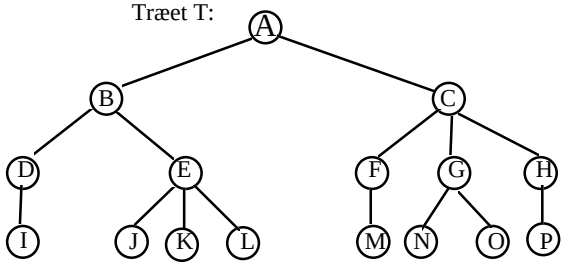
Man kan også tale om dybden af træet. Dette svarer til højden af træet.

Vi lader graden af et tomt træ være udefineret.

Opgave. Betragt en vilkårlig knude K i et træ T. Opstil en invariant, som relaterer højden af K, dybden af K og højden af T.

Eksempel.

Træet T:



Rod i T: A

Indre knuder: {A, B, C, D, E, F, G, H}

Blade: {I, J, K, L, M, N, O, P}

Graden af T: 3

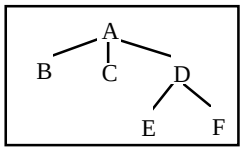
Højden af T: 3

Efterkommere af C: {F, G, H, M, N, O, P}
Direkte -- -- --: {F, G, H}
Forgængere af F: {C, A}
Direkte -- -- --: {C}
Graden af C: 3
Graden af F: 1
Dybden af B: 1
Højden af B: 2

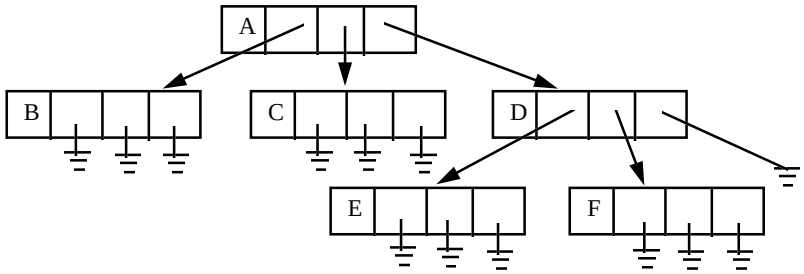
PS1 -- Træer

Noter

Repræsentation af træer (I).



Referencer til alle "børn".

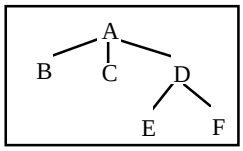


PS1 -- Træer

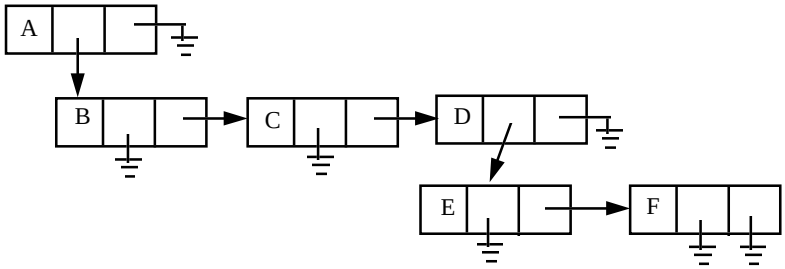
Noter

Denne repræsentation er rettet mod et træ af grad 3. I hvert knude-objekt er der afsat plads til netop 3 referencer til direkte efterkommere. Hvis der i træet er mange knuder, med variende grad, spildes der en del plads på dette. Det største pladsspild finder dog sted i bladene. Det er måske attraktivt at lade bladene være objekter af en specialiseret knudetype, som ikke allokerer plads til efterkommer-referencer. (Dette er ikke vist på denne slide).

Repræsentation af træer (II).



Reference til første "barn" og højre "søskende".



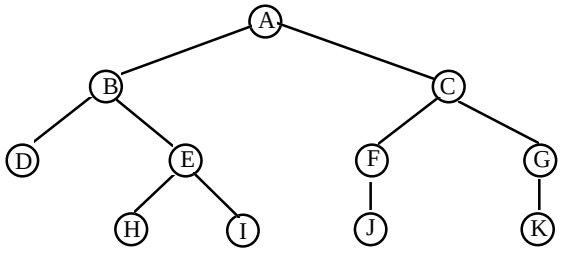
Noter

I denne repræsentation er undertræerne af en knude kædet sammen i en liste. Derfor er det tilstrækkeligt for en knude at referere til det første undertræ. Endvidere skal knuden referere til sit nabotræ (sit højre søskendetræ). Så uanset graden af træet (og graden af knuderne) indeholder hver knude-objekt netop to referencer.

