Woodworkers handbook on trees

Anton Augustsson

June 2020

Innehåll

Kapitel 1

basic-course-in-mathematics - Formel Samling

1.1 Förord

Formel samlingen omfatar kursmatrialet från baskursen i matematik. Utöver formlerna, lyfts tillvägagonsätt för övertekande exempel. Syftet med formelsamlingen är att lyfta fram det viktigaste delarna så att det påmineren. Med att skriva en fusk papper lär ens sig oftas det man lätt glömer. Teori är en del i kursen en annan del som är lik så viktigt är utförandet av uppgifterna, i förods paragraphen så näms viktiga tips för att lyckas med upgifterna på tentamen.

Den försa tipset är att altid tesa rötterna till lösningen, exempelvis vid logaritmer då negativa logaritmer är odefinerad. Vid komplexa tall skall det altid finnas lika många lösningar som ledande termens exponent. Trigometri formlärna är bra att kunna utantill far att det är inte altid så uppenbara. Polärform måste också kommas ihåg utantill men är mer lågisk. Förflyting i x och y led i kordinatsystem är lätt att förväxla f(x-1) betyder en förlyting åt höger –> x behöver vara en mer för att upnå sama värde.

1.2 Basic

1.2.1 Mängder

Naturliga tal: $\mathbb{N} = \{1, 2, 3...\}$
Heltal: $\mathbb{Z} = \{... - 2, 1, 0, 1, 2...\}$
Rationella tal: $\mathbb{Q} = \{\frac{a}{b} | a, b \in \mathbb{Z}, b$
Irrationella tal: $\mathbb{P} = \frac{\mathbb{R}}{\mathbb{Q}} eller \{x | x \in \mathbb{R}, x \notin \mathbb{Q}\}$
Reella tal: $\mathbb{R} = \mathbb{P} \cup \mathbb{Q}$

1.2.2 Potenslagar

$$(\frac{a}{b})^{-3} = (\frac{b}{a})^3$$
$$\sqrt{a} = a^{\frac{1}{2}}$$

1.2.3 Logaritmer

Logaritmlagar:

Va vaksam över när a<0 då är logarytmen odefinerad.

(1): $b = a^x \Leftrightarrow \log_a(b) = x$ för: $a > 0, b > 0, a \neq 1$
(2): $\log_a(\frac{b}{c}) = \log_a(b) - \log_a(c)$
(3): $\log_a(b * c) = \log_a(b) + \log_a(c)$
(4): $\log_a(b^d) = d \log_a(b)$
(5): $\log_a(b) = \frac{\log_f(b)}{\log_f(a)}$
(6): $\log_a(a) = 1$
(7): $\log_a(1) = 0$
(8): $a^{\log_a(x)} = x$
(9): $\log_{a^c}(b) = \frac{1}{c} \log_a(b)$

1.2.4 Tillvägagångs sätt

$$\log_3(x) + \log_x(\frac{1}{27}) = 2 \tag{1.1}$$

Lösning

$$\log_3(x) - 3\log_x(3) = 2$$
$$\log_3(x) - 3\frac{\log_3(3)}{\log_3(x)} = 2$$
$$\log_3(x) - 3\frac{1}{\log_3(x)} = 2$$
$$\log_3(x)^2 - 3 = 2\log_3(x)$$
$$y = \log_3(x)^2$$
$$y^2 - 2y - 3 = 0$$
pq-formeln: $y = 1 \pm \sqrt{4} = 1 \pm 2$
$$\log_3(x) = 3 \Leftrightarrow x = 3^3 = 27$$
$$\log_3(x) = -1 \Leftrightarrow x = 3^{-1} = \frac{1}{3}$$

1.2.5 Intervall

$$[\,a, b] = \{x | a \leq x \leq b\} \tag{1.2}$$

$$[\,a, \infty[\tag{1.3}$$
$$]-\infty, \infty[\tag{1.4}$$

1.2.6 Tillvägagångs sätt

$$\frac{2}{x-3} < \frac{5}{x} \tag{1.5}$$

Lösning

$$\frac{2}{x-3} - \frac{5}{x} < 0$$
$$\frac{(x)2}{x(x-3)} - \frac{5(x-3)}{x(x-3)} < 0$$
$$\frac{2x - 5x - 15}{x(x-3)} < 0$$
$$\frac{-3x - 15}{x(x-3)} < 0$$
$$\frac{-3(x-5)}{x(x-3)} < 0$$
$$x \neq 0, x \neq 3$$

Värde tabell:

	x<0	0<x<3	3<x<5	5<x
x-5	-	-	-	+
-3	-	-	-	-
x	-	+	+	+
x-3	-	-	+	+
hela	+	-	+	-

1.3 Komplexa tal

$z = a + bi$
$\mathrm{Re}(z) = a$
$\mathrm{Im}(z) = b$
$\overline{z} = a - bi$
$|z| = \sqrt{a^2 + b^2}$ Där b(Im(z)) är för y-axeln och a(Re(z)) är för x-axeln

Samma räkneregler för reella tal gäller för komplexa tal

1.3.1 Polärform

$\arg(z) = \alpha + 2\pi * n$
arg(z) är vinkeln mellan a (x-axeln) linjen |z|. n är heltal
$\mathrm{Arg}(z) \in]-\pi, \pi[$
$\mathrm{Arg}(z) \in \arg(z)$
$a = |z| \cos\alpha$
$b = |z| \sin\alpha$
$z = |z| \cos\alpha + i * |z| \sin\alpha$

Polärform:

$$z = |z|(\cos\alpha + i * \sin\alpha) \tag{1.6}$$

Eulers formel:

$$z = |z|e^{i\alpha} \tag{1.7}$$

Sats:

