Sammanfattning
TDDC30

Variabler

- Static
  - Istället för att varje ny instans av en klass ska allokera ett nytt minne för en variabel så delas de över hela klassen.

- Final
  - Gör att man endast kan instansiera en variabel 1 gång. Instansieringen lägger man lämpligast i konstruktor. Ligger den i metoden själv kan den kallas på flera gånger och finalvariabeln kan ändras flera gånger, vilket ej är tillåtet.

Metoder

- Instansmetod
  - Är en metod är en metod som inte är <u>static</u>.

- Klassmetod
  - Är en metod som är likadan för alla instanser av en klass.

  Bra tumregel: Kommer jag kalla på metoden utan att ett objekt har blivit instansierat än?

  T.ex för en bil är mphToKm().

Synlighet

- Public
  - Synlig för alla
- Protected
  - Synlig inom paketet och subklasser
- Private
  - Synlig inom klassen

God sed synlighet

Klasser - Som regel <u>public</u>

Interna hjälpmetoder - private

Övriga metoder - public

Attribut/var - private

Arv

- Om någonting "extends" en annan klass så får den automatiskt alla metoder som klassen får. Den kan även overridea metoder som finns i huvudklassen

- Abstract

  Om man lägger till abstract i en metod så måste alla subklasser implimentera den klassen i sin egen klass. Skrivs utan body:

  public abstract void printString();

  Detta så vida inte subklassen själv är abstract, då behöver den det inte.

- En subklass ärver från en superklass
- För att komma åt superklassen, använd "super"
- Använd final för att förbjuda subklassning eller override.

UML - Unified Modelling Language

- Grafiskt språk för att beskriva objektorienterade system.
- Industristandard idag

Klassdiagram

public	+
private	-
protected	#
package private	~
static	understruken
abstract	*Italic*
derived	/

T.ex

Car
- model - registryNumber - owner
+ startEngines() + shutOfEngines() + driveForward() + reverse() + break

- Klassnamn
- Attribut
- Metoder

Relation mellan klasser (UML)

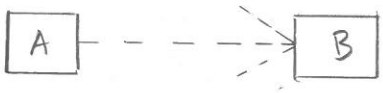
- Aggregation - A använder en B

- Komposition - A "äger en" B

- Generalisering (arv) - A ärver från B
  (A kan användas som en B)

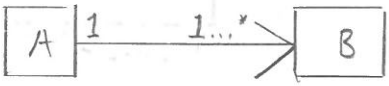
- Realisering (interface) - A implementerar B

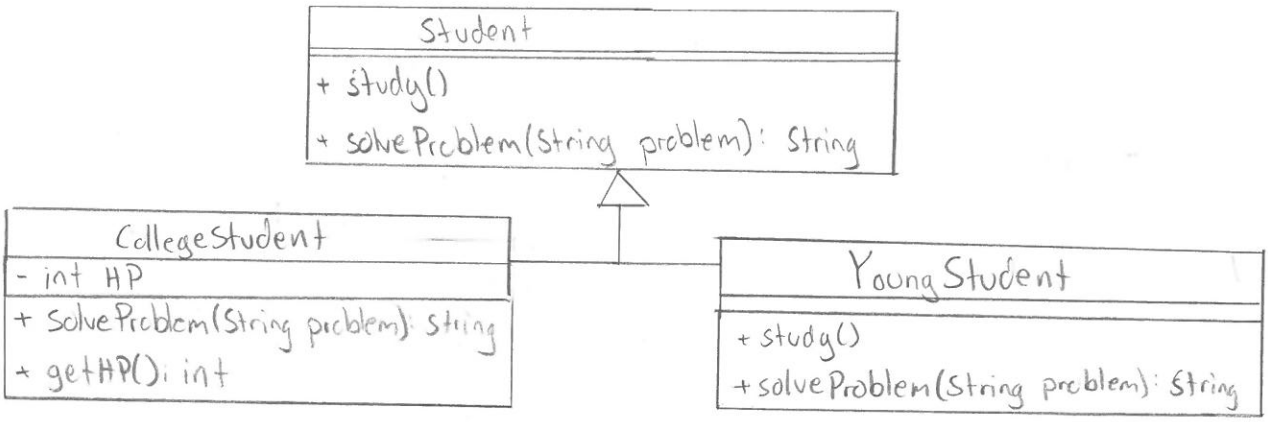
- Beroende - A är beroende av B



Exempel:

En A använder en eller flera B



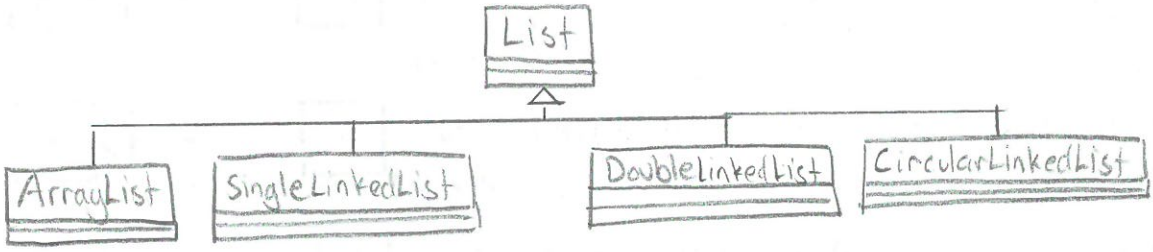


- En metod kan ta in alla subklasser till en superklass genom att man skriver superklassen som parameter.

Abstrakta datatyper

- En datatyp där man bortser från implementationsspecifika egenskaper och istället uppmärksammar en eller några få egenskaper.

