Indhold

1 Indledning

En *datatype* er defineret til at være en værdimængde, der er udstyret med en samling karakteristiske værdi- og variabeludtryk.

En *datastruktur* vil vi i denne sammenhæng definere til at være en konkret implementation af en abstrakt datatype. Forskellige datastrukturer for den samme datatype vil føre til forskellige effektivitetsegenskaber.

Denne note vil præsentere en samling klassiske datastrukturer, der implementerer velkendte og naturlige datatyper, som finder anvendelse i utallige algoritmer.

I adskillige tilfælde vil vi observere, at de indlysende implementationer giver *lineære* udførelsestider, men at vi ved at introducere *træstrukturer* kan opnå *logaritmiske* tider.

I [Programmeringsteori] er der i visse tilfælde angivet en eksplicit abstraktionsfunktion α. Af hensyn til at holde symbolmængden under kontrol vil den kun forekomme implicit i denne note. En specifikation som

$$\textbf{Insert}[P](x) \quad \textbf{sat} \quad P' = P \cup \{x\}$$

skal således læses som

$$\textbf{Insert}[P](x) \quad \textbf{sat} \quad \alpha(P') = \alpha(P) \cup \{\alpha(x)\}$$

Bemærk, at denne oversættelse altid er entydig, da programvariabler kun giver mening i abstraheret form til højre for **sat**.

2 Stakke og køer

To af de mest fundamentale datatyper er *stakke* og *køer*, som vi skal se på først.

2.1 Stakke

Vi er givet en elementtype E. En *stak* over E er en datatype, hvis værdimængde er følger af E-værdier. Lad S være en stakvariabel, og lad x være af type E. Så har stakken følgende udtryk

$$\begin{aligned}
\textbf{Init}[S] \ \ &\textbf{sat} \ \ S' = () \\
\textbf{Push}[S](x) \ \ &\textbf{sat} \ \ S = (e_0, e_1, \ldots, e_{n-1}) \rightarrow S' = (e_0, e_1, \ldots, e_{n-1}, x) \\
\textbf{Pop}[S, x] \ \ &\textbf{sat} \ \ (S = (e_0, e_1, \ldots, e_{n-1})) \wedge (n > 0) \rightarrow \\
&\qquad (S' = (e_0, e_1, \ldots, e_{n-2})) \wedge (x' = e_{n-1}) \\
\textbf{Empty}[S] \ \ &\textbf{sat} \ \ (S' = S) \wedge (\text{Empty}' = (S = ()))
\end{aligned}$$

En stak med fast øvre størrelse kan implementeres ved hjælp af en liste, så alle operationer finder sted i optimal tid. En polybox, der implementerer en stak med plads til N elementer, ser ud som følger

```
Box S(Element)
    Type Elist = List(Element)
    Type Stack = Prod(high: Int, data: Elist)

    Proc Init[S: Stack](n: Int)
        S := Stack(0, Elist(?-Element | n))
    end Init

    Proc Push[S: Stack](x: Element)
        if S.high<|S.data| →
            S.data.(S.high) := x
            S.high := S.high+1
        fi
    end Push
```

```
    Proc Pop[S: Stack, x: Element]
        if S.high>0 →
            S.high := S.high-1
            x := S.data.(S.high)
        fi
    end Pop

    Proc Empty[S: Stack] → (Bool)
        return S.high = 0
    end Empty
end S
```

Vi får med denne datastruktur følgende kompleksiteter

$$\begin{aligned} T[\![\mathbf{Init}[S](N)]\!] &\in O(N) \\ T[\![\mathbf{Push}[S](x)]\!] &\in O(1) \\ T[\![\mathbf{Pop}[S](x)]\!] &\in O(1) \\ T[\![\mathbf{Empty}[S]]\!] &\in O(1) \end{aligned}$$

2.2 Køer

En *kø* over en elementtype E er en datatype, hvis værdimængde er følger af E-værdier. Lad Q være en køvariabel, og lad x være af type E. Så har køen følgende udtryk

$$\begin{aligned} \mathbf{Init}[Q] \ \mathbf{sat} \ & Q' = () \\ \mathbf{Enter}[Q](x) \ \mathbf{sat} \ & Q = (e_0, e_1, \ldots, e_{n-1}) \rightarrow Q' = (e_0, e_1, \ldots, e_{n-1}, x) \\ \mathbf{Remove}[Q, x] \ \mathbf{sat} \ & (Q = (e_0, e_1, \ldots, e_{n-1})) \wedge (n > 0) \rightarrow \\ & \qquad (Q' = (e_1, \ldots, e_{n-1})) \wedge (x' = e_0) \\ \mathbf{Empty}[Q] \ \mathbf{sat} \ & (Q' = Q) \wedge (\mathrm{Empty}' = (Q = ())) \end{aligned}$$

En kø med fast øvre størrelse kan implementeres ved hjælp af en cyklisk liste, så alle operationer finder sted i optimal tid. En polybox, der implementerer en kø med plads til N elementer, ser ud som følger.

```
Box Q(Element)
    Type Elist = List(Element)
    Type Queue = Prod(low, high, size: Int, data: Elist)

    Proc Init[Q: Queue](n: Int)
        Q := Queue(0, 0, 0, Elist(?-Element | n))
    end Init

    Proc Enter[Q: Queue](x: Element)
        if Q.size<| Q.data | →
            Q.data.(Q.high) := x
            Q.high := (Q.high+1) mod | Q.data |
            Q.size := Q.size+1
        fi
    end Enter

    Proc Remove[Q: Queue, x: Element]
        if Q.size>0 →
            x := Q.data.(Q.low)
            Q.size := Q.size-1
            Q.low := (Q.low+1) mod | Q.data |
        fi
    end Remove

    Proc Empty[Q: Queue] → (Bool)
        return Q.size = 0
    end Empty
end Q
```

Vi får med denne datastruktur følgende kompleksiteter

$$
\begin{aligned}
T[\![\mathbf{Init}[Q](N)]\!] &\in O(N) \\
T[\![\mathbf{Enter}[Q](x)]\!] &\in O(1) \\
T[\![\mathbf{Leave}[Q, x]]\!] &\in O(1) \\
T[\![\mathbf{Empty}[Q]]\!] &\in O(1)
\end{aligned}
$$

2.3 Implementationsovervejelser

Hvis vi ikke ønsker at (eller ikke kan) angive den maksimale størrelse, så kan stakke og køer implementeres ved hjælp af pointere, så samtlige operationer får udførelsestider i $O(1)$. Vi viser implementationen af en stak

```
Box S(Element)
    Type Stack = Pointer(Node)
    Type Node = Prod(first: Element, next: Stack)

    Proc Init[S: Stack]
        S := nil
    end Init

    Proc Push[S: Stack](x: Element)
        S := Stack(Node(x, S))
    end Push

    Proc Pop[S: Stack, x: Element]
        if S ≠ nil →
            x := ref(S).first
            S := ref(S).next
        fi
    end Pop

    Proc Empty[S: Stack] → (Bool)
        return S = nil
    end Empty
end S
```

2.4 En anvendelse: beregning af postfix udtryk

Regneudtryk kan skrives på andre måder end med operatoren *imellem* argumenterne. I de såkaldte *postfix* udtryk skrives begge argumenterne *før* operatoren. Dette kendes blandt andet fra HP-lommeregnere, og kaldes også for *omvendt polsk notation*. Postfix udtryk har den fordel, at man aldrig får brug for parenteser. Fx vil regneudtrykket `(2+3)*7-4` i postfix

notation skrives som `2 3 + 7 * 4 -`. Følgende program benytter en stak af heltal til at beregne værdien angivet af et postfix udtryk; vi antager for simpelheds skyld, at talkonstanter kun har et enkelt ciffer.

```
(+ @"stack.tri"

   Box R
      S(Int)
   end R

   Var T: Text
   Var Z: R'Stack

   R'Init[Z]
   read[T]
   (+ Var i, r: Int
      i:=0
      do i<| T |→
            if '0' ≤ T.(i) ≤ '9' → R'Push[Z](ci(T.(i))-ci('0'))
            & true →
                  (+ Var x, y: Int
                     R'Pop[Z, y]
                     R'Pop[Z, x]
                     if T.(i) = '*' → R'Push[Z](x*y)
                      | T.(i) = '+' → R'Push[Z](x+y)
                      | T.(i) = '-' → R'Push[Z](x-y)
                      | T.(i) = '/' → R'Push[Z](x/y)
                     fi
                  +)
            fi
            i:=i+1
      od
      R'Pop[Z, r]
      write(r)
   +)
+)
```

3 Prioritetskøer

Vi er givet en elementtype E, hvis værdier har en total ordning $\sqsubseteq$. En *prioritetskø* over E er en datatype, hvis værdimængde består af endelige multimængder af værdier af type E.

En *multimængde* er ligesom en mængde, bortset fra at den kan indeholde gentagelser af elementer.

Lad P være en prioritetskøvariabel, og x af type E. Så skal prioritetskøen have følgende operationer

$$\begin{array}{rll}
\textbf{Init}[P] & \textbf{sat} & P' = \emptyset \\
\textbf{Empty}[P] & \textbf{sat} & (P' = P) \wedge (\text{Empty}' = (P = \emptyset)) \\
\textbf{Insert}[P](x) & \textbf{sat} & P' = P \cup \{x\} \\
\textbf{DeleteMin}[P, x] & \textbf{sat} & P \neq \emptyset \rightarrow \\
 & & \quad (x' \in P) \wedge (P' = P - \{x'\}) \wedge (\forall a \in P : x' \sqsubseteq a)
\end{array}$$

Vi kan tænke på dette som en "udemokratisk" kø, hvor ankomster er vilkårlige, men hvor man altid lader den "fineste" ventende forlade køen først.

Prioritetskøen omfatter flere andre datatyper. Man kan få en almindelig *stak* over E ved at knytte *tidsinformation* til elementerne og definere $e_1 \sqsubset e_2$, hvis e_1 ankom *efter* e_2. Omvendt kan man konstruere en *kø* ved på tilsvarende måde at definere $e_1 \sqsubset e_2$, hvis e_1 ankom *før* e_2.