Binære træer.

Et ordnet træ af grad 2 kaldes et **binært træ.**

Når en knude har grad 2, taler vi om det **venstre undertræ** og det **højre undertræ** af knuden.



PS1 -- Træer

Noter

Definitionen siger, at et binært træ er et træ af grad 2. Ser man tilbage på definitionen af 'graden af et træ' betyder dette altså, at der *maksimalt* må være to ikke-tomme undertræer af hver knude. En knude i et binært træ kan altså godt have ét undertræ.

Binære træer repræsenteres typisk med en reference til begge børn (altså som repræsentationen vist på slide "Repræsentation af træer (I)").

Selv om en knude K i et binært træ er af grad 1, angiver vi ofte, om knudens efterkommer er højre eller venstre efterkommer af K. Bemærk, at dette er muligt netop når vi vælger repræsentationen med direkte referencer til begge børn. Denne konvention viser sig at være praktisk i den specialisering af binærer træer, som vi kalder binære søgetræer.

Opgave: Beregn hvor mange knuder der maksimalt kan være i et binært træ af højde n.

Algebraisk specifikation af binære træer.

Type Binary-tree[X]
Functions
emptytree: -> Binary-tree[X]
make: Binary-tree[X] x X x Binary-tree[X] -> Binary-tree[X]
left: Binary-tree[X] -> Binary-tree[X]
right: Binary-tree[X] -> Binary-tree[X]
root: Binary-tree[X] -> X
isempty: Binary-tree[X] -> Boolean
height: Binary-tree[X] -> Integer
isin: X x Binary-tree[X] -> Boolean

Axioms
....

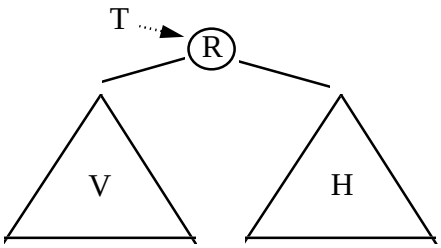
PS1 -- Træer

Noter

Opgave: Skriv ligningerne for ovenstående funktioner i den algebraiske specifikation af binære træer. Operationerne left, right og root selekterer hhv. venstre undertræ, højre undertræ og roden (af typen X) af et træ. Isempty returnerer, hvorvidt træet er tomt, og isin returnerer, om parameteren er at finde i træet. Endelig returnerer height højden af træet. Afprøv jeres specifikation i GAS.

Gennemløb af binære træer.

Et gennemløb af et binært træ giver en liste af knuder.
Hver knude i træet repræsenteres netop én gang i gennemløbet.



Pre-order(T) : insert(R, append(pre-order(V), pre-order(H)))

In-order(T): append(in-order(V), insert(R, in-order(H)))

Post-order(T): append(post-order(V), post-order(H), insert(R, empty-list()))

PS1 -- Træer

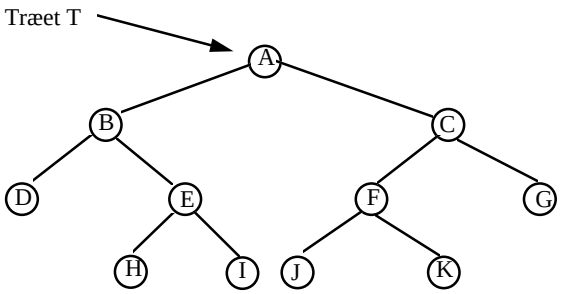
Noter

Sliden "Eksempel på gennemløb af binære træer (II)" giver motivationen for valg af *navnene* for ovenstående gennemløb af binære træer.