Exempel: En lista



Referenser

- Variabler av icke primitiva datatyper är <u>alltid</u> referenser i Java.
- Måste instansieras innan de används.
- Två variabler kan referera till samma minnesutrymme

Iteratorer

- Sätt att iterera över alla element i en datorstruktur utan att behöva veta om implementationen

Exempel:

```
List<String> mylist = new List<String>();
Iterator<String> it = mylist.iterator();
```

Krav

```
hasNext();
next();
remove();
```

ADT

~~ATD~~ - Abstrakta DataTyper

- Kö (Queue) - Först in först ut (FIFO)

  front()/peek()
  - Returnerar främsta elementet

  dequeue()
  - Tar bort och returnerar det främsta elementet

  void enqueue()
  - Lägg till ett element i slutet av kön

  boolean isEmpty()
  - Returnerar sant om kön är tom

  int size()
  - Returnerar antalet element i kön

Stack - Sist in först ut (LIFO)

top()/peek()

- Returnerar stackens toppelement utan att ta bort det

pop()

- Tar bort och returnerar stackens toppelement

void push(Element)

- Lägger till ett element på toppen av stacken

boolean isEmpty()

- Returnar sant om stacken är tom

int size()

- Returnerar antalet element i stacken

Interface

Fungerar som abstract klass med enbart abstrakta metoder. Dvs den subklass som implementerar ett interface måste skapa samtliga metoder.

```
public interface Dictionary {
    public boolean lookup(String key);
}

public class MyDictionary implements Dictionary {

}
```

- Samtliga metoder blir public abstract
- Samtliga variabler blir public static final (konstanter)
- Det går att implementera flera interface i samma klass

Generiska klasser

- En mall där utseendet först definieras när klassen används.

```
public class MyListNode<E>{
    E data;
    MyListNode<E> next;
}
⋮
MyListNode<String> newlist = new MyListNode<String>
```

Nu blir E en String!

- Det går även att begränsa hur generisk en klass ska vara genom <u>extends</u>.

```
public class MyListNode<E extends Animal>{
... }
```

Cat & Dog är subklasser till Animal och kan därför skapas.

```
MyListNode<Cat> cat = new MyListNode<Cat>;
MyListNode<Dog> dog = new MyListNode<Dog>;
```

Undantag

- Skrivs med i metodtiteln. Antingen skriver man en grupp av exceptions vilket gör att att alla exceptions i gruppen kan kastas, eller så skrivs det exakta exceptionet ut.

- Om en metod kastar ett exception skriver man:

```
public boolean lookup(String pnr) throws PnrNotFoundException{
  for(....){

  }
  throw new PnrNotFoundException(
```

- För att fånga ett exception används try and catch:

```
try{
   erikskia.drive();
}catch(NoFuelException e){
   erikskia.reFill();
}catch(EngineOverHeatedException e){
   erikskia.Stop();
   throw e;
}finally{
   erikskia.getOutOfTheCar();
}
```

Körs oavsett om undantag uppstått eller inte

- Förbjudet:

```
try {
    erikskia.drive();
}catch(Exception e) {

}
```

Fångar alla exceptions. Fånga bara de du förväntar dig!

Och sen låtsas som att ingenting har hänt

Ordo / Tidskomplexitet

- Bunkra alltid in alla andra $\mathcal{O}()$ i den största!

T.ex: $\mathcal{O}(1 + 5n + 5n^2) + \mathcal{O}(n^7) = \mathcal{O}(n^7)$

Konstant $\rightarrow$ $\mathcal{O}\left(\frac{1}{\log_b(a)}\right) * \mathcal{O}(\log_b(n)) = \mathcal{O}(\log_b(n))$

Ordo	Benämning
$\mathcal{O}(1)$	Konstant
$\mathcal{O}(\log(n))$	Logaritmisk
$\mathcal{O}(n)$	Linjär
$\mathcal{O}(n\log(n))$	
$\mathcal{O}(n^2)$	Kvadratisk
$\mathcal{O}(n^3)$	Kubisk
$\mathcal{O}(2^n)$	Exponentiell
$\mathcal{O}(n!)$	Faktoriell
$\mathcal{O}(n^x)$	Polynomisk

Effektivare

Högre Komplexitet

- Oftast värsta fallet som analyseras

Sorteringsalgoritmer

- Insertionsort
  - Delar in arrayen i en sorterad & en osorterad del
  - Lägger successivt in varje osorterat element där det ska in.

Algoritm

1) Tur ut första elementet i osorterad lista
2) Skjut på de element som är större ett steg uppåt
3) Sätt in element på tom plats

Fakta

Tidskomplexitet: $\Theta(n^2)$
Stabil? Ja

- Selectionsort

Algoritm

1) Dela upp arrayen i en sorterad ah en osorterad del. Första elementet är sorterad i början.

2) Leta upp minsta elementet i osorterade delen och lägg in i den sorterade

3) Upprepa

Fakta

Tidskomplexitet: $O(n^2)$
Stabil? Nej

Den är rätt långsam, men har få byten.

- **Shellsort**

"Upprepad" insertionsort

Algoritm

1) Markera vart N:te element från och med n i listan och sortera dessa inbördes med insertionsort

2) Öka n med ett och upprepa (1) tills n == N

3) Minska N, låt n vara 0 och upprepa (1), (2) tills N < 1

Notering

- Börja med ett stort N, upp till size/2
- När N == 1 i slutet körs en vanlig insertionsort.

Val av lucksekvenser

- Hibbards: $[1, 3, 7, \ldots, 2^k - 1]$
- Knuths: $[1, 4, 13, \ldots, (3^k - 1)/2]$
- Gonnets: Dividera N stegvis med 2.2. Avrunda nedåt. