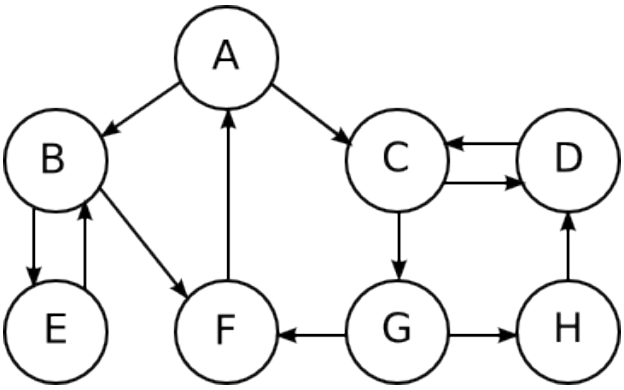
Lösningsförslag till SI-möte #14
Programkonstruktion och datastrukturer

Elias Castegren
elca7381@student.uu.se



i) I vilken ordning besöks noderna i en Breadth-First-Search (BFS)?
Om man till exempel börjar på nod A och väljer vägar motsols får man ordningen $\langle A, B, C, E, F, G, D, H\rangle$
Se föreläsningsslides för metod

ii) I vilken ordning besöks noderna i en Depth-First-Search (DFS)?
Om man börjar på nod A och väljer vägar motsols besöker man noderna i ordningen $\langle A, B, E, F, C, G, H, D\rangle$
(notera att "finishing order" är $\langle E, F, B, D, H, G, C, A\rangle$ *)*
Se föreläsningsslides för metod

iii) Är grafen ett träd?
Nej, grafen innehåller cykler

iv) Är grafen en komponent?
Ja, man kan nå alla noder från vilken nod som helst

v) Finns det någon väg som besöker alla noder exakt en gång?
Ja, t.ex. $\langle E, B, F, A, C, G, H, D\rangle$

Komplexitet

i) $f(n) = \Theta(n) \Rightarrow f(n) = O(n^2)$	**T**	*ii)* $f(x) = \Omega(x) \Rightarrow f(x) = O(x^2)$	**F**
iii) $T(n) = O(\log n) \Rightarrow T(n) = O(n \log n)$	**T**	*iv)* $p(x) = \Theta(q(x)) \Rightarrow q(x) = \Theta(p(x))$	**T**
v) $f(t) = O(g(t)) \Rightarrow g(t) = \Omega(f(t))$	**T**	*vi)* $A(u) = \Theta(B(u)) \Rightarrow A(u) = O(B(u))$	**T**
vii) $k(t) = \Theta(y(t)) \Rightarrow y(t) = \Omega(k(t))$	**T**	*viii)* $g(n) = O(f(n)) \Rightarrow f(n) = \Theta(g(n))$	**F**

Komplexitetsanalys

```
fun foo([]) = 0
  | foo(f::r) = 1 + foo(r);
```

Rekursionen för körtiden (där n är antalet element i argumentlistan) är

$$T_{foo}(n) = \begin{cases} k_0 & \text{om } n = 0 \\ T_{foo}(n-1) + k_1 & \text{om } n > 0 \end{cases}$$

En sluten formel för rekursionen är

$$T_{foo}(n) = nk_1 + k_0 = \Theta(n)$$

vilken enkelt bevisas med induktion. *Se lösningsförslagen till SI-möte #8 för ett mer utförligt bevis*

```
fun bar([]) = 2
  | bar(f::r) = foo(r) + bar(r);
```

Rekursionen för körtiden blir

$$T_{bar}(n) = \begin{cases} c_0 & \text{om } n = 0 \\ T_{foo}(n) + T_{bar}(n-1) + c_1 & \text{om } n > 0 \end{cases}$$

Informellt kan man säga att $T_{foo}(n) = \Theta(n)$ och därefter hitta den slutna formeln på samma sätt som för den förra uppgiften:

$$T(n) = n \cdot \Theta(n) + nc_1 + c_0 = \Theta(n^2)$$

Vill man vara mer noggrann får man använda den exakta sluta formeln $T_{foo}(n) = nk_1 + k_0$, och då blir den slutna formeln för T_{bar}

$$T_{bar}(n) = \left(\frac{n^2 + n}{2}\right) \cdot k_1 + n(k_0 + c_1) + c_0 = \Theta(n^2)$$

vilket också kan bevisas med induktion.

```
fun baz(0) = 42
 | baz(n) = baz(n div 2) + baz(n div 2);
```

Rekursionen för körtiden är

$$T_{baz}(n) = \begin{cases} k_0 & \text{om } n = 0 \\ 2T_{baz}(\frac{n}{2}) + k_1 & \text{om } n > 0 \end{cases}$$

Applicering av Master theorem ger att $n^{\log_2(2)} = n^1 = n$, och eftersom n dominerar k_1 (det finns ett ϵ så att $\frac{n}{k_1} = \Omega(n^\epsilon)$) hamnar vi i fall ett. Tidskomplexiteten blir alltså

$$T_{baz}(n) = \Theta(n^{\log_2(2)}) = \Theta(n)$$

Sortering

Se föreläsningsslides

Imperativ programmering

i) Skriv en funktion som skriver ut alla element i en sträng-lista i omvänd ordning (alltså sista elementet först) till terminalen.

```
(* printListRev l
   TYPE: string list -> ()
   PRE: ()
   POST: ()
   SIDE-EFFECTS: elementen i l skrivs ut i omvänd ordning
   EXAMPLES: printListRev [" dig!", " på", "hej"] = (),
             "hej på dig!" skrivs till terminalen
*)
(* VARIANT: length l *)
fun printListRev([]) = ()
  | printListRev(f::r) = (printListRev(r);print f);
```

ii) Skriv en funktion `removeTags(is)` som givet en inström `is` returnerar en sträng med alla tecken från `is` (till filslut) men utan de tecken som står mellan "<" och ">". Om `is` pekar på en fil med innehållet "hej <hopp> på dig!", ska alltså strängen "hej på dig!" returneras.

`input1(is):instream -> char option` returnerar nästa tecken från `is` taggat med `SOME`, och `NONE` vid filslut.

```
(* removeTags'(is, inTag)
   TYPE: instream * bool -> string
   PRE: ()
   POST: alla tecken från is efter första förekomsten av
         ">" utom text mellan "<" och ">" om inTag är true,
         annars alla tecken från is utom text mellan "<" och ">".
   SIDE-EFFECTS: läspekaren för is flyttas fram.
   EXAMPLES: removeTags'(is, true) = " hejsan", om is innehåller
             texten "hopp> hejsan".
*)
(* VARIANT: Antalet tecken i is innan filslut *)
fun removeTags'(is, inTag) =
    let
        val c = TextIO.input1(is)
    in
        if c = NONE then
            ""
        else if not inTag andalso c = SOME #"<" then
            removeTags'(is, true)
        else if inTag andalso c = SOME #">" then
            removeTags'(is, false)
        else
            str(valOf(c))^removeTags'(is, false)
    end;

(* removeTags(is)
   TYPE: instream -> string
   PRE: ()
   POST: alla tecken från is utom text mellan "<" och ">".
   SIDE-EFFECTS: läspekaren för is flyttas fram.
   EXAMPLES: removeTags(is) = "hej hejsan", om is innehåller
             texten "hej<hopp> hejsan"
*)
fun removeTags(is) = removeTags'(is, false)
```

Datastrukturer

Tabellen nedan visar tidskomplexiteten i genomsnittliga och värsta fall. I *åtkomst*, *insättning* och *borttagning* antar man att index (eller nyckeln) är känd. *Extrahera min* betyder att man plockar ut elementet med det minsta värdet på sitt data. En viktig operation som inte finns med i tabellen är sökning, alltså att hitta ett element med ett visst data (utan att känna till dess nyckel). Där presterar sökträd i allmänhet sämre (oftast linjär tid) medan hashtabeller och sorterade arrayer är bättre (konstant respektive logaritmisk komplexitet i genomsnittliga fall).

	Åtkomst		Insättning		Borttagning		Extrahera min	
(enkellänkad) Lista	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Sorterad lista	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Array*	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Sorterad array	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Binärt sökträd	$O(\lg n)$	$O(n)$	$O(\lg n)$	$O(n)$	$O(\lg n)$	$O(n)$	$O(n)$	$O(n)$
AVL-träd	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$	$O(n)$	$O(n)$
Binomial min heap	$O(n)$	$O(n)$	$O(\lg n)$	$O(\lg n)$	$O(n)$	$O(n)$	$O(\lg n)$	$O(\lg n)$
Hashtabell (chaining)	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Hashtabell (öppen adr.)	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$

**Antag att arrayen hålls helt fylld från vänster och att man håller reda på index för elementet längst till höger*

I en (osorterad) lista måste man gå igenom hela listan från början varje gång man letar efter ett visst element. Man kan dock alltid stoppa in ett nytt element först i listan. I en sorterad lista måste ett nytt element stoppas in på rätt plats, och att hitta rätt plats tar linjär tid. Däremot kommer man åt det minsta elementet på konstant tid eftersom det är det första elementet (om listan är sorterad i stigande ordning).