Ovenstående definitioner af de tre gennemløb viser ikke, hvad gennemløbene af tomme træer er. Det skulle dog ikke overraske nogen, at gennemløbet af et tomt træ er den tomme liste. Vi kan også sige, at et gennemløb af et træ med kun én knude giver listen bestående af denne knude.

Operationerne append, insert, og empty-list er listeoperationer. Disse er defineret på sliden "algebraisk specifikation af lister" fra kapitlet "Arrays og Lister samt Stakke og Køer". Bemærk dog, at vi har taget os den frihed at benytte append på tre lister.

Eksempel på gennemløb af binære træer (I).



pre-order(T): A B D E H I C F J K G
in-order(T): D B H E I A J F K C G
post-order(T): D H I E B J K F G C A

PS1 -- Træer

Noter

Opgave:

Man kan definere en manifest notation for binære træer på følgende måde:

(1) Det tomme træ skrives som ().

(2) Træet med rod r og undertræer T1 ... Tm skrives som (r T1 ... Tm).

(3) Ethvert undertræ, som kun består af en enkelt knude skrives uden paranteser.

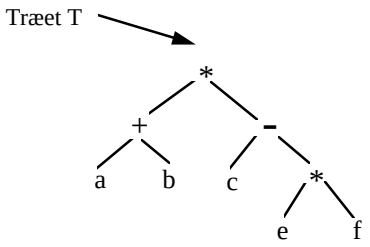
Denne notation er reelt en Lisp repræsentation.

A. Hvad er Lisp repræsentationen af det binære træ, som er vist på denne slide? Forsøg at gå systematisk til værks, frem for blot at opskrive træet på en lineær måde i "ét hug".

B1. *(For dem, der er gode til Lisp).* Definer Lisp funktioner, der realiserer pre-order, in-order, og post-order gennemløb af træer, og afprøv disse funktioner på det viste træ. (Til dette spørgsmål vil jeg anbefale, at man benytter Emacs og Emacs Lisp.)

B2. *(For dem, der ikke er gode til Lisp).* Sæt jer ind i, og afprøv de pre-order, in-order og post-order funktioner, jeg har programmeret på forhånd. Demonstrer, at ovenstående ordninger af det viste træ er korrekte. Hent funktionerne fra filen /user/normark/public/p1/tree-traversal.el. Se endvidere den lille brugervejledning i kørsel af Emacs lisp, som findes først i denne fil.

Eksempel på gennemløb af binære træer (II).



pre-order(T): * + a b - c * e f	(* (+ a b) (- c (* e f)))	prefix
in-order(T): a + b * c - e * f	((a + b) * (c - (e*f)))	infix
post-order(T): a b + c e f * - *		postfix

PS1 -- Træer

Noter

Træet repræsenterer strukturen og indholdet af et udtryk. Denne repræsentation opnås ved at parse et udtryk, som det defineres i programmeringssprog. Det er syntaks-analysens opgave at omdanne en streng, i f.eks. infix notation, til dette træ. Parsning og syntakstræer er vigtige emner på "Programmeringssystemer 2" på Dat2/D6D.

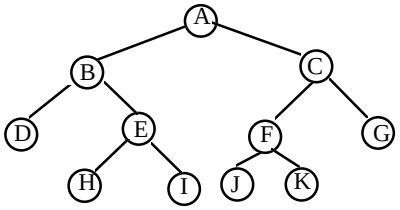
Det illustreres, at et in-order gennemløb af træet giver den sædvanlige infix notation for udtrykket. Dog skal der konsekvent sættes paranteser om deludtryk for at bevare informationen fra træet i inorder udtrykket.

Et pre-order gennemløb giver prefix notationen. Hvis man igen konsekvent sætter paranteser om deludtryk opstår Lisp notationen, også kaldet "Cambridge polsk" notation.

Et post-order gennemløb giver postfix notationen, også kaldet "omvendt polsk notation". Det er også notationen, som benyttes på visse typer af lommeregnere. Notationen er hensigtsmæssig idet den lægger op til en stakdisciplin, når udtrykket skal evalueres. Paranteser er ikke nødvendige i postfix notation for at bevare udtrykkets struktur.