Vi ved, at disse to specialtilfælde kan implementeres optimalt ved hjælp af lister. I det generelle tilfælde er det vanskeligere. Der er to indlysende måder at bruge en liste i den generelle situation. For det første kan man ved indsættelse blot anbringe det nye element bagest i listen og ved fjernelse lede efter det mindste element. Det fører til følgende kompleksiteter (N er igen det maksimale antal elementer).

$$\begin{array}{rcl}
T[\![\textbf{Init}[P](N)]\!] & \in & O(N) \\
T[\![\textbf{Empty}[P]]\!] & \in & O(1) \\
T[\![\textbf{Insert}[P](x)]\!] & \in & O(1) \\
T[\![\textbf{DeleteMin}[P, x]]\!] & \in & \Omega(|P|)
\end{array}$$

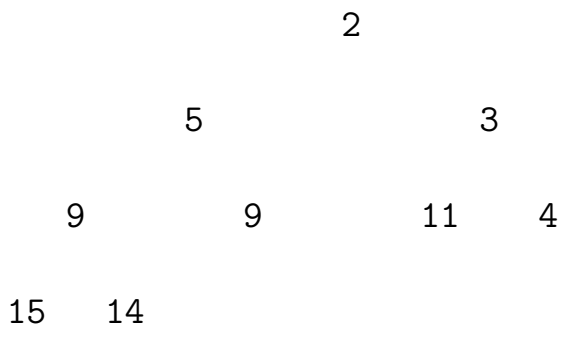
Omvendt kan man sørge for hele tiden at have det mindste element forrest i listen. Så bliver fjernelsen billig og indsættelsen tilsvarende dyr

$$\begin{aligned} T[\![\mathbf{Init}[P](N)]\!] &\in O(N) \\ T[\![\mathbf{Empty}[P]]\!] &\in O(1) \\ T[\![\mathbf{Insert}[P](x)]\!] &\in \Omega(|P|) \\ T[\![\mathbf{DeleteMin}[P,x]]\!] &\in O(1) \end{aligned}$$

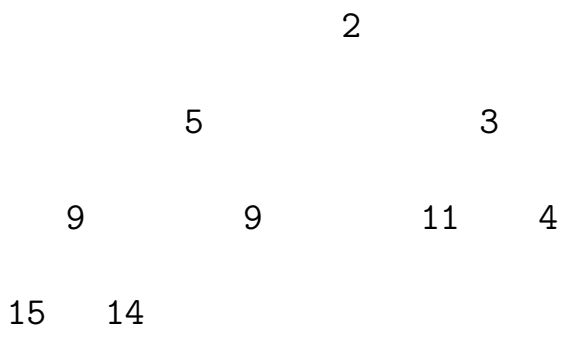
I mange situationer har man brug for en bedre balance mellem operationerne. Dette kan opnås ved hjælp af en træstruktur.

3.1 Bunker

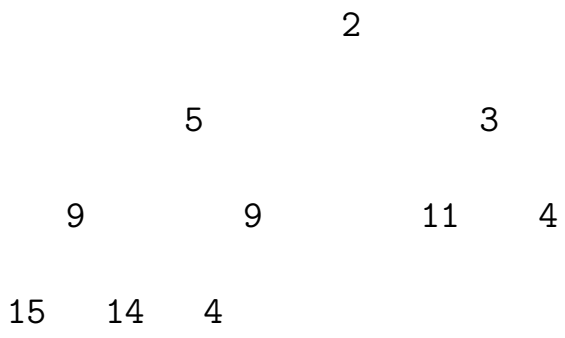
En *bunke* over E er et (balanceret) binært træ, hvis knuder indeholder værdier af type E. Endvidere gælder det for enhver knude, med indhold e_1, at hvis den har en efterfølger, med indhold e_2, så er $e_1 \sqsubseteq e_2$. Det vil sige, at elementerne på enhver vej fra roden er i ikke-aftagende orden. Det følger umiddelbart, at roden af bunken indeholder det mindste element. Det følgende er en bunke af heltal



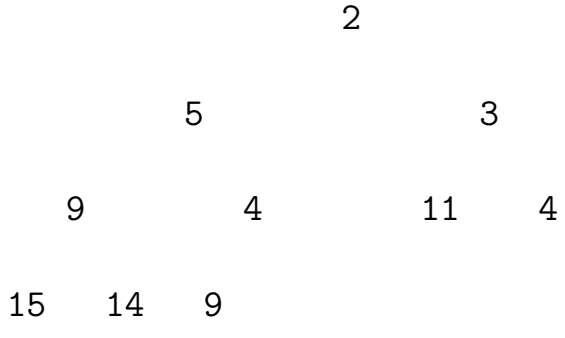
Lad os se hvordan man kan indsætte i en bunke. Fx kan vi indsætte tallet 4 i ovenstående eksempel. For det første skal træet have en ny knude. Dette kan klares ved at tilføje et blad, det første sted der "mangler" et



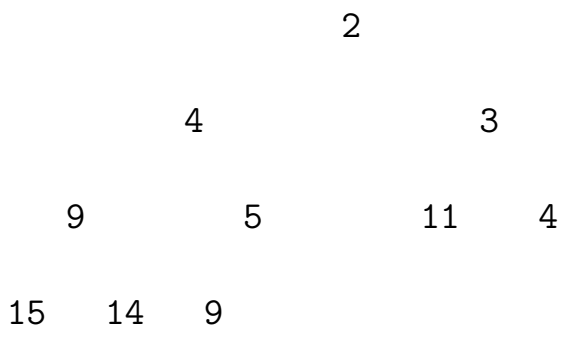
Desværre kan vi ikke blot indsætte det nye element her. Det kan jo, som i dette tilfælde, være for lille. Denne skavank er dog meget let at udbedre. Når vi har indsat det nye element, kan vi ombytte det med dets far, sålænge sønnen er mindre end faderen. På denne måde bliver det nye element "skubbet op" i bunken, indtil det falder på plads. Hvis det nye element er mindre end alle de gamle, vil det ende i roden af bunken. Indsættelsen af tallet 4 vil fortsætte i følgende skridt



Da 4 er mindre end 9 bytter vi rundt.



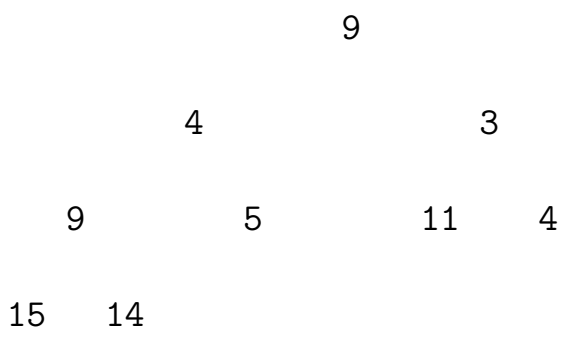
4-tallet er endnu ikke på plads, da dets far er et 5-tal.



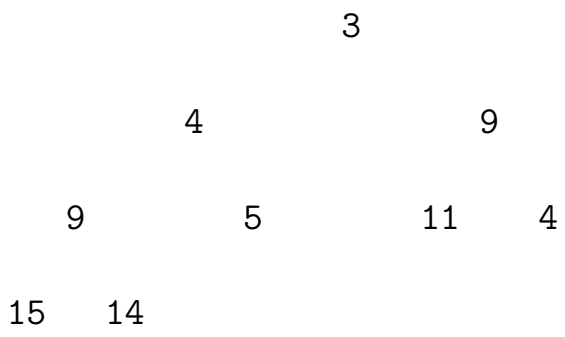
Til sidst har vi fået 4-tallet anbragt, hvor det hører hjemme. Bemærk, at vi aldrig kan komme til at skubbe et element længere end *højden* af bunken.

Fjernelsen af det mindste element foregår på tilsvarende facon. Som sagt, kan vi altid finde det mindste element i roden. Når vi har fjernet det, står vi desværre med et "hul", der skal fyldes ud

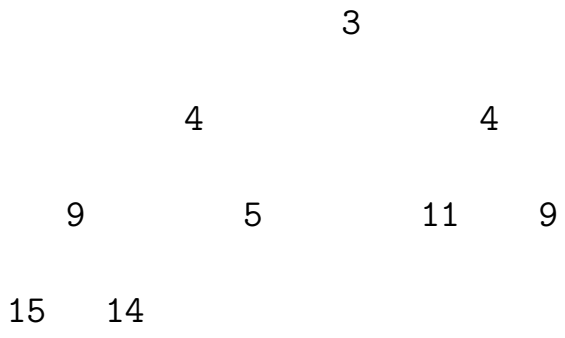
Vi fjerner bladet længst til højre og flytte dets indhold op i roden



Nu er situationen ligeså slem som før: det nye element i roden er for stort. Analogt med tidligere, kan vi dog "skubbe" 9-tallet ned i bunken, til det kommer på plads. Vi ombytter det med det *mindste* af dets sønner



Vi kan heller ikke have et 9-tal siddende her, så vi ombytter igen med den mindste søn



Nu er situationen normaliseret. Som før kan vi aldrig skubbe elementet længere end *højden* af bunken.

Vi kan observere, at vores anstrengelser fører til, at bunken altid er et *perfekt balanceret* binært træ. Det vil sige, at træet altid er helt fyldt ud, undtagen måske i det nederste lag, der til gengæld er udfyldt fra venstre. Træet har altså følgende udseende

Hvis et perfekt balanceret træ har n knuder, så er dets højde $O(\log n)$.

Antag nemlig, at træet har højde h; så må det have mindst

$$2^0 + 2^1 + \cdots + 2^{h-1} + 1 = 2^h$$

knuder (hvorfor?). Så hvis træet har n knuder, så er $2^h \leq n$, det vil sige $h \leq \log n$. Kompleksiteterne bliver således i dette tilfælde

$$\begin{aligned} T[\![\mathbf{Init}[P](N)]\!] &\in O(N) \\ T[\![\mathbf{Empty}[P]]\!] &\in O(1) \\ T[\![\mathbf{Insert}[P](x)]\!] &\in O(\log |P|) \\ T[\![\mathbf{DeleteMin}[P, x]]\!] &\in O(\log |P|) \end{aligned}$$

Da bunken er perfekt balanceret, kan man implementere den på en særligt snedig måde. Hvis

$$e_0, e_1, e_2, \ldots, e_{|P|-1}$$

er en *lag-for-lag* udskrift af en bunke, gælder det, at e_0 er roden, og hvis e_i er en vilkårlig knude, så er dens to sønner e_{2i+1} og e_{2i+2}. Vi kan derfor repræsentere bunken i en liste i den ovennævnte lag-for-lag rækkefølge. Den sidste version af vores eksempelbunke ovenfor ville blive repræsenteret som

$$(3, 4, 4, 9, 5, 11, 9, 15, 14)$$

Denne repræsentation er mere kompakt end en rekursiv træstruktur. Desuden kan det sidste element findes i konstant tid, hvilket ikke er tilfældet for træer.

En box med en prioritetskø af heltal ser med dette valg af datastruktur ud som følger

```
Type Element = Int

Box Pri
    Type Elist = List(Element)
    Type Queue = Prod(size: Int, data: Elist)

    Proc Init[Q: Queue](N: Int)
        Q := Queue(0, Elist(?-Element | N))
    end Init
```

```
Proc Empty[Q: Queue] → (Bool)
    return Q.size = 0
end Empty

Proc Insert[Q: Queue](x: Element)
    (+ Proc Up[Q: Queue](n: Int)
        (+ Var father: Int
           father := (n-1)/2
           if (n > 0) ∧ (Q.data.(father) > Q.data.(n)) →
                Q.data.(father) :=: Q.data.(n)
                Up[Q](father)
           fi
        +)
      end Up

      if Q.size < | Q.data | →
          Q.data.(Q.size) := x
          Q.size := Q.size+1
          Up[Q](Q.size-1)
      fi
    +)
end Insert

Proc DeleteMin[Q: Queue, x: Element]
    (+ Proc Down[Q: Queue](n: Int)
        (+ Var son: Int
           son := 2*n+1
           if son < Q.size →
                if (son < Q.size-1) ∧ (Q.data.(son+1) < Q.data.(son)) →
                     son := son+1
                fi
                if Q.data.(n) > Q.data.(son) →
                     Q.data.(n) :=: Q.data.(son)
                     Down[Q](son)
                fi
           fi
        +)
      end Down
```

```
            if Q.size > 0 →
                x := Q.data.(0)
                Q.size := Q.size-1
                if Q.size > 0 →
                    Q.data.(0) := Q.data.(Q.size)
                    Down[Q](0)
                fi
            fi
        +)
    end DeleteMin
end Pri
```

3.2 En polymorf prioritetskø

I dette afsnit beskrives en mere generelt anvendelig prioritetskø. Den implementeres som en *polybox*, således at elementerne kan have vilkårlige typer. Da man skal have en ordning mellem elementer, så kræver vi, at man sammen med et element angiver en *nøgle*, som er et heltal der angiver elementets prioritet. Til gengæld kan man ved initialisering, foruden køens maksimale størrelse, også angive hvilken vej ordningen på nøgler skal vende; DeleteMin er defor omdøbt til DeleteBest.

```
Box PolyPri(E)
    Type Element = Prod(val: E, key: Int)
    Type Elist = List(Element)
    Type Queue = Prod(size: Int, data: Elist, less: Bool)

    Proc Init[Q: Queue](N: Int, less: Bool)
        Q := Queue(0, Elist(?-Element | N), less)
    end Init

    Proc Order[Q: Queue](i, j: Int) → (Bool)
        if ¬Q.less → i :=: j fi
        return Q.data.(i).key < Q.data.(j).key
    end Order
```

```
Proc Empty[Q: Queue] → (Bool)
    return Q.size = 0
end Empty

Proc Insert[Q: Queue](val: E, key: Int)
    (+ Proc Up[Q: Queue](n: Int)
        (+ Var father: Int
           father := (n-1)/2
           if (n > 0) ∧ Order[Q](n, father) →
                Q.data.(father) :=: Q.data.(n)
                Up[Q](father)
           fi
        +)
      end Up

      if Q.size < | Q.data | →
          Q.data.(Q.size) := Element(val, key)
          Q.size := Q.size+1
          Up[Q](Q.size-1)
      fi
    +)
end Insert

Proc DeleteBest[Q: Queue, val: E, key: Int]
    (+ Proc Down[Q: Queue](n: Int)
        (+ Var son: Int
           son := 2*n+1
           if son < Q.size →
                if (son < Q.size-1) ∧ Order[Q](son+1, son) →
                     son := son+1
                fi
                if Order[Q](son, n) →
                     Q.data.(n) :=: Q.data.(son)
                     Down[Q](son)
                fi
           fi
        +)
      end Down
```

```
            if Q.size > 0 →
                val := Q.data.(0).val
                key := Q.data.(0).key
                Q.size := Q.size-1
                if Q.size > 0 →
                    Q.data.(0) := Q.data.(Q.size)
                    Down[Q](0)
                fi
            fi
        +)
    end DeleteBest
end PolyPri
```

Boxen Pri kan nu fås som

```
Box Pri
    PolyPri(Int)
end Pri
```

hvor man skal initialisere med

```
Pri'Init[P](N, true)
```

En prioritetskø uden maksimal størrelse kan som i de andre tilfælde implementeres ved hjælp af polyboxen Sequence.

3.3 En anvendelse: træsortering

En indlysende anvendelse af en sådan prioritetskø er til *sortering*. Man indsætter først alle tallene og fjerner dem derefter et ad gangen i størrelsesorden. Følgende program indlæser en vektor og udskriver den igen i sorteret orden.

```
Process Treesort
    (+ @"pri.tri"

        Var S: Vector
        Var H: Pri'Queue
        write("Indlæs vektor: ")
        read[S]
        Pri'Init[H](| S |)
        (+ Var i: Int
           i:=0
           do i<| S | →
                Pri'Insert[H](S.(i))
                i:=i+1
           od
        +)
        (+ Var i: Int
           i:=0
           do i<| S | →
                Pri'DeleteMin[H,S.(i)]
                i:=i+1
           od
        +)
        write(S)
    +)
end Treesort
```

Hvis vi indlæser n tal, så får vi

$$T[\![\text{Treesort}]\!](n) \approx \sum_{i=0}^{n-1} T[\![\textbf{Proc } \text{Insert}]\!](i) + \sum_{i=1}^{n} T[\![\textbf{Proc } \text{DeleteMin}]\!](i)$$

hvor $T[\![\textbf{Proc } \text{Insert}]\!](i)$ ($T[\![\textbf{Proc } \text{DeleteMin}]\!](i)$) betegner udførelsestiden for indsættelse (fjernelse) i (fra) en kø med i elementer. Heraf fås

$$T[\![\text{Treesort}]\!](n) \approx \sum_{i=1}^{n} O(\log i) = O(\sum_{i=1}^{n} \log i) \subseteq O(n \log n)$$

hvilket kan vises at være *optimalt* blandt sorteringsalgoritmer, der kun må sammenligne og ombytte elementerne (jfr. [Programmeringsteori]).

Den første del af denne algoritme, indsættelsen af de n tal i bunken, kan gøres mere effektivt. Man kan starte med at bygge et perfekt balanceret træ med tallene i tilfældig orden. De mindste undertræer, bladene, er jo allerede små bunker i sig selv. Hvergang man har gjort to undertræer til bunker, kan man bygge dem sammen til en større bunke, ved at skubbe deres fælles rod på plads i bunken. Dette kan gøres med følgende forbløffende simple procedure, som vi kan tilføje til boxen Pri:

```
Proc Build[Q: Queue](v: Vector)
    Q := Queue(| v |, v)
    (+ Var i: Int
       i := Q.size
       do i>0 →
            i := i-1
            Down[Q](i)
       od
    +)
end Build
```

Det skulle være klart, at udførelsestiden for Build er givet ved

$$T[\![\textbf{Proc Build}]\!](n) \approx \sum_{i=0}^{n-1} T[\![\textbf{Proc Down}]\!](n,i)$$

hvor n er antallet af elementer i Q og i er værdien af parameteren i kaldet Down[Q](i). For selve proceduren Down gælder der

$$T[\![\textbf{Proc Down}]\!](n,i) \approx \begin{cases} 1 & \text{hvis } i \geq \frac{n}{2} \\ 1 + T[\![\textbf{Proc Down}]\!](n, 2i+1) & \text{ellers} \end{cases}$$

Løsningen til denne ligning er

$$T[\![\textbf{Proc Down}]\!](n,i) \approx \log(\frac{n}{i+1})$$

hvorfor vi får

$$T[\![\textbf{Proc Build}]\!](n) \approx \sum_{i=1}^{n} \log(\frac{n}{i})$$

Denne sum kan "deles op i bidder" på følgende måde

$$\sum_{i=\frac{n}{2}}^{n} \log(\frac{n}{i}) + \sum_{i=\frac{n}{4}}^{\frac{n}{2}-1} \log(\frac{n}{i}) + \cdots + \sum_{i=2}^{3} \log(\frac{n}{i}) + \log(n)$$

$$
\begin{aligned}
&< \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots + 2 \cdot (\log(n) - 1) + 1 \cdot \log(n) \\
&= n(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots + \frac{\log(n)}{n}) \\
&< n \sum_{j=1}^{\infty} \frac{j}{2^j} = 2n
\end{aligned}
$$

Vi kan således opbygge en bunke med n elementer i tid $O(n)$. Sorteringen er dog selvfølgelig stadig i $O(n \log(n))$, fordi elementerne skal tages ud af bunken igen.

Et sorteringsprogram, der benytter Build, ser ud som følger

```
(+ @"pri.tri"

   Var S: Vector
   Var H: Pri'Queue
   write("Indlæs vektor: ")
   read[S]
   Pri'Build[H](S)
   (+ Var i: Int
      i:=0
      do i<| S |→
            Pri'DeleteMin[H,S.(i)]
            i:=i+1
      od
   +)
   write(S)
+)
```

4 Ordbøger

Vi er givet en elementtype E. En *ordbog* over E er en datatype, hvis værdimængde består af endelige mængder af værdier af type E. Lad D være en ordbogsvariabel og x en værdi af type E. Så skal ordbogen have følgende operationer

$$\begin{aligned}
\mathbf{Init}[D] \ \ \mathbf{sat} \ \ & D' = \emptyset \\
\mathbf{Insert}[D](x) \ \ \mathbf{sat} \ \ & D' = D \cup \{x\} \\
\mathbf{Member}[D](x) \ \ \mathbf{sat} \ \ & (D' = D) \wedge (\mathrm{Member}' = (x \in D)) \\
\mathbf{Delete}[D](x) \ \ \mathbf{sat} \ \ & D' = D - \{x\}
\end{aligned}$$

Den simpleste datastruktur for en ordbog ville igen være en liste, der indeholdt dens elementer. Det vil dog give følgende kompleksiteter

$$\begin{aligned}
T[\![\mathbf{Init}[D]]\!] &\in O(N) \\
T[\![\mathbf{Insert}[D](x)]\!] &\in O(1) \\
T[\![\mathbf{Member}[D](x)]\!] &\in \Omega(|D|) \\
T[\![\mathbf{Delete}[D](x)]\!] &\in \Omega(|D|)
\end{aligned}$$

Man kan under specielle omstændigheder klare sig langt bedre.

4.1 Bitvektorer

Hvis E er intervallet $0..N$ hvor N er et passende lille heltal, så kan man repræsentere en ordbog over E som en *bitvektor*, der er en tabel af sandhedsværdier. Man kunne bruge typen

$$\mathbf{Type}\ \mathrm{Dictionary} = \mathbf{List}(\mathrm{Bool})$$

Hvis D er en variabel af denne type, er mængden den repræsenterer

$$\{i \in 0..N \mid D.(i) = \mathrm{true}\}$$

Med denne datastruktur opnår vi naturligvis optimale kompleksiteter

$$\begin{aligned} T[\![\mathbf{Init}[D]]\!] &\in O(N) \\ T[\![\mathbf{Insert}[D](x)]\!] &\in O(1) \\ T[\![\mathbf{Member}[D](x)]\!] &\in O(1) \\ T[\![\mathbf{Delete}[D](x)]\!] &\in O(1) \end{aligned}$$

Vi skal jo blot inspicere og opdatere en enkelt indgang i bitvektoren i hvert af disse tilfælde.

En box med en ordbog, implementeret som en bitvektor, ser i dette tilfælde således ud

```
Type Element = Int

Box B
    Type Dictionary = List(Bool)

    Proc Init[D: Dictionary](N: Int)
        D := Dictionary(false | N)
    end Init

    Proc Insert[D: Dictionary](x: Element)
        if 0≤x<|D| → D.(x) := true fi
    end Insert

    Proc Member[D: Dictionary](x: Element) → (Bool)
        if 0≤x<|D| → return D.(x) fi
    end Member

    Proc Delete[D: Dictionary](x: Element)
        if 0≤x<|D| → D.(x) := false fi
    end Delete
end B
```

Desværre vil grundmængden E i realistiske sammenhænge sjældent være tilstrækkeligt lille til, at man kan bruge bitvektorer.

4.2 Hash-tabeller

Selv om E er for stor til bitvektorer, kan man opnå en lignende effekt, hvis selve de mængder, man ønsker at behandle, ikke er for store.

En *hash-funktion* på E er en afbildning

$$h : E \to 0..M$$

for et passende M, der angiver størrelsordenen af kardinaliteten af de mængder, vi ønsker at behandle. Hash-funktionen har til opgave at "sprede" E-værdierne over hele tabellen. Da $|E| > M$ kan flere elementer blive sendt til samme indgang i tabellen. Vi må samle disse op i en liste, så vi kan definere

Type Elist = **List**(Int)
Type Dictionary = **List**(Elist)

Hvis D er en variabel af denne type, så er mængden den repræsenterer

$$\{e \in E \mid \exists i, j : D.(i, j) = e\}$$

Hvis elementerne er heltal, så er en typisk hash-funktion

$$h(e) = e \textbf{ mod } M$$

Bemærk, at hvis $0 < p \leq M$ er divisor i M *og* divisor i alle $e \in E$, så vil vi kun kunne bruge M/p indgange i tabellen. Ved at vælge M som et *primtal* minimaliserer vi risikoen for dette. Det præcise valg af hash-funktionen afhænger helt af den aktuelle anvendelse.

Hvis $M = 17$, så vil en hash-tabel med $h(e) = e$ **mod** 17, der indeholder elementerne

$$\{0, 3, 9, 11, 12, 17, 24, 28, 31, 43, 55, 68, 71, 88, 101\}$$

have følgende udseende

0	→	(0,17,68)
1	→	()
2	→	()
3	→	(3,71,88)
4	→	(55)
5	→	()
6	→	()
7	→	(24)
8	→	()
9	→	(9,43)
10	→	()
11	→	(11,28)
12	→	(12)
13	→	()
14	→	(31)
15	→	()
16	→	(101)

En box, der implementerer denne hashtabel, ser ud som følger

```
Box H
    Type Dictionary = List(Elist)
    Type Elist = List(Int)

    Proc Hash(e: Int) → (Int)
        return e mod 17
    end Hash

    Proc Init[D: Dictionary]
        D := Dictionary(Elist() | 17)
    end Init
```

```
    Proc Insert[D: Dictionary](x: Int)
        if ¬ Member[D](x) →
            (+ Var h: Int
               h := Hash(x)
               D.(h) := D.(h)++Elist(x)
            +)
        fi
    end Insert

    Proc Member[D: Dictionary](x: Int) → (Bool)
        (+ Var h, i: Int
           h, i := Hash(x), 0
           do i <| D.(h) | →
                if x = D.(h, i) → return true fi
                i := i+1
           od
           return false
        +)
    end Member

    Proc Delete[D: Dictionary](x: Int)
        (+ Var h, i: Int
           h, i := Hash(x), 0
           do (i <| D.(h) |) ∧ (x ≠ D.(h, i)) → i := i+1 od
           if i <| D.(h) | →
                D.(h) := D.(h)(0 .. i)++D.(h)(i+1 .. | D.(h) |)
           fi
        +)
    end Delete
end H
```

I værste fald, hvor hash-funktionen sender alle elementer til samme værdi, vil dette degenerere til kompleksiteterne

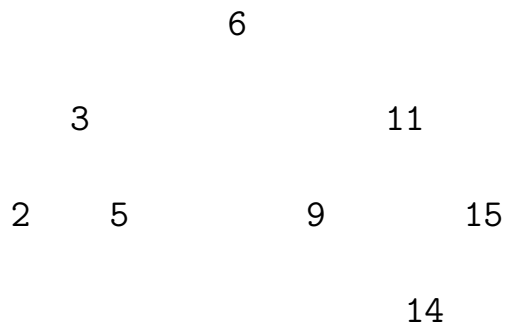
$$\begin{aligned} T[\![\mathbf{Init}[D]]\!] &\in O(M) \\ T[\![\mathbf{Insert}[D](x)]\!] &\in \Omega(|D|) \\ T[\![\mathbf{Member}[D](x)]\!] &\in \Omega(|D|) \\ T[\![\mathbf{Delete}[D](x)]\!] &\in \Omega(|D|) \end{aligned}$$

men med en rimelig fordeling af elementerne i tabellen, så vil alle de små lister kun have få elementer, og kompleksiteterne bliver

$$\begin{aligned} T[\![\mathbf{Init}[D]]\!] &\in O(M) \\ T[\![\mathbf{Insert}[D](x)]\!] &\in O(1) \\ T[\![\mathbf{Member}[D](x)]\!] &\in O(1) \\ T[\![\mathbf{Delete}[D](x)]\!] &\in O(1) \end{aligned}$$

4.3 Søgetræer

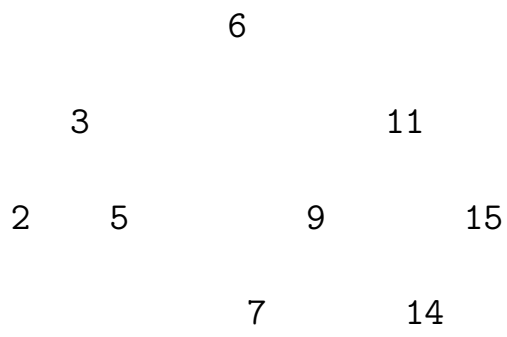
Hvis E har en total ordning $\sqsubseteq$, så kan vi igen få fordel af en træstruktur. Et *søgetræ* over E er et binært træ med E-værdier i knuderne. Hvis en knude, med indhold e, har en efterfølger i dens venstre undertræ, med indhold a, så skal det gælde, at $a \sqsubset e$; hvis knuden har en efterfølger i dens højre undertræ, med indhold b, så skal det gælde, at $e \sqsubset b$. Et søgetræ over heltallene kan se ud som følger



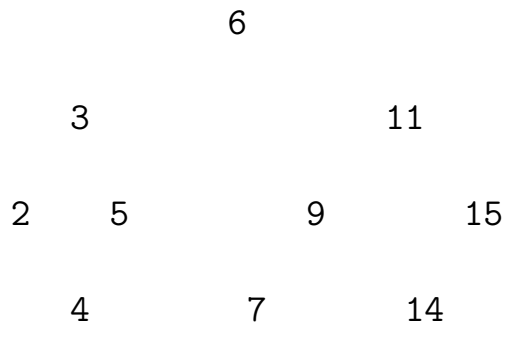
Det er meget let at finde et element i et søgetræ. Man sammenligner med værdien i den aktuelle knude. Hvis den er lig med den søgte værdi er man færdig. Hvis den er større end den søgte værdi skal man lede videre i venstre undertræ, og hvis den er mindre fortsættes søgningen i højre undertræ.

Indsættelse er ligeså simpel. Først leder man efter det aktuelle element, da det jo ikke skal indsættes to gange. Hvis man ikke finder det, er man til sidst i et blad, hvor det aktuelle element kan indsættes enten til højre eller til venstre.

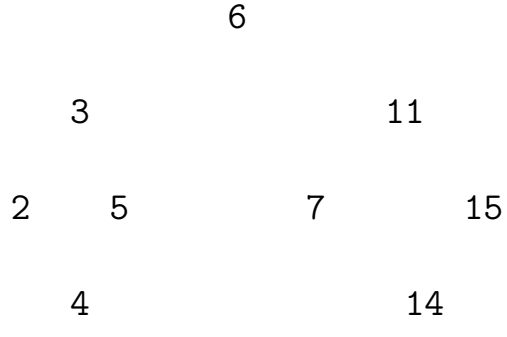
Hvis vi indsætter tallet 7 i ovenstående søgetræ får det udseendet



Indsættelse af et 4-tal resulterer i dette træ

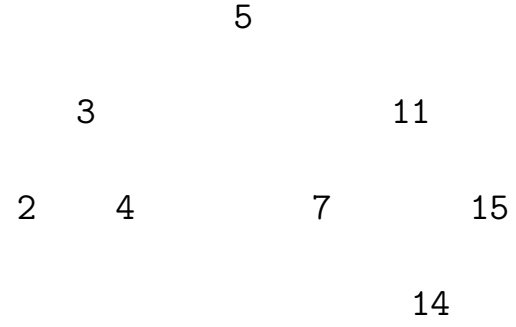


Man kan også fjerne et element e fra søgetræet. Hvis den uønskede knude højst har et enkelt undertræ er det ligetil. Så kan man blot erstatte knuden med roden i dens undertræ. I ovenstående eksempel kan vi fjerne tallet 9 og opnå træet



Hvis knuden har to undertræer er det knap så simpelt. Hvis man blot udtager knuden med elementet, så efterlader man et uklædeligt hul i træet. Ligesom ved bunken skal vi finde en passende erstatning. Det er heldigvis let at gøre. Søgetræets invariant bliver nemlig overholdt, hvis man erstatter elementet e med dets *umiddelbare forgænger*. Dette er det største element,

der er mindre end e, og det findes i den højreste knude i e's venstre undertræ. Denne knude kan ikke have noget højre undertræ og er således let at fjerne. Hvis vi fjerner tallet 6 fra eksempeltræet, observerer vi først, at dets umiddelbare forgænger er 5. Derfor bliver det nye træ som følger



Man kunne naturligvis også have valgt den *umiddelbare efterfølger*, hvilket er det mindste element, der er større.

Alle disse operationer tager tid proportionalt med træets højde. For et træ D med højde h fås derfor

$$\begin{aligned} T[\![\mathbf{Init}[D]]\!] &\in O(1) \\ T[\![\mathbf{Insert}[D](x)]\!](h) &\in O(h) \\ T[\![\mathbf{Member}[D](x)]\!](h) &\in O(h) \\ T[\![\mathbf{Delete}[D](x)]\!](h) &\in O(h) \end{aligned}$$

Man kan bevise, at hvis elementerne bliver indsat og fjernet i helt tilfældig orden, så er det rimeligt at antage, at h er proportional med $\log |D|$. Men hvis elementerne fx bliver indsat i *sorteret* orden, vil man få den værst tænkelige situation, hvor træet bliver *lineært*.

I dette triste tilfælde får vi disse deprimerende kompleksiteter

$$\begin{aligned} T[\![\mathbf{Init}[D]]\!] &\in O(1) \\ T[\![\mathbf{Insert}[D](x)]\!] &\in \Omega(|D|) \\ T[\![\mathbf{Member}[D](x)]\!] &\in \Omega(|D|) \\ T[\![\mathbf{Delete}[D](x)]\!] &\in \Omega(|D|) \end{aligned}$$

I næste afsnit skal vi betragte en variant af søgetræer, der *med garanti* forbliver pæne og balancerede.

En ordbog af heltal, implementeret som søgetræer, ser ud som følger

```
Type Element = Int

Box Search
    Type Tree = Prod (val: Element, left, right: Tree)

    Proc Init [T: Tree]
        T := ?-Tree
    end Init

    Proc Member [T: Tree] (x: Element) → (Bool)
        if ¬ is(T) → return false
        & true →
            if T.val = x → return true
            | T.val>x → return Member [T.left] (x)
            | T.val<x → return Member [T.right] (x)
            fi
        fi
    end Member

    Proc Insert [T: Tree] (x: Element)
        if ¬ is(T) → T := Tree(x, ?-Tree, ?-Tree)
        & true →
            if T.val>x → Insert [T.left] (x)
            | T.val<x → Insert [T.right] (x)
            fi
        fi
    end Insert

    Proc Delete [T: Tree] (x: Element)
        (+ Proc DeleteMax [T: Tree, m: Element]
            if ¬ is(T.right) →
                m := T.val
                T :=: T.left
            & true → DeleteMax [T.right, m]
            fi
          end DeleteMax
```

```
            if is(T) →
                if T.val = x →
                    if ¬ is(T.left) → T :=: T.right
                    | ¬ is(T.right) → T :=: T.left
                    & true → DeleteMax[T.left, T.val]
                    fi
                | T.val>x → Delete[T.left](x)
                | T.val<x → Delete[T.right](x)
                fi
            fi
        +)
    end Delete
end Search
```

Denne datatype kan naturligvis gøres polymorf i lighed med prioritetskøen.

4.4 Rød-sorte træer

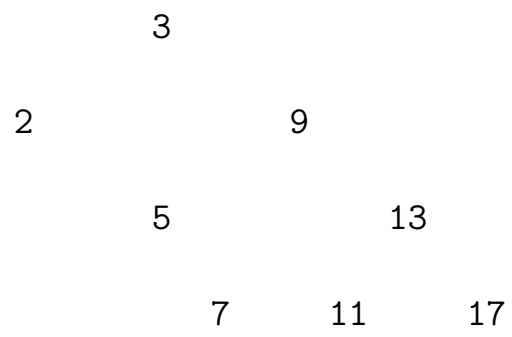
Problemet med ubalancerede søgetræer kan afhjælpes på mange måder. I alle tilfælde styrker man den invariant, som træerne skal opfylde. Udfra denne ekstra information skal man så kunne slutte, at træerne har logaritmisk højde.

Et *rødt-sort træ* er et søgetræ, hvori alle knuderne er farvet enten røde eller sorte. Roden kan kun være sort, og en rød knude kan aldrig have en rød søn. Herudover kræves det, at der er samme antal sorte knuder på alle veje fra roden til en knude med 0 efterfølgere (blade) eller 1 efterfølger.

Af typografiske årsager vil vi angive en rød knude som

og en sort knude som

Det følgende er et eksempel på et rødt-sort træ



Da mindst hveranden knude på en vej skal være sort er den længste vej til en knude med 0 eller 1 efterfølger højst dobbelt så lang som den korteste. Det følger deraf, at alle veje i træet har logaritmiske længder: Lad træet have n knuder, og lad antallet af knuder i den korteste og længste vej være henholdsvis k_{min} og k_{max}. Det følger, at

$$2^{k_{min}} \leq n + 1 \leq 2^{k_{max}}$$

og dermed

$$k_{min} \leq \log(n + 1) \leq k_{max}$$

Da $k_{max} \leq 2k_{min}$ får vi, at

$$k_{min} \leq \log(n + 1) \leq 2k_{min}$$

og til sidst

$$\frac{1}{2}\log(n + 1) \leq k_{min} \leq \log(n + 1)$$

så den korteste vej er logaritmisk. Da den længste vej højst er dobbelt så lang, er den (og dermed enhver anden vej) også logaritmisk.

Vores problem er nu at skrive operationerne, så de overholder den rød-sorte invariant. **Init** og **Member** kræver ingen ændringer; et rødt-sort træ har jo samme struktur som et almindeligt søgetræ. Problemerne ligger ved **Insert** og **Delete**; her kan de hidtidige implementationer føre til overtrædelser af den rød-sorte invariant.

4.5 Indsættelse i rød-sorte træer

Når vi har indsat en ny knude (som et nyt blad), har vi samtidigt forøget en vej i træet med en ekstra knude. Vi kan ikke umiddelbart farve denne knude. Hvis den bliver rød kunne det være, at den havde en rød far. Hvis den bliver sort risikerer vi, at få en sort knude for meget på en vej.

Vi kan imidlertid altid slippe afsted med enten at farve knuden eller at sende problemet højere op i træet. Det følgende er et transitionssystem, der for forskellige situationer viser, hvordan vi skal bære os ad. På hver side af transitionsrelationen $\triangleright$ er der et rødt-sort træ. Ideen er, at hvis vi kan genkende en venstreside som et undertræ, så kan vi erstatte det med højresiden. Den "besværlige" knude, der skal farves, betegnes som

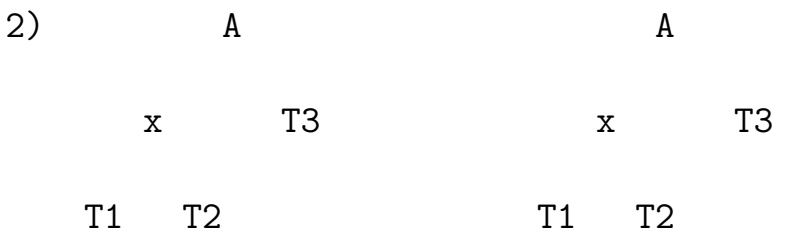
Algoritmen for indsættelse er nu, at vi først indsætter elementet som i et normalt søgetræ. Derefter mærker vi den nye knude som besværlig og anvender reglerne i følgende transitionssystem på træet. Hvis vi betegner den besværlige knude som en rød *uægte* knude (de øvrige knuder er *ægte*) vil følgende være en invariant for det "forfalskede" rød-sorte træ under udførelsen af transitionssystemet: (en ægte rod er sort) og (en ægte rød knude har en sort far) og (alle veje far roden til en knude med 0 eller 1 efterfølgere indeholder lige mange sorte knuder). Da vi altid gør fremskridt i transitionssystemet (fordi vi enten eliminerer den besværlige knude eller skubber den højere op i træet) og da mindst en af reglerne (eller de symmetriske tilfælde) altid kan anvendes, følger det at transitionssystemet er korrekt.