$|z * w| = |z| * |w|$
$\arg(z * w) = \arg(z) + \arg(w)$
$\arg(\frac{z}{w}) = \arg(z) - \arg(w)$

De Movre's formel:

$$(\cos(\theta) + i * \sin(\theta))^n = \cos(n * \theta) + i * \sin(n * \theta)$$

Binomisk ekvation

$$(\cos(\theta) + i * \sin(\theta))^n = \cos(n * \theta) + i * \sin(n * \theta)$$

Tillvägagångs sätt

Visa lösningarna i det komplexa tal planet $z^5 = \sqrt{3} + i$ (1.8)

$|\sqrt{3} + i| = \sqrt{\sqrt{3}^2 + 1^2} = \sqrt{4} = 2$

$\begin{cases} \sqrt{3} = 2\cos(\alpha) \\ 1 = 2\sin(\alpha) \end{cases}$

$\alpha = \dfrac{\pi}{6}$

$|z|^5(\cos(5 * \theta) + i * \sin(5 * \theta)) = 2(\cos(\dfrac{\pi}{6}) + i * \sin(\dfrac{\pi}{6}))$

Vilket ger:

$|z|^5 = 2 \Leftrightarrow |z| = 2^{\frac{1}{5}}$

$5 * \theta = \dfrac{\pi}{6} + 2\pi n \Leftrightarrow \theta = \dfrac{\pi}{30} + \dfrac{2\pi n}{5} n \in \mathbb{Z}$

I polärform blir det då:

$z = 2^{\frac{1}{5}}(\cos\dfrac{\pi}{30} + \dfrac{2\pi n}{5} + i * \sin\dfrac{\pi}{30} + \dfrac{2\pi n}{5})$

Eulers formel: $z = 2^{\frac{1}{5}} e^{i(\frac{\pi}{30} + \frac{2\pi n}{5})}$

$n = 0, 1, 2, 3, 4$

$z_1 = 2^{\frac{1}{5}} e^{i(\frac{\pi}{30})}$

$z_2 = 2^{\frac{1}{5}} e^{i(\frac{\pi}{30} + \frac{2\pi}{5})}$

$z_3 = 2^{\frac{1}{5}} e^{i(\frac{\pi}{30} + \frac{4\pi}{5})}$

$z_4 = 2^{\frac{1}{5}} e^{i(\frac{\pi}{30} + \frac{6\pi}{5})}$

$z_5 = 2^{\frac{1}{5}} e^{i(\frac{\pi}{30} + \frac{8\pi}{5})}$

1.4 Absolut Belopp

1.4.1 Tillvägagångs sätt

$$|2x-8|+|1-x|-2|x-3|=8+3x \tag{1.9}$$

lösning

$$\left(\begin{array}{c} 2x-8=0 \\ x=4 \end{array}\right)\left(\begin{array}{c} 1-x=0 \\ x=1 \end{array}\right)\left(\begin{array}{c} x-3=0 \\ x=3 \end{array}\right)$$

$$\begin{array}{ll}
\text{I} & \left\{\begin{array}{l} x\leq 1 \\ -(2x-8)+(1-x)+2(x-3)=8+3x \end{array}\right. \\
\text{I} & \left\{\begin{array}{ll} x\leq 1 & \left\{\begin{array}{l} x\leq 1 \\ x=-\frac{5}{4} \text{ lösning} \end{array}\right. \\ -4x=5 & \end{array}\right. \\
\text{II} & \left\{\begin{array}{l} 1<x\leq 1 \\ -(2x-8)-(1-x)+2(x-3)=8+3x \end{array}\right. \\
\text{II} & \left\{\begin{array}{ll} 1<x\leq 3 & \left\{\begin{array}{l} 1<x<3 \\ x=-\frac{7}{2} \text{ Ej lösning} \end{array}\right. \\ -2x=7 & \end{array}\right. \\
\text{III} & \left\{\begin{array}{l} 3\leq x<4 \\ -(2x-8)-(1-x)-2(x-3)=8+3x \end{array}\right. \\
\text{III} & \left\{\begin{array}{ll} 3\leq x\leq 4 & \left\{\begin{array}{l} 3x<4 \\ x=\frac{5}{6} \text{ Ej lösning} \end{array}\right. \\ -(2x-8)-(4x-3)=8+3x & \end{array}\right. \\
\text{IV} & \left\{\begin{array}{l} x\geq 4 \\ (2x-8)-(1-x)-2(x-3)=8+3x \end{array}\right. \\
\text{IV} & \left\{\begin{array}{l} x\geq 4 \\ x=-\frac{11}{2} \text{ Ej lösning} \end{array}\right.
\end{array}$$

1.5 Summor

1.5.1 Aritmetiska summor

$$s_n = a_1 + a_2 + a_3 + \ldots + a_n = \frac{n\,(a_1 + a_n)}{2} \tag{1.10}$$

1.5.2 Geometriska summor

Börjar altid med exponenten 0 och gör om summan så att den passar i följade talföljd:

$$s_n = a + ak + ak^2 + \ldots + ak^{n-1} = \frac{a\,(k^n - 1)}{k - 1} \tag{1.11}$$

1.5.3 Tillvägagångs sätt

$$\sum_{k=n}^{2n} (2^k - k) \tag{1.12}$$

Lösning

sätter f = 0 = k -n

$$\sum_{f=0}^{n} (2^{f+n} - (f+n)) = 2^n * \sum_{f=0}^{n} (2^f) - \sum_{f=0}^{n} (f+n)$$

$$\frac{2^n(2^{n+1} - 1)}{2 - 1} - \frac{3n(n+1)}{2} = 2^{2n+1} - 2^n - \frac{3n(n+1)}{2}$$

1.6 Kombinatorik

1.6.1 Multiplikations principen

permetationer p(a,b) då ordningen spelar roll.
Antal sätt: (exemplet: antal sätt av måltider 7 företer 5 varmrätter 4 efterätter (7*5*4)

$$n_1 \cdot n_2 \cdot \ldots \cdot n_m \tag{1.13}$$

1.6.2 Kombinationer

Kombination c(a,b) då ordningen inte spelar roll.
Antal sätt: (exemplet: antal del-mängder två element (n = element k = antal element som kan väljas))

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!} \tag{1.14}$$

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n \cdot (n-1)(n-2) \cdot \ldots \cdot (n-1(k-1))}{k!} \tag{1.15}$$

Pascal triangel

n							
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1
	0	1	2	3	4	5	6
				k			

1.6.3 Binomial satsen

$$(a+b)^n = \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) a^k b^{n-k} \tag{1.16}$$

1.7 Funktioner och kordinatsystem

Kom ihåg att när det stor (x-a) förflytas den i x-axeln a steg till höger –> medans (x+a) förflytas den a steg till vänster <–

1.7.1 Avståndsformeln

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{1.17}$$

1.7.2 Elipser

Förenkla till denna formeln:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{1.18}$$

Där a är avståndet på x-axeln och b är avståndet på y-axeln

Hyperbol

Ser väldigt anerlunda ut från elipser

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1 \tag{1.19}$$

Circkel

$$\frac{x^2}{a^2} + \frac{y^2}{a^2} = 1 \tag{1.20}$$

Där a = r

1.8 Polynom division

1.8.1 Tillvägagångs sätt

Man vet att ekvationen$z^4-2z^3-7z^2+26z-20=0$ har roten $z=2+i$. Lös ekvationen fullständigt. (1.21)

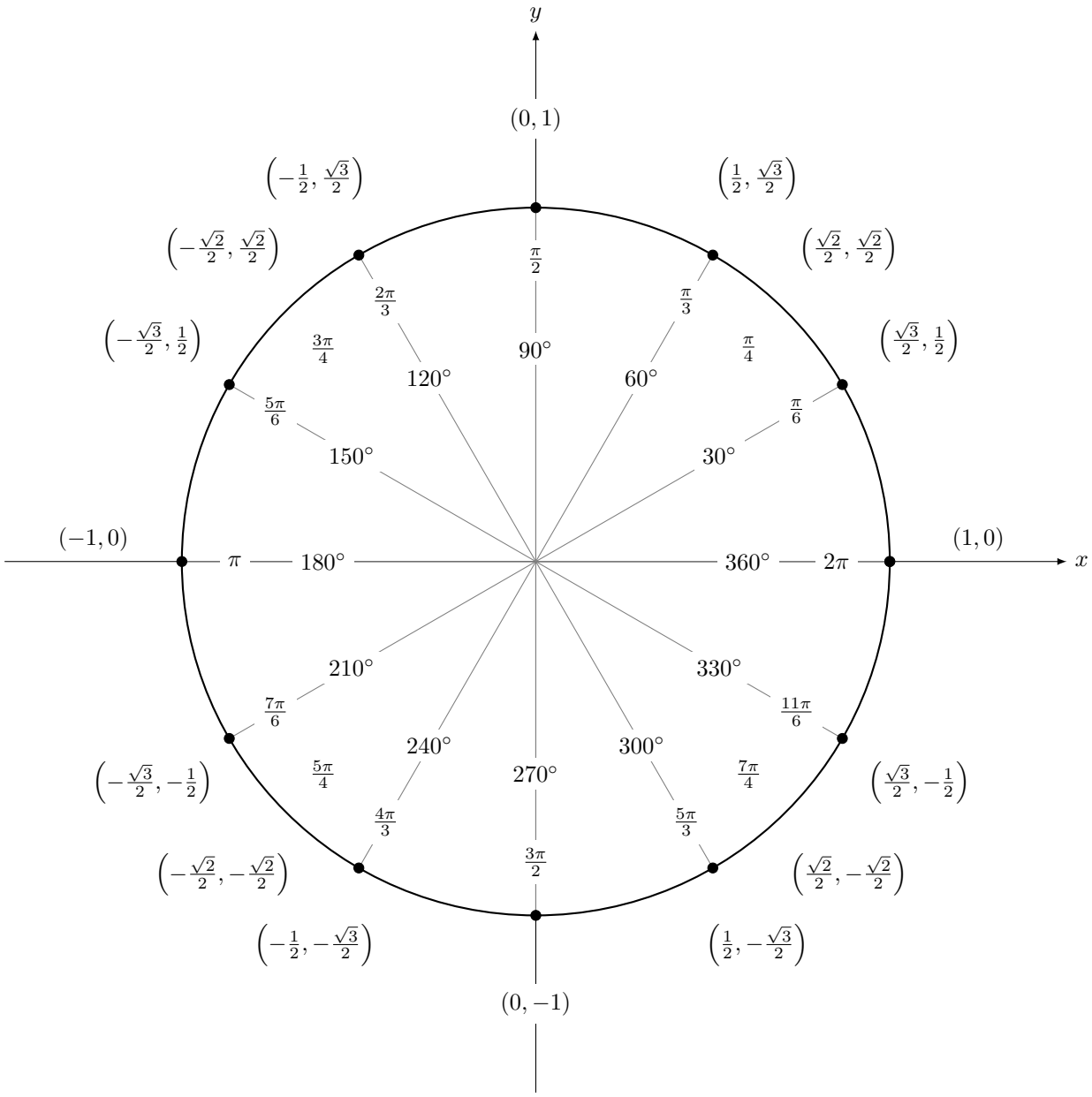
$z=2+i$ Är en läsning är också konjugatet en lösning enligt faktorsatsen $\bar{z}=a-bi$
$z=2\pm i$
Vilket betyder att följande går att factorisera ut polynomet
$(z-(2+i))(z-(2-i))=z^2-4z+5$

Långdivition (liggande stolen):

$$
\begin{array}{rl}
 & x^2 + 2x - 4 \\
\hline
x^2-4x+5\big) & x^4 - 2x^3 - 7x^2 + 26x - 20 \\
 & -x^4 + 4x^3 - 5x^2 \\
\hline
 & 2x^3 - 12x^2 + 26x \\
 & -2x^3 + 8x^2 - 10x \\
\hline
 & -4x^2 + 16x - 20 \\
 & 4x^2 - 16x + 20 \\
\hline
 & 0
\end{array}
$$

$z^2+2z-4=0$ Är också en läsning som tilslut ger följande
$z=-1\pm\sqrt{5}$
$z=2\pm i$
Varge n grads polynom har altid n stycken komplexa lösningar

1.9 Trigonometri



Sats:

$$360^\circ = 2\pi rad$$
$$v_g = v_r * \frac{180^\circ}{\pi}$$
$$v_r = v_g * \frac{\pi}{180^\circ}$$

Sats:

$$-1 \leq \sin t \leq 1$$
$$-1 \leq \cos t \leq 1$$

Sats:

$$\cos(-t) = \cos(t)$$
$$\sin(-t) = -\sin(t)$$
$$\tan(-t) = \frac{\sin(-t)}{\cos(-t)} = \frac{-\sin(t)}{\cos(t)}$$

Additionsformlerna:

$$\sin(\alpha+\beta) = \sin(\alpha)\cos(\beta) + \sin(\beta)\cos(\alpha)$$
$$\sin(\alpha-\beta) = \sin(\alpha)\cos(\beta) - \sin(\beta)\cos(\alpha)$$
$$\cos(\alpha+\beta) = \cos(\alpha)\cos(\beta) - \sin(\beta)\sin(\alpha)$$
$$\cos(\alpha-\beta) = \cos(\alpha)\cos(\beta) + \sin(\beta)\sin(\alpha)$$

Trigonometriska ettan:

$$(\sin t)^2 + (\cos t)^2 = 1$$
$$\sin^2 t + \cos^2 t = 1$$

1.9.1 Tillvägagångs sätt

$$\cos\frac{\pi}{12} = \cos\left(\frac{\pi}{3} - \frac{\pi}{4}\right) = \cos\frac{\pi}{3}\cos\frac{\pi}{4} + \sin\frac{\pi}{3}\sin\frac{\pi}{4}$$
$$= \left(\frac{1}{2}\right)\left(\frac{1}{\sqrt{2}}\right) + \left(\frac{\sqrt{3}}{2}\right)\left(\frac{1}{\sqrt{2}}\right) = \frac{1+\sqrt{3}}{2\sqrt{2}}$$

Kapitel 2

PKD Samanfatning

2.1 Förord

PKD is a 20hp curse focusing on functional programming in Haskell and data structures.

Inför tentan del 1 kom ihåg att PRE är omm en vis data som tillhör värde mängden inte fungerar på functionen altså ger error (runtime-error).

Inför tenta del 2 är det en fråga som är altid litte jobbig om den hadlar om binomial heaps. När det står bara inserting då ska man inte merga om det står brobing merging eller något anat då ska man göra det.

In topology sort go with indegre 0 then 1 then 2 it dosent mater how they are connecten only the ordering.

2.2 Coding convension

VARIANT
A (recursive) function terminates if it has a variant. The variant need to follow all of the flowing rules

- Needs to decrease every recursive call
- All ways positive

Side effects
All IO functions have side effects in order to separate pure Haskell function with impure functions that changes the state with is commonly the case with imperative and object oriented programming. Every IO function has a side effects.
INVARIANT
A data types invariant is what value are allowed for the data type to work. Similar to preconditions for a function. An example is integer data type that can only use positive integers therefor the invariant is positive integers.

2.3 Design approach

- top-down design (Cheating): Is to break down a complex system in to subsystems to solve the problem. Most often is to write everything by scratch.
- bottom-up design (Stacking): Is to pies existing system together to create a more complex system. little is programmed, most is copied.
- dodging: Get some code working more quickly, make progess with some part of the system and back-paddle to the dodged part later. The reason is to develop insight that will help solve the larger problem.

2.3.1 Process

1. Data Description
2. Data Examples
3. Function Description
4. Function Examples
5. Function Template
6. Code
7. Tests
8. Review and Refactor

Programming to an Interface More dynamic, can change models, less code to write and a layer of abstraction. ADT

2.4 Recursion

2.4.1 Recursion types

1. Simple recursion: There is at most one recursive call (in each branch) and the argument is decremented by one.
2. Complete recursion: Some argument becomes smaller in the recursive call, but not necessarily.
3. Multiple recursion: There are multiple recursive calls (in the same branch).
4. Mutual recursion: Two or more functions are defined in terms of each other.
5. Nested recursion: An argument to a recursive call is computed by a recursive call.
6. Recursion on a generalized problem: Sometimes, no suitable recursion scheme is obvious.