Iterativt gennemløb af binære træer: Bredde først.

```
Bredde-først-gennemløb(T: Binært-træ): Liste
Q: Kø
  Q.enter(roden af T );
  while not Q.empty
  do let N = Q.front
        indsæt N i bagenden af resultat-listen ;
        Q.leave;
        for alle sønner S af N
        fra venste mod højre  do Q.enter(S)
```



Bredde-først gennemløb

(A B C D E F G H I J K)

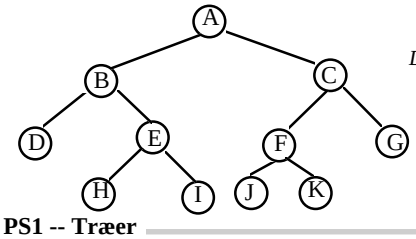
Noter

De forskellige former for rekursive gennemløb (pre-order, in-order og post-order gennemløb af binære træer), vi hidtil har studeret, har alle været såkaldte dybde-først gennemløb. Et rekursivt kald af gennemløbsfunktionen gennemløber et helt undertræ U uden at bevæge sig over i U's søskendetræer.

Gennemløbet, som er vist på denne slide er anderledes. Her gennemløbes træet på tværs af søskendetræer, et niveau ad gangen. Dette kaldes et bredde-først gennemøb. Rekursion er ikke velegnet til dette. Det er mere naturligt at programmere gennemløbet iterativt, ved hjælp af en kø. Vi benytter kø-operationerne enter, front og leave fra kapitlet om "Arrays og lister samt stakke og køer".

Iterativt gennemløb af binære træer: Dybde-først.

```
Dybde-først-gennemløb-in-order(T: Binært_træ): Liste
S: Stak;  U: Binært-træ
  repeat forever
     U := T,
     while U ikke tomt
     do  S.push(U);
         U := venstre undertræ af U
     if not S.empty
     then U := S.top; S.pop;
          indsæt roden af U i bagenden af resultat-listen ;
          U := højre undertræ af U
     else exit repeat
```



Dybde-først inorder gennemløb

(D B H E I A J F K C G)

PS1 -- Træer

Noter

Det iterativt programmerede, bredde-først gennemløb af et binært træ skærper vor interesse for iterativt programmerede dybde-først gennemløb. En sådan gennemløbsfunktion er vist på denne slide.

Det er ikke en overraskelse, at der internt i denne funktion skal benyttes en stak. Generelt realiseres rekursive procedure- og funktionskald på en stak. Stakken indeholder på ethvert tidspunkt aktiverede, men endnu ikke-afsluttede kald.

Stakken i ovenstående indeholder de undertræer, som vi har mødt i vort gennemløb, men som vi endnu ikke har kunnet udskrive roden af.

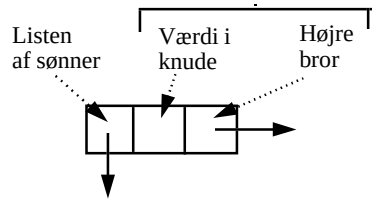
Dette eksempel viser med stor tydelighed værdien af rekursive procedurer og funktioner. Programmer bliver meget mindre gennemskuelige hvis vi eksplicit skal håndtere en stak i en iterativ beregningsfunktion. Man kan sige, at vi med rekursiv programmering abstraherer over de detaljer, som er vist i funktionen på denne slide.

Træer i Eiffel.

Et træ er en rod-knude, som har en liste af undertræer samt en reference til sin højre bror.

```
class Tree[T]
inherit
  Linkable[T]
    rename ...
    redefine ...
  Linked_list[T]

feature
  ...
end; --class Tree
```



Noter

Den trærepræsentation, som Eiffel lægger op til, svarer nøje til trærepræsentationen fra sliden "Repræsentation af træer (II)".

Det skitseres, hvordan Eiffel benytter multipel nedarvning til at beskrive de to aspekter af træ-knuder:

1. En træ-knude kan have en liste af undertræer (fra Linked_list).
2. En træ-knude har en værdi, og den kan have en højre bror (fra Linkable).

Detaljerne i ovenstående kan findes i A.7 i OOSC, og de er desværre relativt vanskelige at hitte ud af. På denne baggrund vil jeg selv foretrække en mere direkte og "lige ud af landevejen" definition af træer.