Det letteste tilfælde er hvis den besværlige knude er roden i træet.

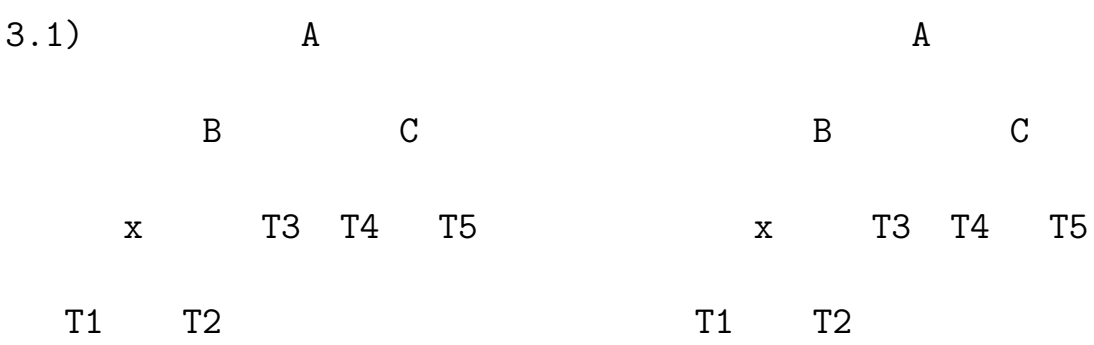
```
1)          x                          x

          T1    T2                 T1    T2
```

I så fald står det os frit at farve den sort og dermed forøge antallet af sorte knuder i *samtlige* veje.

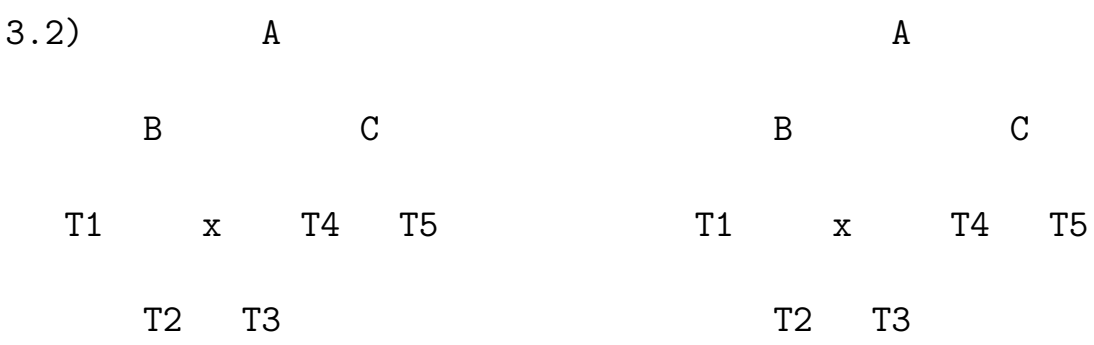
Hvis faderen er sort, kan vi uden videre farve knuden rød. Bemærk, at hvis `T1` og `T2` er tomme, svarer dette til situationen, hvor vi har indsat den nye knude som søn af en sort far.



Hvis faderen er rød, er der fire tilfælde. Hvis farbroderen eksisterer og er rød har vi situationerne 3.1 og 3.2.

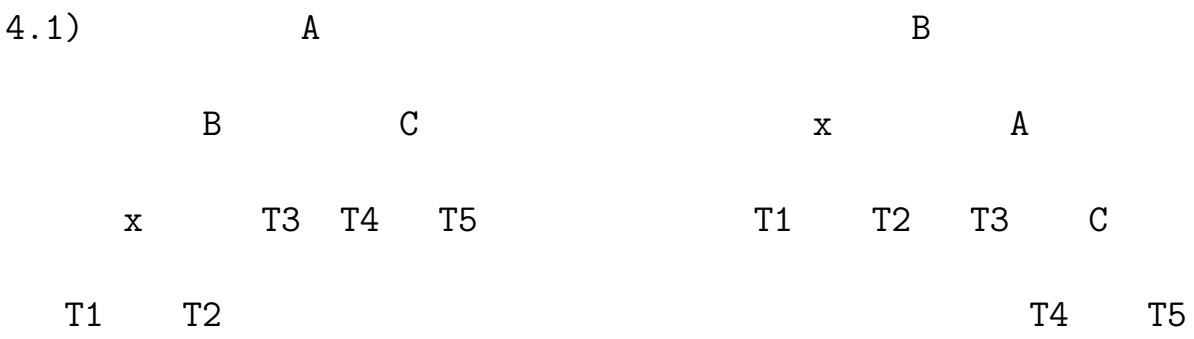


Da `B` og `C` skifter farve fra rød til sort, har vejene gennem dem en overskydende sort knude. Den kan vi slippe af med, mod til gengæld at gøre `A` til en "besværlig" knude.

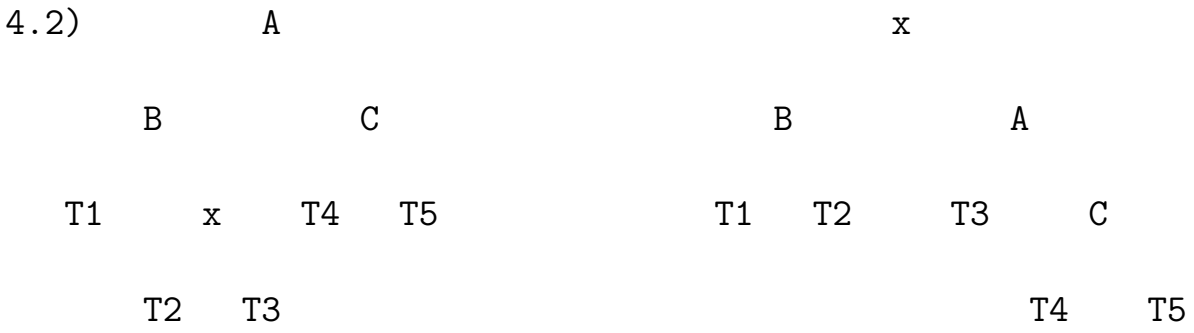


Her kan vi igen nøjes med at sende Rød-Sorteper videre.

Hvis farbroderen er sort eller ikke-eksisterende har vi tilfældende 4.1 og 4.2.



Bemærk, at træet bliver "roteret" omkring linjen `x-B-A`. En sådan rotation vil altid bevare den basale ordnings-invariant. Det er let at se, at også den rød-sorte invariant er bevaret.



Bemærk, at denne omfattende transformation lader ordnings-invarianten intakt. Her kan vi tillade os at farve `x` sort uden at spolere den rød-sorte invariant.

4.6 Fjernelse fra rød-sorte træer

Når vi fjerner en knude fra et søgetræ, så har den altid højst ét undertræ. Hvis den havde to undertræer, så ombytter vi den jo først med dens umiddelbare efterfølger (*eller* dens umiddelbare forgænger), der jo ikke kan have noget venstre (højre) undertræ. Når vi fjerner elementet i knuden, så står vi med et "hul", der skal fjernes. Hvis hullet er rødt, så kan vi umiddelbart fjerne det. Hvis hullet er sort, men dets eneste søn er rød, så kan vi retablere invarianten ved at fjerne hullet og farve sønnen sort. Hvis sønnen også

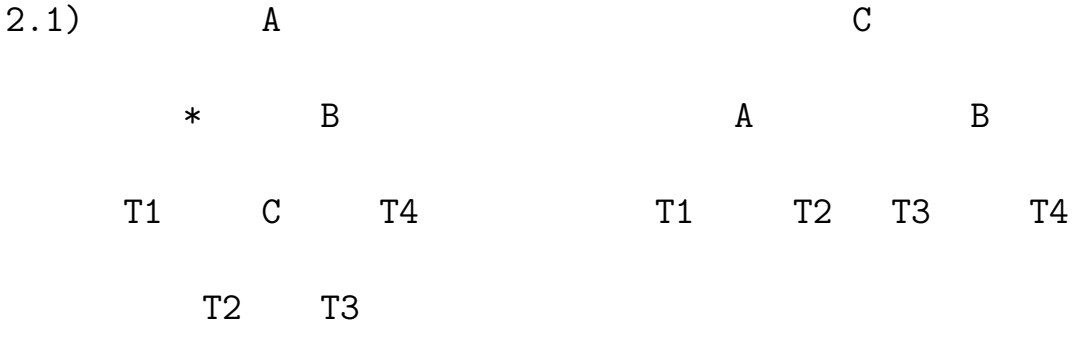
er sort, så må vi benytte os af et transitionssystem, ligesom vi gjorde ved indsættelsen. Vi angiver det som

og betegner det i analogi med ovenfor som en *uægte* sort knude (alle øvrige knuder er *ægte*). Vi eliminerer nu knuden ved hjælp af nedenstående transitionssystem. Invarianten er denne gang: (roden er sort) og (en rød knude har en sort far) og (alle veje fra roden til en ægte knude med 0 eller 1 efterfølgere indeholder lige mange sorte knuder). Da vi igen kan vise, at der altid gøres fremskridt og da der altid findes en regel der kan anvendes sålænge træet indeholder uægte knuder, er transitionssystemet igen korrekt.

Transitionssystemet ser ud som følger; igen gælder det, at hullets søn er enten sort eller ikke-eksisterende.

Hvis hullet er ved roden, er sagen ganske enkel.

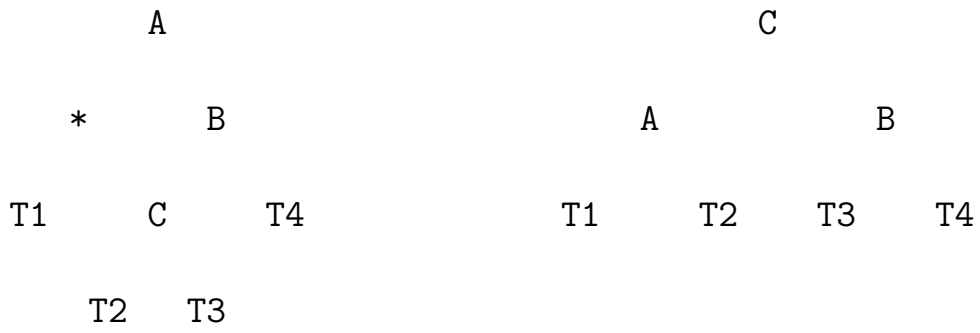
Hvis faderen er rød, er der to tilfælde. Nevøen eksisterer og er rød.