2.5 Complexity

2.5.1 Growth

1. Big θ Notation: estimate of growth in intervals determine by constants Definition For non-negative functions f and g, $f(n) = \theta(g(n))$ if and only if there exist $n_0 \geq 0 and c_1, c_2 > 0$ such that for all $n > n_0$ $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$. $\theta(g(n))$ is the set of all functions $f(n)$ that are bounded below and above
2. Big Ω Notation: estimate of growth Lower bound
3. Big O: Notation: estimate of growth upper bound

Relation

$O(g(n)) \cap \omega(g(n)) = \theta(g(n))$

2.6 Recurrences

Example:
sumList [] = 0
sumList (x:xs) = x + sumList xs

1. pattern matching [] takes t_0 time
2. pattern matching $(x : xs)$ takes t_1 time
3. Adding x with recursive call takes t_{add}
4. Then the recursive call takes $T(n-1)$

$$T(n) = \begin{cases} t_0 & \text{if } n = 0 \\ T(n-1) + t_{add} + t_1 & \text{if } n > 0 \end{cases}$$

2.6.1 Closed Form

1. Use the substitution method to obtain a closed form for the following recurrence:

$$f(0) = 5$$
$$f(n) = f(n-1) + n + 2, n > 0$$

Hint: $1 + 2 + 3 + \Delta\Delta\Delta + n = \frac{n(n+1)}{2}$ **— you do not need to prove this fact.**

Expansion Method

$$f(0) = 5$$
$$f(1) = f(0) + n + 2 = n + 5 + 2$$
$$f(2) = f(1) + n + 2 = 2n + 5 + 2 + 2$$
$$f(3) = f(2) + n + 2 = 3n + 5 + 2 + 2 + 2$$
$$f(n) = +n^2 + 5 + n \cdot 2$$

Induction proof

Step1: test with base case sense the base case is predefine for 0 we do $n = 1$
$f(1)_{VL} = 1^2 + 5 + 1 \cdot 2 = 8,$
$f(1)_{HL} = f(0) + 1 + 2 = 5 + 3 = 8$
$f(1)_{VL} = f(1)_{HL}$

Step2: assumption for p $f(p) = p^2 + 5 + p \cdot 2$
$f(p+1)_{VL} = f(p) + (p+1) + 2 = (p^2 + 5 + p \cdot 2) + (p+1) + 2 =$
$= (p^2 + (p+1)) + 5 + (p+1) \cdot 2 = (p^2 + 2p + 1) + 5 + (p+1) \cdot 2$
$f(p+1)_{HL} = (p+1)^2 + 5 + (p+1) \cdot 2 = (p^2 + 2p + 1) + 5 + (p+1) \cdot 2$
$f(p+1)_{VL} = f(p+1)_{HL}$

Conclusion: according to induction hypothesis the recurrence of the function is equal to
$2n + 5 + \frac{n(1+n)}{2}$

Substitution Method

$$\begin{aligned}
f(n) &= f(n-1)+n+2\\
&= (f(n-2)+(n-1)+2)+1n+2\\
&= f(n-2)+(n-1)+n+2\cdot 2\\
&= (f(n-3)+(n-2)+2)(n-1)+n+2\cdot 2\\
&= f(n-3)+(n-2)+(n-1)+n+3\cdot 2\\
&\cdot\\
&= f(n-k)+(n-(k-1))+(n-(k-2))+(n-(k-3))+\ldots+n+k\cdot 2\\
&\cdot\\
&= f(n-n)+(n-(n-1))+(n-(n-2))+(n-(n-3))+\ldots+n+n\cdot 2\\
&= f(0)+1+2+3+\ldots+n+n\cdot 2 = 5+1+2+3+\ldots+n+n\cdot 2
\end{aligned}$$

We can see the following patterns

$$2n+5+\sum_{k=1}^{n}(k) = 2n+5+\frac{n(1+n)}{2}$$

Induction proof

Step1: test with base case sense the base case is predefine for 0 we do $n=1$

$f(1)_{VL} = 2\cdot 1+5+\frac{1(1+1)}{2} = 2+5+1 = 8,$

$f(1)_{HL} = f(0)+1+2 = 5+3 = 8$

$f(1)_{VL} = f(1)_{HL}$

Step2: assumption for p $f(p) = 2p+5+\frac{p(1+p)}{2}$

$f(p+1)_{HL} = f(p)+(p+1)+2 = (2p+5+\frac{p(1+p)}{2})+(p+1)+2 =$

$= 2(p+1)+5+\frac{p(1+p)}{2}+p = 2(p+1)+5+\frac{2(p+1)+p(1+p)}{2} =$

$= 2(p+1)+5+\frac{p^2+2p+p+2}{2} =$

$f(p+1)_{HL} = 2(p+1)+5+\frac{(p+1)(p+2)}{2} = 2(p+1)+5+\frac{p^2+2p+p+2}{2}$

$f(p+1)_{VL} = f(p+1)_{HL}$

Conclusion: according to induction hypothesis the recurrence of the function is equal to

$2n+5+\frac{n(1+n)}{2}$

2.7 Higher-Order Function

2.7.1 Higher-Order Functions on Lists

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldl :: (a -> b -> a) -> a -> [b] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b
```

map

maps a function to each element in list.

filter

filters elements with a condision

```
filter :: (a -> Bool) -> [a] -> [a]
filter (<6) [6,3,0,1,8,5,9,3] == [3,0,1,5,3]
```

foldl

foldl recurses over a list "from the left," i.e., it initially applies the given operation to the first list element and the given start value. starts from the left (first element) and apply the function to with each element. Similar to an accumulator. No one uses it since it is some what useless.

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl (*) 1 [1,2,3,4] == 24
```

foldr

foldr recurses over a list "from the right," i.e., it initially applies the given operation to the last list element and the given start value. starts from the right (last element) and apply the function to with each element. Similar to an accumulator.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (*) 1 [1,2,3,4] == 24
```

2.8 Data types

2.8.1 Basic

```
String :: ['char'] --list of charecters
List :: []         --undefine element types and elements
Tuple :: ()        --Predefine elements
Char :: ''         --single carecter
Int :: 1           --Hole number with define size
Integear :: 1      --Hole number with undefine size
Float :: 1.1       --Real number with double-precision
Double :: 1.1      --Real number with single-precision
```

2.8.2 Maybe Type

If the return is maybe "nothing" then the "Maybe type" is used, since it dose not have to return the a specific value. It is not polymorphic since you have to specify the type, however "Just" is at of it self polymorphic function. If a operation that requires a specific type one needs to remove "Just", for instance by a let function.

2.8.3 New types and enumeration types

New types: One create more relevant names and format of existing enumeration types. Overload is a problem with the use of the same operations that can not be used for the same data types. One needs to create new operations if it is not from the same type class.

Enumeration types: Instead of new types *type* enumeration types uses the operation call *data*. The difference is that enumeration types is independent from existing types therefor one becomes more flexible and precise.

example

```
data newTypeOfDataDerection = North | South | East | West
   deriving (Show)  -- inorder to print

:t North
North :: newTypeOfDataDerection

-- We can use new types in pattern matching
oposit :: newTypeOfDataDerection -> newTypeOfDataDerection
oposit North = South
```

Type classes

A type class is a set of types that support certain related operations.

No function is applied by default to the new type, therefor you can write "deriving" the following functions are good to have

type classes to deriving for new data types

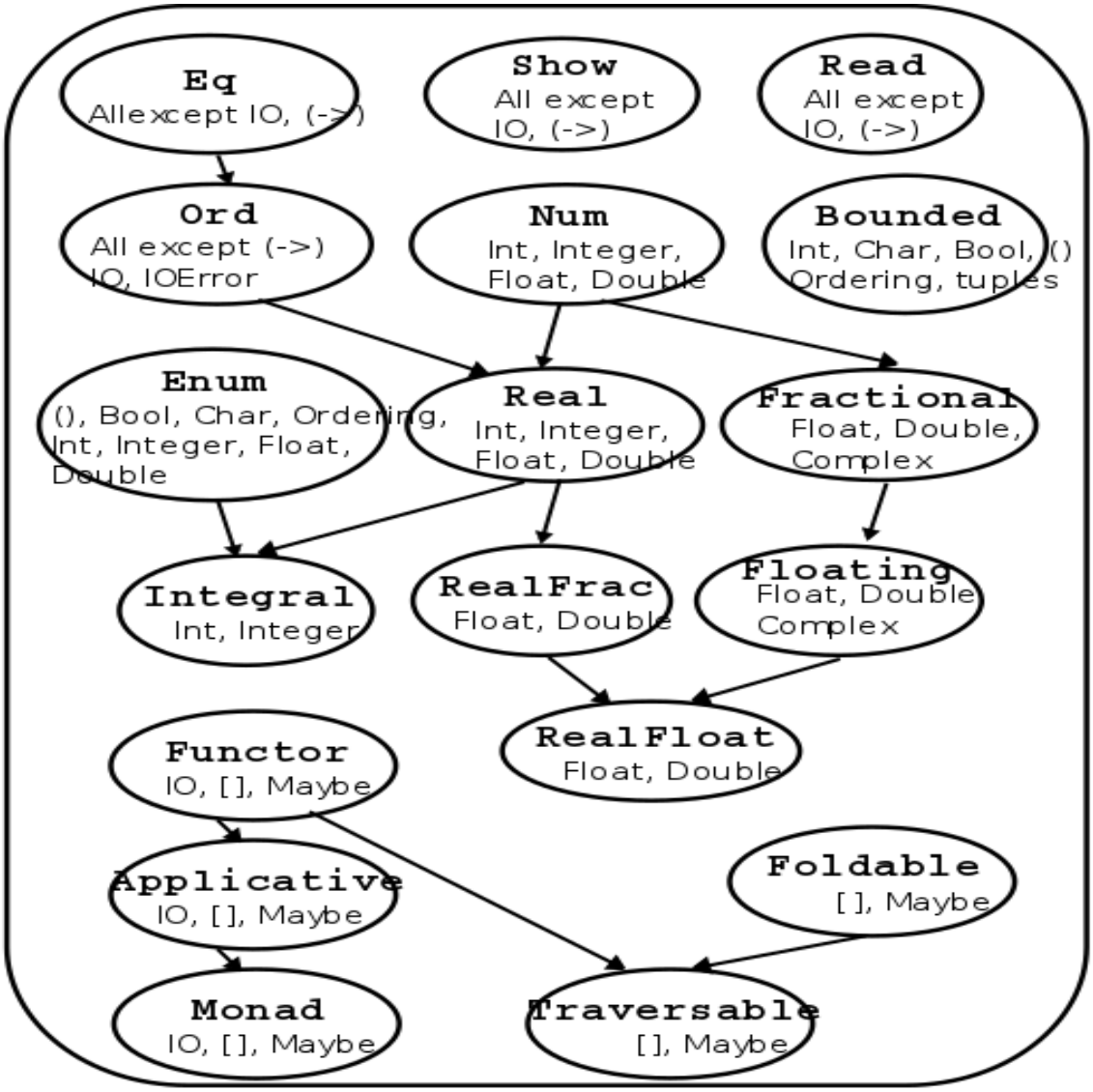
```
  deriving (Show)  -- in order to print in ghci and print normal
  deriving (Eq)  -- To test equality
  deriving (Ord)  -- the first one has the smallest value, order matter for comparison
```

New type classes

```
class Eq a where
   (==) :: a -> a -> Bool
```

New type classes Instances

```
instance Eq Colour where
  (==) = eqColour
```



Figur 2.1: Type class

2.8.4 Inductive Data Types

Uses a base case then the inductive step as cases of. Multiple arguments. Type constructure

```
data AExp = Atom Int
    | Plus AExp AExp
    | Times AExp AExp

eval (Atom i)     = i
eval (Plus a b)   = eval a + eval b
eval (Times a b) = eval a * eval b
```

2.8.5 Trees

Terminology

1. Search tree: All nodes on the left side is less then the parent and opposite on the right side
2. Out-degree: is how many children it has. Binary trees has out-degree $0 - 2$
3. root Node: is a parent to any number of children at the top of the hierarchy
4. sub Node: is a parent to any number of children
5. Leaf: is a nod that has no children
6. Node: every element
7. Height: most steps from the root node to a leaf

Representation

1. Inorder: (Left, Root, Right)
2. Pre-order: (Root, Left, Right)
3. Post-order: (Left, Right, Root)

```
data FBTree a = Leaf a
   | Node (FBTree a) a (FBTree a)
   deriving (Show)

Node (Leaf 1) 5 (Leaf 21)
rootValue :: FBTree a -> a
rootValue (Node _ a _) = a
rootValue (Node x) = x

height :: FBTree a -> Int
height (Leaf _) = 0
height (Node b a c) = 1 + max (height b) (height c)
```

Binary tree

Insertion: $O(1)$ ($O(h)$ if search tree)
Delition: $O(n)$ ($O(h)$ if search tree)
Search: $O(n)$ ($O(h)$ if search tree)
Height: $O(n)$ (n nodes)

1. Full binary tree: has node out-degree either 2 or 0 worst-case complexity of $O(\log n)$.
2. Binary tree: each node has up to an out-degree of 2

Binary search tree

Insertion: $O(h)$ ($O(n)$ if search tree)
Delition: $O(h)$
Search: $O(n)$
Height: $O(n)$ (n nodes)

Red and black trees

Insertion: $O(\log n)$
Deletion: $O(\log n)$
Search: $O(\log n)$
Element: $O(2^r)$ (rank r)
Height: $O(2 \cdot \log_2(n+1))$ (n nodes)

Better then a normal binary tree since it balance the tree, therefor it becomes smaller and more efficient to search in. One should use red and black tree when there is a large number of nodes, say 50.

Definition: A red-black tree is a binary search tree where every node is colored either red or black, with the following balancing invariants:

1. No red node has a red parent.
2. Every path from the root to an empty subtree contains the same.
3. A red-black tree with n nodes has height at most $2 \cdot \log 2(n+1)$.
4. there are 4 cases to rebalance (1;4) is similar so is (2;3).

Algorithm

1. Perform a standard binary-search-tree insertion.
2. Color the new node red.
3. Rebalance the tree, if there is a red node with a red parent.

Binomial Trees and heaps

Insertion: $O(\log n)$
Search: $O(\log n)$ (number of trees n)
Element: 2^r (rank r, n=1 then r=0)

A heap can be used to implement a priority queue, where elements are added to a pool and assigned a priority. In a min-priority queue, extraction of an element yields an element with minimum priority. The smallest node is the root and every child is equal or larger then its parent.

Binomial Trees is a data structure by linking trees of rank $r-1$ together. A binomial heap (Vuillemin, 1978) is a list of binomial trees such that each tree satisfies the min-heap property (hence the root of each tree contains its minimum key); and the trees have strictly increasing ranks.

Terminology

1. Link: putting together two trees.
2. Merge: putting together two heaps
3. Binomial Heap: a list (forest!) of Binomial Trees
4. Binomial Trees have the largest subtree to the left, while Binomial

Binomial heaps

1. Heaps, extracting minimum element in worst case $O(\log |h|)$
2. Binomial Trees, The height (here: number of edges on the longest branch) is r.
3. Binomial Trees, There are 2^r nodes in the tree.
4. Binomial Trees, There are $\begin{pmatrix} r \\ k \end{pmatrix}$ nodes at level k. (Hence its name!)
5. Binomial Trees, The root has r subtrees of ranks $r-1, r-2, \ldots, 1, 0$.
6. A binomial heap h has at most $\lceil \lg |h| \rceil + 1$ binomial trees.
7. Inserting a binomial tree into a binomial heap is like addition with base 2.
8. merging is made with too cases either (case 1) when one is smaller or (case 2) when they are equal

```
BinoTree = Node Int Int [BinoTree] -- Node rank key (subtrees with decreasing rank)
```

2.8.6 Other data types

Tables, Stacking and queuing

1. Table: a list of key-value pairs
2. Stacks: elements accessed in Last-In First-Out (LIFO) order
3. Queues: elements accessed in First-In First-Out (FIFO) order

Table operations

```
empty :: Table k v
insert :: Eq k => Table k v -> k -> v -> Table k v
exists :: Eq k => Table k v -> k -> Bool
lookup :: Eq k => Table k v -> k -> Maybe v  -- value from key
delete :: Eq k => Table k v -> k -> Table k v
iterate :: Table k v -> (b -> (k, v) -> b) -> b -> b  -- Foldr
keys :: Table k v -> (b -> k -> b) -> b -> b  -- all keys
values :: Table k v -> (b -> v -> b) -> b -> b  -- all values
```

Stack operations

```
-- interface
newtype Stack a = StackImpl [a] -- opaque!

empty :: Stack a
isEmpty :: Stack a -> Bool
push :: a -> Stack a -> Stack a  -- insert
top :: Stack a -> a  -- the first value
pop :: Stack a -> (a,Stack a)  -- take out
```

Queue operations