I en array har man konstant åtkomst till ett element med ett givet index. Man kan också lägga in ett nytt element (sist) i arrayen på konstant tid. Tar man bort ett element i mitten måste man flytta resten av elementen för fylla hålet och hålla arrayen fylld från vänster (Om man tillåter att man byter ordning på elementen kan man förstås flytta dit det sista elementet på konstant tid istället). Har man en (fallande) sorterad array ligger det minsta elementet alltid sist och man kan ta bort det på konstant tid. Däremot måste man sätta in nya element på rätt plats och därför flytta alla andra element för att bereda plats, vilket ger linjär komplexitet.

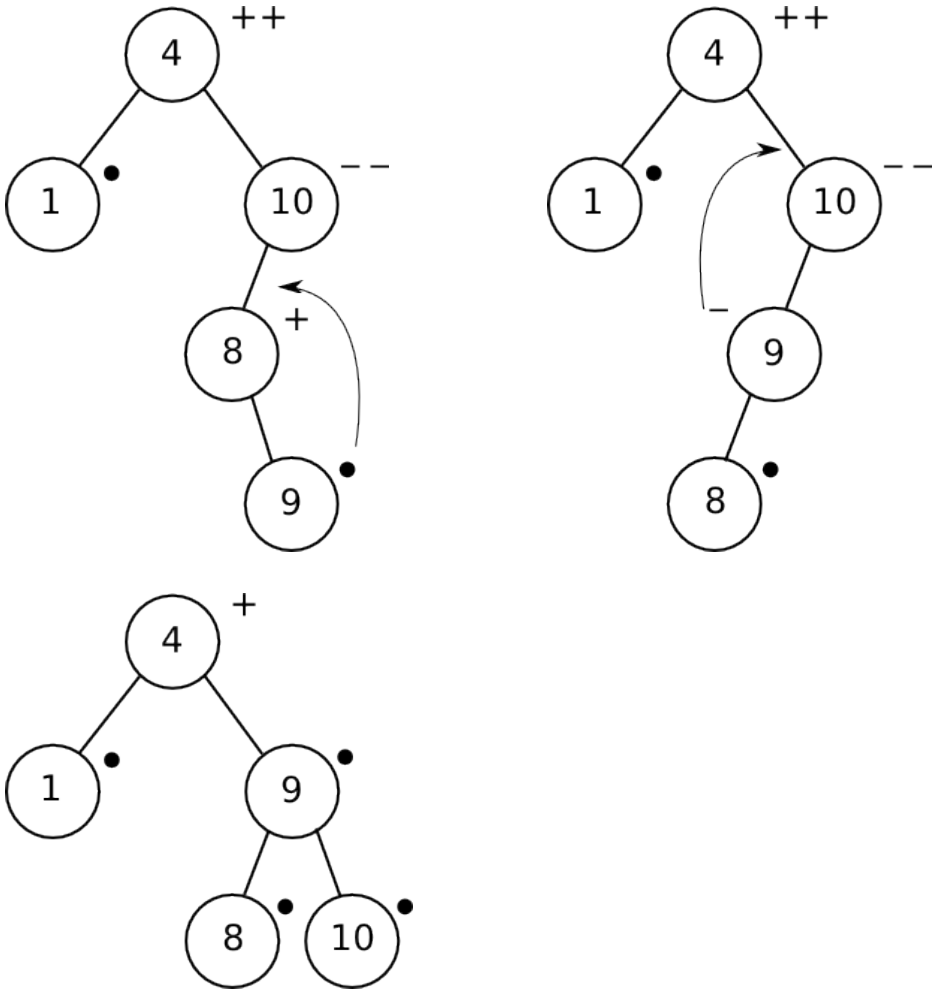
I ett balanserat binärt sökträd kan man i varje steg av en sökning välja bort (ungefär) hälften av elementen, vilket ger logaritmisk tidskomplexitet (enligt Master Theorem). Även borttagning av element kan implementeras med logaritmisk komplexitet. Ifall sökträdet inte är balanserat beter sig datastrukturen som värst som en länkad lista. Ett AVL-träd är som bekant alltid balanserat. När man letar efter det minsta elementet får man däremot problem även om trädet är balanserat, eftersom man sorterar trädet på nycklar, inte på data. Man måste leta igenom hela trädet för att hitta det minsta. Elementet med minst nyckel kan förstås hittas på logaritmisk tid.

En binomial min heap har inte samma egenskaper som ett sökträd, därför kräver åtkomst och borttagning att man letar igenom hela strukturen. Däremot kan man både sätta in ett element och ta ut det minsta elementet med logaritmisk komplexitet.

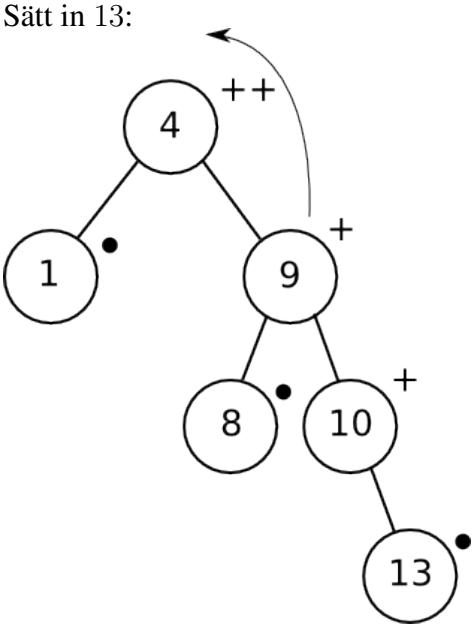
En hashtabell med en dålig hashfunktion kommer leda till många kollisioner, vilket i värsta (och mest osannolika) fall leder till att alla element hamnar i en enda lång kedja på en plats i tabellen, eller att man måste proba förbi alla andra element innan man hittar det man letar efter. Vid bra val av hashfunktion har man dock konstant komplexitet för att hitta element. Däremot finns ingen sortering i en hash-tabell, alltså måste man totalsöka tabellen för att hitta det minsta elementet.

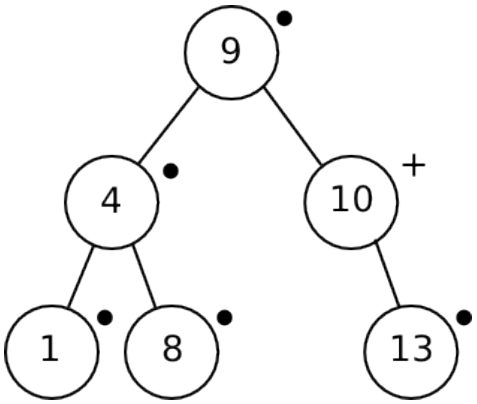
AVL-träd

Sätt in $4, 10, 1, 8$ och 9 i ett AVL-träd:

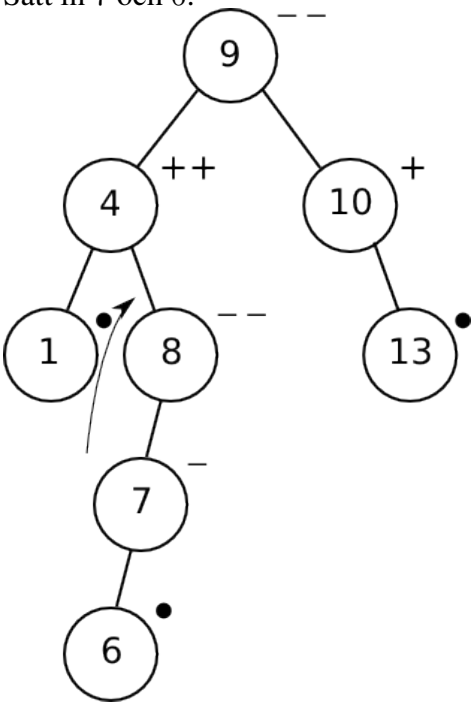


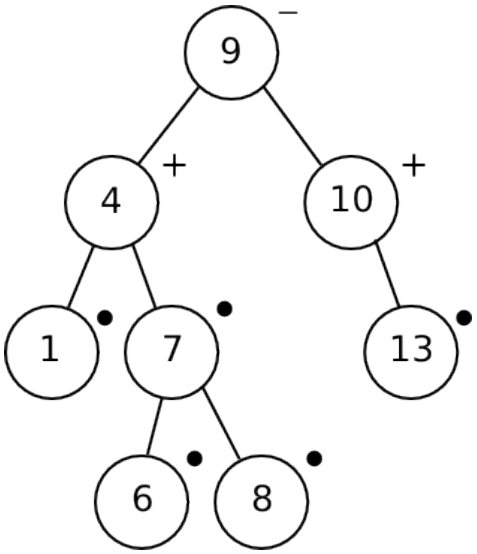
Sätt in 13:



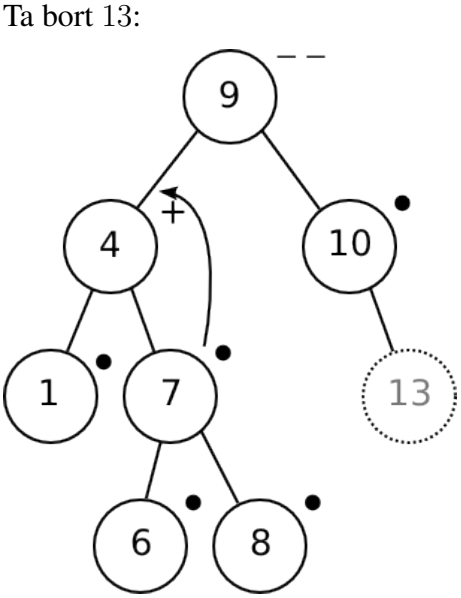


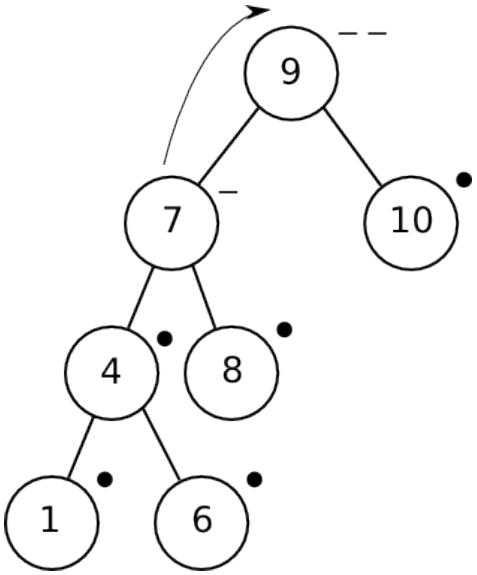
Sätt in 7 och 6:

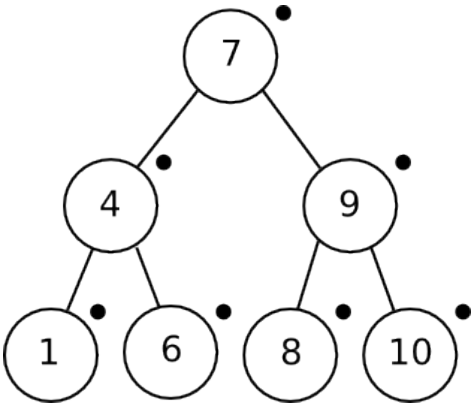




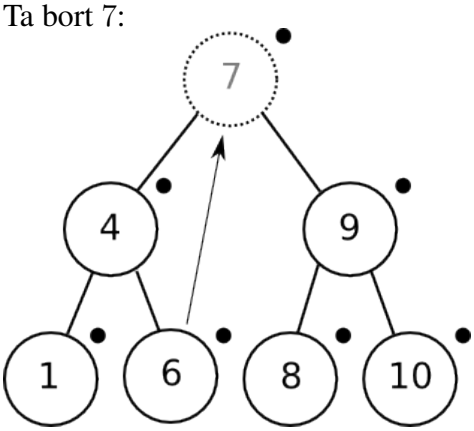
Ta bort 13:

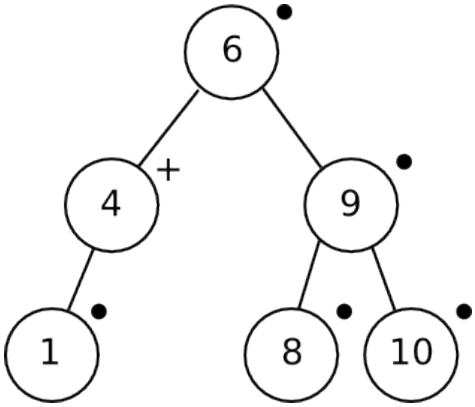






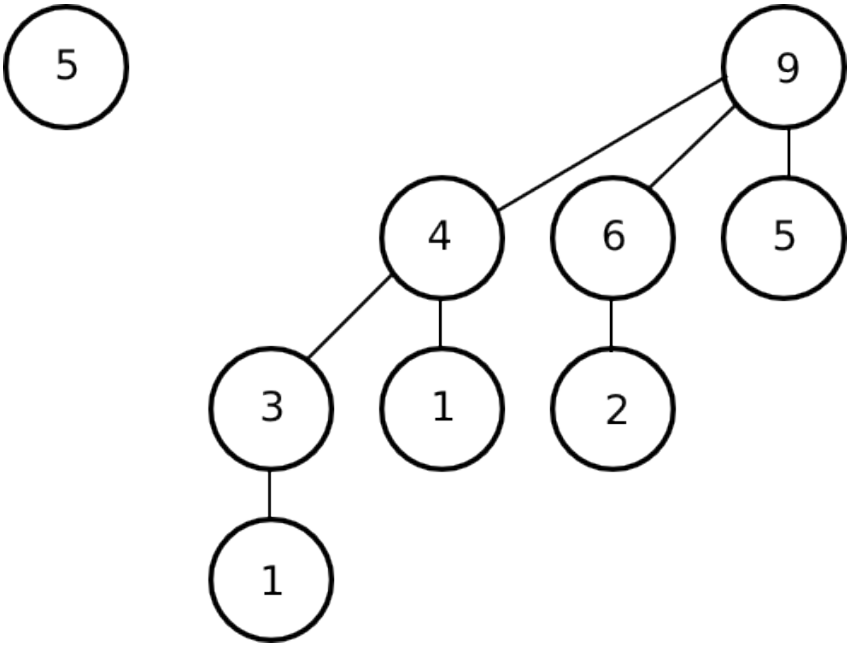
Ta bort 7:



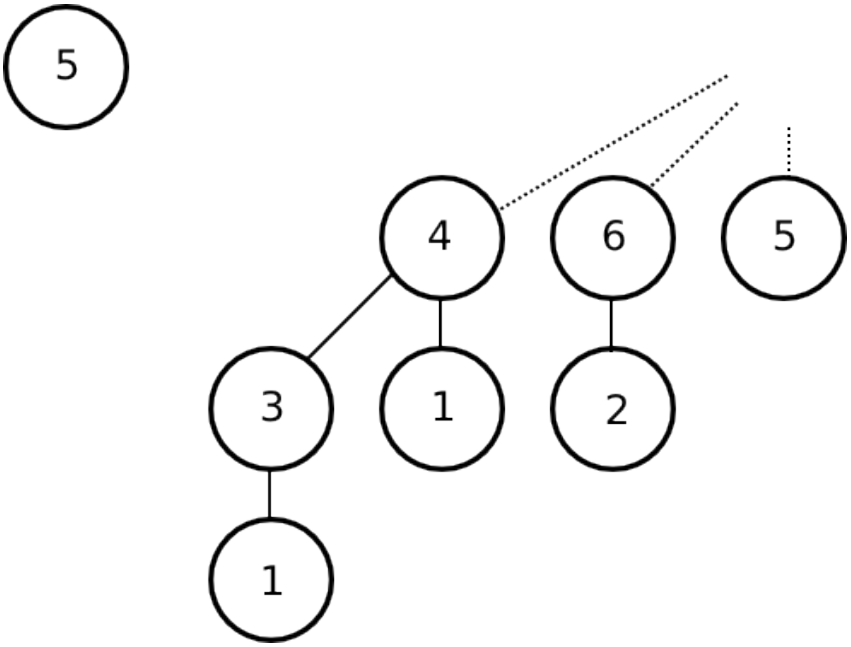


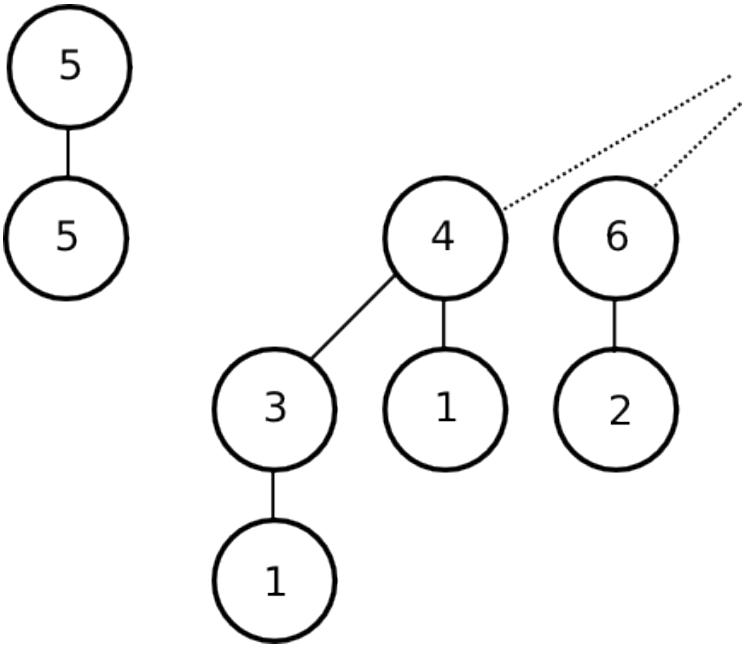
Binomial max heap

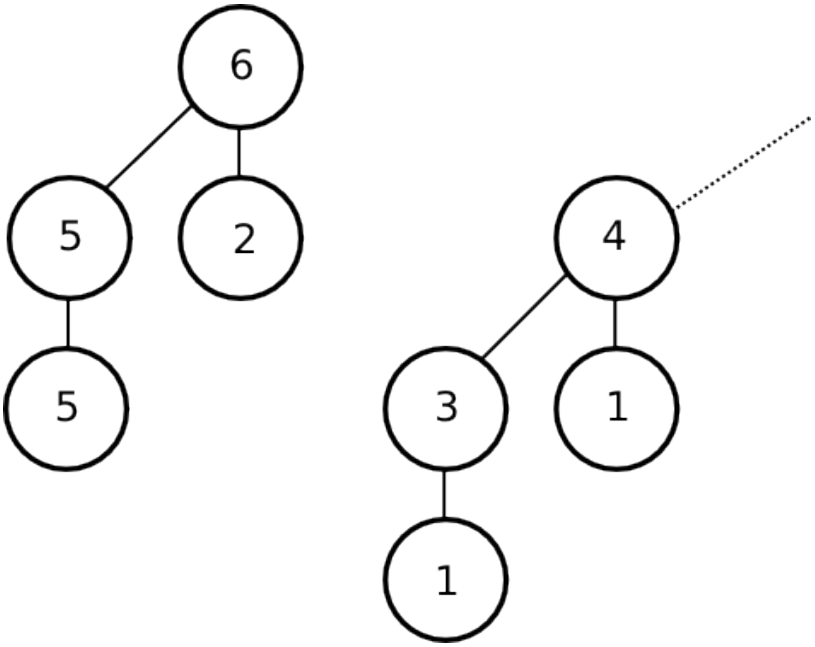
Efter att man har satt in $3, 1, 4, 1, 5, 9, 2, 6$ och 5 i en binomial max heap har den följande utseende:

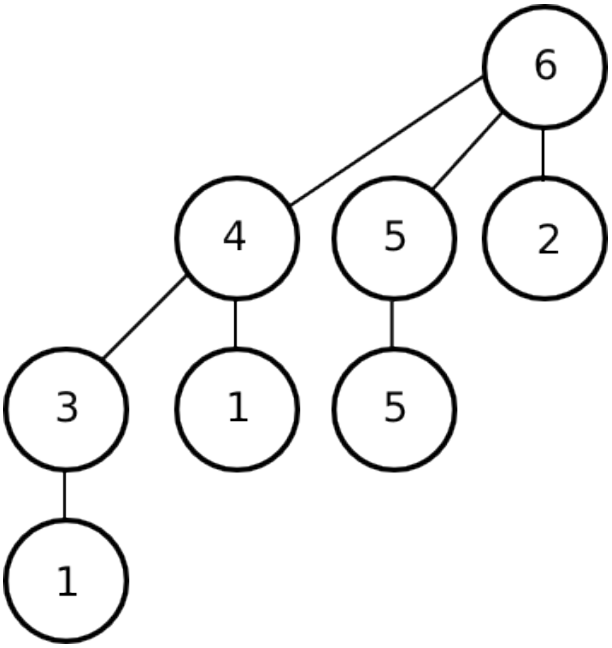


Tar sedan bort det största elementet sker följande förändringar

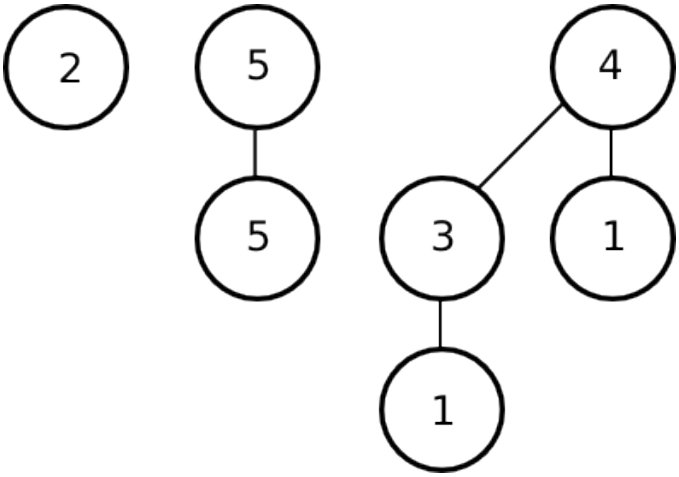


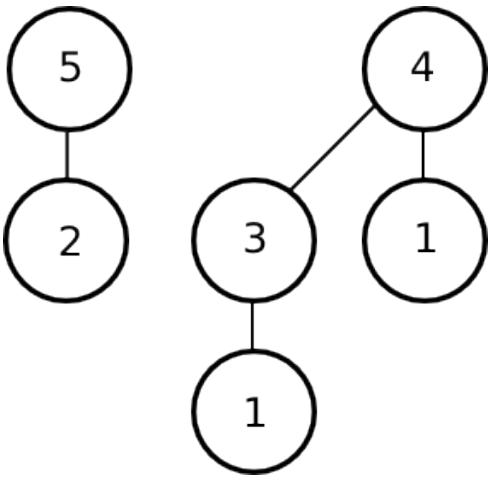

5
5
4
6
3
1
2
1

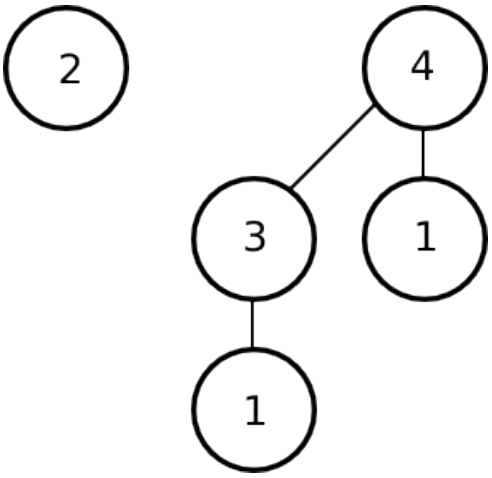

6
5
2
5
4
3
1
1



Resterande tre borttagningar resulterar i följande heapar:







Hashtabeller

Sätt in följande element (i given ordning och utan att rehasha) i en hashtabell med 7 celler, hashfunktionen $h(k) = k \ mod \ 7$ och probfunktionen $p(i) = i^2$.

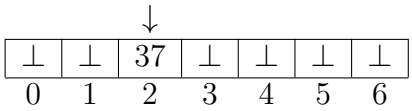
$$37, 13, 31, 48, 23, 69, 22$$

Leta upp och ta bort elementen $37, 22, 31$ och 69.

k	$h(k) = k \ mod \ 7$
37	2
13	6
31	3
48	6
23	2
69	6
22	1

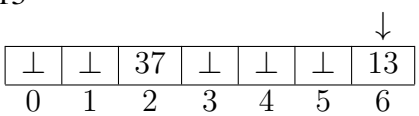
Sätt in 37

$i = 0$



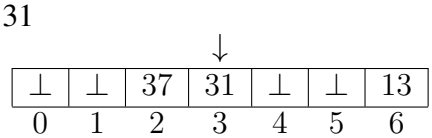
Sätt in 13

$i = 0$



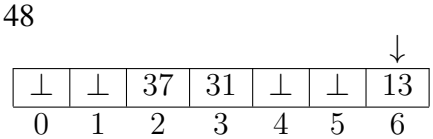
Sätt in 31

$i = 0$

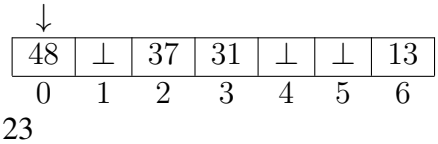


Sätt in 48

$i = 0$

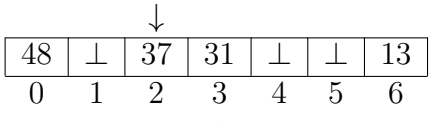


$i = 1$

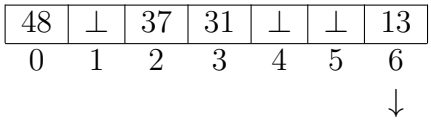


Sätt in 23

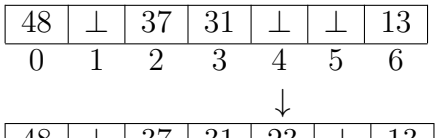
$i = 0$



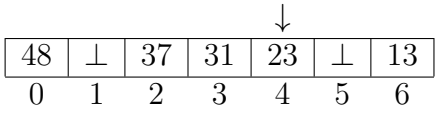
$i = 1$



$i = 2$



$i = 3$



Sätt in 69

$i = 0$

						↓
48	⊥	37	31	23	⊥	13
0	1	2	3	4	5	6

$i = 1$

↓						
48	⊥	37	31	23	⊥	13
0	1	2	3	4	5	6

$i = 2$

			↓			
48	⊥	37	31	23	⊥	13
0	1	2	3	4	5	6

$i = 3$

	↓					
48	69	37	31	23	⊥	13
0	1	2	3	4	5	6

Sätt in 22

$i = 0$

	↓					
48	69	37	31	23	⊥	13
0	1	2	3	4	5	6

$i = 1$

		↓				
48	69	37	31	23	⊥	13
0	1	2	3	4	5	6

$i = 2$

					↓	
48	69	37	31	23	22	13
0	1	2	3	4	5	6

Ta bort 37

$i = 0$

		↓				
48	69	Δ	31	23	22	13
0	1	2	3	4	5	6

Ta bort 22

$i = 0$

	↓					
48	69	Δ	31	23	22	13
0	1	2	3	4	5	6

$i = 1$

		↓				
48	69	Δ	31	23	22	13
0	1	2	3	4	5	6

$i = 2$

					↓	
48	69	Δ	31	23	Δ	13
0	1	2	3	4	5	6

Ta bort 31

$i = 0$

		↓				
48	69	Δ	Δ	23	Δ	13
0	1	2	3	4	5	6

Ta bort 69

$i = 0$

						↓
48	69	Δ	Δ	23	Δ	13
0	1	2	3	4	5	6

$i = 1$

↓						
48	69	Δ	Δ	23	Δ	13
0	1	2	3	4	5	6

$i = 2$

			↓			
48	69	Δ	Δ	23	Δ	13
0	1	2	3	4	5	6

$i = 3$

	↓					
48	Δ	Δ	Δ	23	Δ	13
0	1	2	3	4	5	6