Her kan vi slippe af med hullet, ved at omarrangere træet. Det ses let, at invarianten respekteres.

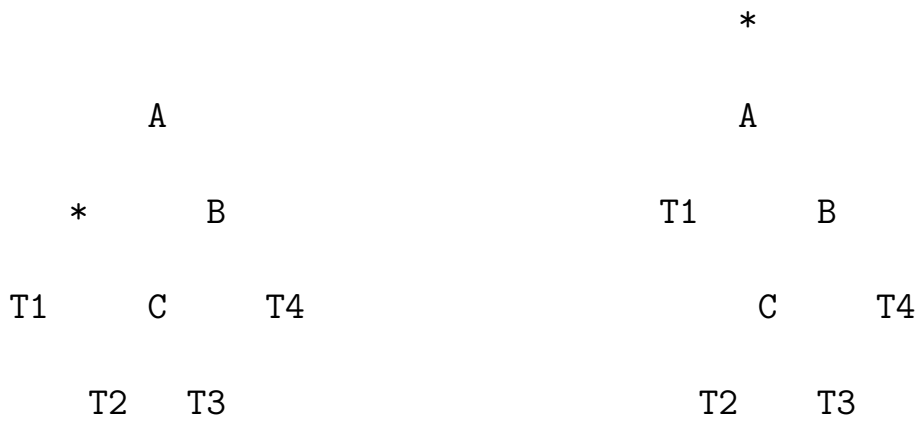
Nevøen er sort eller ikke-eksisterende.

Hvis faderen er sort, er der tre tilfælde. Broderen er sort og nevøen eksisterer og er rød.



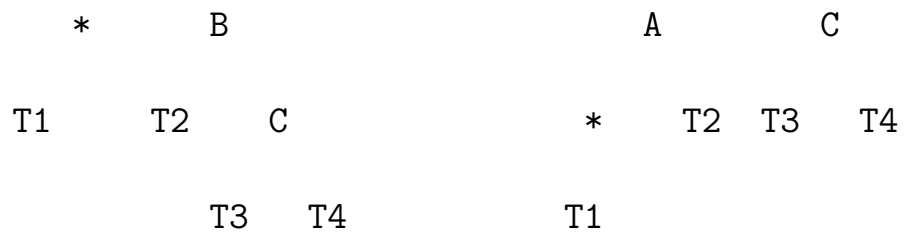
Her forsvinder hullet ved omarrangering.

Broderen er sort og nevøen er sort eller ikke-eksisterende.



Bemærk, at vi roligt kan farve `B` rød, da roden af `T4` må være sort. Ellers ville vi jo have det symmetriske tilfælde af `3.1)`.

Broderen er rød.

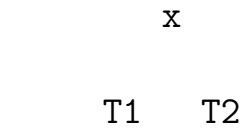


Denne transition ligner ikke nødvendigvis et fremskridt, men da roden af `T2` må være sort, kan vi gøre os færdige med enten regel `2.1` eller `2.2`.

Det overlades til den interesserede læser at realisere disse manipulationer i et TRINE-program.

4.7 Andre højdebalancerede træer

Rød-sorte træer er kun ét eksempel på højdebalancering. Der findes utallige andre. Et andet eksempel er følgende, hvor man kræver, at for ethvert undetræ af formen



skal det gælde, at

$$|\text{højde}(\texttt{T1}) - \text{højde}(\texttt{T2})| \leq K$$

for en konstant K. Hvis $K = 1$ benytter man også navnet *AVL-træer*. Ligesom med de rød-sorte træer kan man "rydde op" efter indsættelser og fjernelser og bevare invarianten.

Lad os her blot se, at denne invariant også fører til logaritmiske højder. Vi prøver at bygge et AVL-træ så magert, som vi kan. Definer M_h til at være et AVL-træ af højde h med så få knuder som overhovedet muligt. Det er let at se, at

$$|M_0| = 1 \text{ og } |M_1| = 2$$

I det generelle tilfælde er M_h et træ af højde h. Det ene undertræ må derfor være af højde $h - 1$. Vi tager selvfølgelig M_{h-1}. Hvor lille kan det andet

undertræ være? Invarianten siger, at det må have højde mindst $h-2$. Vi vælger derfor M_{h-2}. Men så kan vi slutte, at

$$|M_h| = |M_{h-1}| + |M_{h-2}| + 1$$

En simpel induktion viser, at $|M_h| \geq F_{h+2}$, hvor F_n er det n'te *Fibonacci-tal*, defineret som

$$\begin{aligned} F_1 &= 1 \\ F_2 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

Da det yderligere vides, at

$$F_n \in \Theta(\phi^n), \text{ hvor } \phi = \frac{\sqrt{5}+1}{2}$$

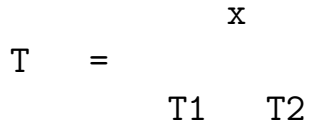
kan vi slutte, at det for ethvert AVL-træ T gælder, at

$$\text{højde}(T) \in O(\log_\phi |T|) = O(\log |T|)$$

så alle højder er logaritmiske.

4.8 Vægtbalancerede træer

I stedet for eksplicit at sætte begrænsninger på træernes højder, så kan man forlange, at *størrelserne* af en knudes undertræer ikke må være alt for forskellige. Et eksempel herpå er de såkaldte BB[α]-træer. Her står BB for Bounded Balance, og α er et reelt tal i intervallet $0 \leq \alpha \leq \frac{1}{2}$. Et binært træ tilhører BB[α] såfremt der for enhver knude gælder, at forholdet mellem antallet af knuder i dens undertræer er begrænset af α. Mere præcist forlanges det for et træ af formen



at

i) $|T| \leq 1$, eller

ii) $\alpha \leq \dfrac{|T_1|+1}{|T|+1} \leq 1-\alpha$ og både T_1 og T_2 tilhører BB[α].

Det er nemt at indse, at denne definition er symmetrisk imellem de to undertræer, fordi det af ii) følger, at der også gælder

$$\alpha \leq \frac{|T_2|+1}{|T|+1} \leq 1-\alpha$$

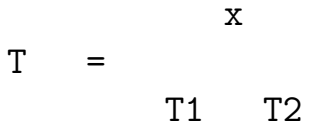
At BB[α]-træer også har logaritmiske højder følger af, at der for et vilkårligt BB[α]-træ T gælder

$$(*) \qquad \text{højde}(T) \leq \frac{\log(|T|+1)-1}{\log(1/(1-\alpha))}$$

Dette vises ved induktion efter $|T|$ på følgende måde.

Basis $|T| = 1$: Når T kun indeholder en enkelt knude er højde$(T) = 0$. Men da log betyder 2-talslogaritmen er $\log(1+1)-1$ også lig med 0.

Induktionsskridt $|T| = n+1$: T har formen



hvor $|T_1| \leq n$ og $|T_2| \leq n$, dvs $(*)$ kan antages at gælde for T_1 og T_2. T's højde er givet ved

$$\begin{aligned} \text{højde}(T) &= 1 + \max\{\text{højde}(T_1), \text{højde}(T_2)\} \\ &\leq 1 + \max\{\frac{\log(|T_1|+1)-1}{\log(1/(1-\alpha))}, \frac{\log(|T_2|+1)-1}{\log(1/(1-\alpha))}\} \end{aligned}$$

på grund af induktionsantagelsen. Da såvel T_1 som T_2 er BB[α]-træer, og mindst et af dem er ikke-tomt, følger det at

$$\max\{\frac{\log(|T_1|+1)-1}{\log(1/(1-\alpha))}, \frac{\log(|T_2|+1)-1}{\log(1/(1-\alpha))}\} \leq \frac{\log((1-\alpha)(|T|+1))-1}{\log(1/(1-\alpha))}$$

hvorfor

$$\begin{aligned}\text{højde}(T) &\leq 1 + \frac{\log((1-\alpha)(|T|+1)) - 1}{\log(1/(1-\alpha))} \\ &= 1 + \frac{\log(1-\alpha) + \log(|T|+1) - 1}{\log(1/(1-\alpha))} \\ &= \frac{\log(|T|+1) - 1}{\log(1/(1-\alpha))}\end{aligned}$$

hvilket skulle vises.

Vi skal ikke komme yderligere ind på BB[α]-træers egenskaber, men blot bemærke, at det, i lighed med højdebalancerede træer, er muligt at opretholde BB[α] egenskaben under indsættelse og og fjernelse med en omkostning der er proportional med træets højde. Dette sker for såvel BB[α]-træer som for de højdebalancerede træer ved hjælp af *rotationer* og *dobbeltrotationer*, som er omorganiseringer af træet svarende til reglerne 4.1 og 4.2 for rød-sorte træer.

4.9 Tider for balancerede søgetræer

Det følger af ovenstående, at hvis en ordbog implementeres ved hjælp af balancerede søgetræer, så fås følgende kompleksiteter

$$\begin{aligned} T[\![\mathbf{Init}[D]]\!] &\in O(1) \\ T[\![\mathbf{Insert}[D](x)]\!] &\in O(\log|D|) \\ T[\![\mathbf{Member}[D](x)]\!] &\in O(\log|D|) \\ T[\![\mathbf{Delete}[D](x)]\!] &\in O(\log|D|) \end{aligned}$$

Læseren vil også have opdaget, at vi kan benytte (balancerede) søgetræer til en effektiv implementation af *prioritetskøer*. Som nævnt kan de ekstreme værdier findes som de yderligste knuder i søgetræet. Det leder til en version af prioritetskøer, hvor man kan fjerne både det største og mindste element med kompleksiteter

$$\begin{aligned} T[\![\mathbf{Init}[P]]\!] &\in O(1) \\ T[\![\mathbf{Insert}[P](x)]\!] &\in O(\log|P|) \\ T[\![\mathbf{DeleteMin}[P,x]]\!] &\in O(\log|P|) \\ T[\![\mathbf{DeleteMax}[P,x]]\!] &\in O(\log|P|) \end{aligned}$$

Forskellige udvidelser af søgetræer kan realisere mange andre operationer i logaritmisk tid.