```
-- interface
newtype Queue a = Q [a] -- opaque

empty :: Queue a
isEmpty :: Queue a -> Bool
head :: Queue a -> a
enqueue :: Queue a -> a -> Queue a  -- take out element
dequeue :: Queue a -> Queue a  -- insert element
toList :: Queue a -> [a]
```

Hastables

- Key value lookup (an index)
- Array is only define for small index, hash has no limit on avalible keys.
- Typically we have n possible keys from set U for a hashtable (which is an array) with m slots, where $n \geq m$.
- since there is infinitely many elements and limited amount of key there will be element with the same key, therefor a coalition is created.
- Worst-Case Retrieval: time complexity of retrieving a element.
- Load Factor: How much data is in the table $\frac{elements}{slots}$
- Rehashing: make the hastable more balanced.

Collision Resolution by Chaining

- Most commonly used collision resolution
- Let each array slot (also called a bin) hold a list of elements (called a chain).
- In other words, When collision then add it to a list in that element

Collision Resolution by Open Addressing

- Start with a table with each element is $\perp$ previously used Δ.
- Probing: is a function to insert items in a hastable therefore resolves collations
- Types of probing:
  Linear probing: $f(i) = i$.
  Quadratic probing: $f(i) = c_2 \cdot i^2 + c_1 \cdot i$, *where* $c_2 \neq 0$.
  Double hashing: $f(i) = i \cdot h''(i)$, where h'' is another hash function.

- Inserting with Linear Probing: Insert it to the next avalible key
- Deleting with Linear Probing: Ignores $\perp$ and Δ idex will change.

2.8.7 Graphs

Types of graphs

- list
- tree
- forest
- Directed Acyclic Graph (DAG)

Treminolage

- Node, vertex (plural: vertices)
- Edge connects two nodes.
- Self-loop edge from node to itself
- Adjacent nodes connected by an edge
- Degree number of edges from or to a node
- In-degree number of edges to a node
- Out-degree number of edges from a node

Representation

- **Adjacency Matrix** — a 2-dimensional array A of 0/1 values, with $A[i, j]$ containing the number of edges between nodes i and j (undirected graph), or from node i to node j
  + edge existence testing in $\theta(1)$ time
  − finding next outgoing edge in $O|V|$ time
  + compact representation for dense graphs (when $|E|$ is close to $|V|^2$)
- **Adjacency List** — a 1-dimensional array Adj of adjacency lists, with Adj[i] containing a list of the nodes adjacent to node i.
  + finding next outgoing edge in $\theta(1)$ time
  − edge existence testing in $O(|V|)$ time
  + compact representation for sparse graphs (when $|E|$ is much smaller than $|V|^2$)
- **Edge List** — a list of tuples, (i, j), for each edge (i, j) (plus a list of the nodes).
  − edge existence test in $\theta(|E|)$.
  − finding next outgoing edge in $O(|E|)$ (unless appropriately sorted).
  + compact representation for sparse graphs (when $|E|$ is much smaller than $|V|^2$)

topological sort

A topological sort is a linear ordering of all the nodes in a directed acyclic.

Algorithm

1. Select a node with in-degree 0.
2. Output it.
3. Remove it.
4. Repeat (from 1) until no nodes are left.

Total running time is $\theta(|V| + |E|)$.

Graph Traversals

- Breadth-first search (BFS). Uses a queue for eatch grey node.
  Time complexity: $\theta(|V| + |E|)$ — linear in the size of the graph.
- Depth-first search (DFS). Uses a stack for eatch (grey) node and has a rest of nodes (white).
  Time complexity: $\theta(|V| + |E|)$ — linear in the size of the graph.

Breadth-First Search: Algorithm
Input: Some node A.

1. Paint A gray. Paint other nodes white. Add A to an initially empty FIFO queue of gray nodes. All grey nodes is in the queue. It is a queue not a stack so first in first out.
2. Dequeue head node, X. Paint its undiscovered (white) adjacent nodes gray and enqueue them. Paint X black. Repeat until queue is empty. For every black node add it to BFS order (black)

Depth-First Search: Algorithm
Input: Some node A.

DFS(G)

1. Paint all nodes white.
2. For each node v in G: if v is (still) white, DFS-Visit(G,v). Each subsequent call to DFS-Visit in line 2 is called a restart.

DFS(G)

1. Colour v gray.
2. For each node u adjacent to v: if u is white, DFS-Visit(G,u).

Strongly-Connected Components

- Strongly connected component (SCC): maximal set of nodes where there is a path from each node to each other node.
- Many algorithms first divide a digraph into its SCCs, then process these SCCs separately, and finally combine the sub-solutions. (This is not divide & conquer, since a different algorithm is run on each SCC!)
- An undirected graph can be decomposed into its connected

1. Enumerate the nodes of G in DFS finish order, starting from any node
2. Compute the transpose G T (that is, reverse all edges)
3. Make a DFS in G T, considering nodes in reverse finish order from original DFS
4. Each tree in this depth-first forest is a strongly connected component

2.9 Important syntax

2.9.1 Let

```
  let x = 1 in x * 2 == 2
  case x of
  1 -> "Hello"
  2 -> "H"
  3 -> "Hel"

  input: 3 == "Hel"

  # other examples
  let f x y = x + 3 >= y + 3.1 in f 1 1 == False

  f x = let g z = z+1 in g (g x)
  f 1 == 3

:t div
div :: Integral a => a -> a -> a

:t (/)
(/) :: Fractional a => a -> a -> a
```

2.9.2 IO

monads Is used to return an IO and uses a operation *(»=)* that replaces *do-notation*

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

2.10 Sorting Algorithms

1. Insertion Sort: One at a time
2. Bubble Sort: sort in order two a time starting from left and work to the right side.
3. Merge Sort: Divide and Conquer, split into even pies and sort them and then merge/sort again.
4. Quicksort: Divide and Conquer, takes a pivot to spit smaller and larger.