5 Ækvivalensrelationer

Lad E være en type med endelig (og lille) værdimængde. En *ækvivalensrelation* over E er en datatype, hvis værdimængde består af klassedelinger af E's værdimængde.

En *klassedeling* af en mængde S er en samling af *klasser* $\{S_i\}$, således at

- $\forall i : (S_i \subseteq S) \wedge (\ S_i \neq \emptyset)$
- $\bigcup_i S_i = S$
- $i \neq j \Rightarrow S_i \cap S_j = \emptyset$

En klassedeling repræsenterer en matematisk ækvivalensrelation, hvor to elementer er relaterede, hvis og kun hvis de tilhører den samme klasse.

Denne datastruktur finder som regel anvendelse, når elementerne kan nummereres på en enkel måde. Vi vil derfor fremover antage, at

$$E = \{0, 1, 2, \ldots, N-1\}$$

Lad R være en variabel af type ækvivalensrelation og lad x_1,x_2 være værdier af type E. Der skal være følgende udtryk

$$\begin{array}{rll} \textbf{Init}[R] & \textbf{sat} & R' = \{\{0\}, \{1\}, \ldots, \{N-1\}\} \\ \textbf{Equiv}[R](x_1, x_2) & \textbf{sat} & (R' = R) \wedge (\text{Equiv}' = (\exists K \in R : x_1, x_2 \in K)) \\ \textbf{Join}[R](x_1, x_2) & \textbf{sat} & R' = R \setminus \{K_1, K_2\} \cup \{K_1 \cup K_2\} \\ & & \text{hvor } x_1 \in K_1 \in R \text{ og } x_2 \in K_2 \in R \end{array}$$

Man starter med den diskrete ækvivalensrelation, som man kan gøre grovere og grovere. Undervejs kan man spørge, om to elementer er relaterede.

5.1 Kanoniske repræsentanter

En simpel implementation får man, hvis man indfører *kanoniske repræsentanter* for klasserne, i.e. et specielt element fra hver klasse, der repræsenterer denne.

En klassedeling over E kan nu repræsenteres ved hjælp af en afbildning, $\rho : E \to E$, hvor $\rho(e)$ er den kanoniske repræsentant for den klasse som e tilhører. Vi kan repræsentere ρ ved hjælp af typen

$$\textbf{Type}\ \mathrm{EqRel} = \textbf{List}(\mathrm{Int})$$

Hvis P er en variabel af denne type, der repræsenterer en ækvivalensrelation over E, så er $|P| = N$ og den kanoniske repræsentant for $e \in E$ er $P.(e)$. $\textbf{Init}[P]$ skal sætte P til identitetsafbildningen, $\textbf{Equiv}[P](x_1, x_2)$ skal teste om $P.(x_1) = P.(x_2)$, og $\textbf{Join}[P](x_1, x_2)$ skal ændre alle elementer med værdi $P.(x_1)$ til at have værdi $P.(x_2)$ (eller omvendt). Vi får følgende kompleksiteter

$$\begin{aligned} T[\![\textbf{Init}[P]]\!](N) &\in \Omega(N) \\ T[\![\textbf{Equiv}[P](x_1, x_2)]\!](N) &\in O(1) \\ T[\![\textbf{Join}[P](x_1, x_2)]\!](N) &\in \Omega(N) \end{aligned}$$

Som sædvanligt kommer træerne os til undsætning ved at tillade en mere effektiv implementation.

5.2 Inverse træer

Et *inverst træ (I-træ)* er enten *tomt*, eller det er en *knude*, der bliver peget på af et antal inverse træer

$$I_1 \quad I_2 \quad \cdots \quad I_k$$

Vi kan repræsentere en klassedeling af E som en skov af I-træer med E-værdier i knuderne. Hvert I-træ svarer til en klasse, der består af elementerne i knuderne.

Ideen er, at man udnævner værdien i roden til kanonisk element for den tilsvarende klasse. Hvis vi kan holde I-træerne rimeligt balancerede, kan logaritmiske udførelsestider skimtes i horisonten.

Det er let at udføre **Join**. Vi skal blot lade roden i det ene I-træ pege på roden af det andet. En indlysende optimalisering er at lade roden af det *letteste* I-træ pege på roden af det *tungeste*, hvor vi med *vægt* tænker på antallet af knuder. Vi må således holde rede på antallet af knuder i hvert I-træ.

Vi får brug for følgende observation: Et I-træ af højde h, opbygget ved hjælp af **Join**-operationer, indeholder mindst 2^h knuder.

Dette kan vises ved induktion i højden. For $h = 0$ er antallet af knuder lig med $1 = 2^0$. Antag nu, at egenskaben gælder for I-træer med højde mindre end h. Betragt et generelt I-træ af højde h. Det ser ud som følger

$$I_1 \quad I_2 \quad \cdots \quad I_k$$

Da I-træet har højde h, så må mindst ét undertræ have højde $h-1$. Antag, uden tab af generalitet, at I_j'erne er ordnet efter den rækkefølge, som de er blevet **Join**'et på, og at I_m har højde $h-1$. Kald I-træet

$$I_1 \quad \cdots \quad I_{m-1}$$

for I'. Så har vi **Join**'et

$$I'$$

$$I_m$$

hvor I' så har flere knuder end I_m. Induktionsantagelsen giver os, at I_m indeholder mindst 2^{h-1} knuder, og da I' har mindst lige så mange, har vi ialt mindst

$$2^{h-1} + 2^{h-1} = 2^h$$

knuder. Vores oprindelige I-træ har alle disse knuder, plus hvad der blev **Join**'et til yderligere, nemlig $I_{m+1}, \ldots, I_k$.

Det er nu let at se, at alle I-træerne i repræsentationen af en klassedeling har logaritmiske højder. Antag, at vi har N forskellige elementer, og at et af I-træerne har højde $h > \log(N)$. Så må dette I-træ indeholde mindst

$$2^{\log(N)+1} = 2(2^{\log(N)}) = 2N$$

knuder, hvilket er dobbelt så mange som vi har. Derfor er højden af ethvert I-træ logaritmisk. Vi får således følgende kompleksiteter

$$\begin{aligned} T[\![\mathbf{Init}[P]]\!](N) &\in \Theta(N) \\ T[\![\mathbf{Equiv}[P](x_1, x_2)]\!](N) &\in O(\log N) \\ T[\![\mathbf{Join}[P](x_1, x_2)]\!](N) &\in O(\log N) \end{aligned}$$

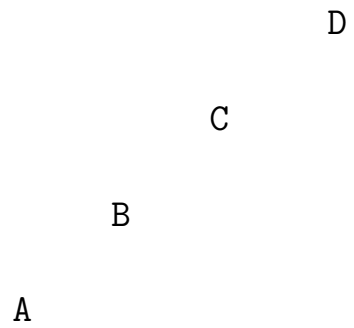
En box, der implementerer ækvivalensrelationer af heltal, ser ud som følger

```
Box Eq
    Type Rel = List(Data)
    Type Data = Prod(father: Int, weight: Int)

    Proc Init[R: Rel](n: Int)
        R := Rel(Data(0, 0) | n)
        (+ Var i: Int
           i := 0
           do i < n → R.(i), i := Data(i, 1), i+1 od
        +)
    end Init

    Proc Root[R: Rel](e: Int) → (Int)
        do R.(e).father ≠ e → e := R.(e).father od
        return e
    end Root
```

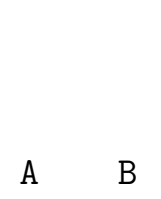
```
    Proc Equiv[R: Rel](x1, x2: Int) → (Bool)
        return Root[R](x1) = Root[R](x2)
    end Equiv

    Proc Join[R: Rel](x1, x2: Int)
        if ¬Equiv[R](x1,x2) →
            (+ Var r1, r2: Int
               r1, r2 := Root[R](x1), Root[R](x2)
               if R.(r1).weight > R.(r2).weight → r1 :=: r2 fi
               R.(r1).father := r2
               R.(r2).weight := R.(r1).weight+R.(r2).weight
            +)
        fi
    end Join
end Eq
```

5.3 Vejforkortning

Man kan foretage en yderligere optimalisering af denne implementation. Når man har fundet roden for en knude, kan man uden yderligere omkostning lade alle knuderne på vejen pege direkte til roden. Det vil jo forkorte vejen ved senere rodsøgninger. Man transformerer altså et træ af formen



til et af formen



Man kan (med noget besvær) vise, at en vilkårlig sekvens af **Join**- og **Equiv**-operationer af længde m nu kan udføres i tid

$$O(m \log^*(N))$$

hvor vi definerer

$$\begin{aligned}
&\log^*(x) = \min\{i \mid \log^i(x) \leq 1\} \\
&\log^0(x) = x \\
&\log^i(x) = \log(\log^{i-1}(x))
\end{aligned}$$

Intuitivt er $\log^*(x)$ det antal gange man skal tage logaritmen for at komme fra x til 1. Dette er bedre end kompleksiteten

$$O(m \log(N))$$

som vi uden vejforkortning ville opnå for sekvensen af operationer. Faktisk er $\log^*(x)$ så langsomt voksende, at den ikke overstiger 5, selv om x er langt større end antallet af elementarpartikler i universet.

Vi kan implementere vejforkortningen ved blot at ændre rodsøgningen således

```
Proc Root[R: Rel](e: Int) → (Int)
    if R.(e).father ≠ e →
        R.(e).father := Root[R](R.(e).father)
    fi
    return R.(e).father
end Root
```

Vejen, der skal forkortes, ligger implicit gemt på rekursionsstakken. Bemærk, at proceduren repræsenterer et eksempel på en værdiprocedure med en hensigtsmæssig sideeffekt.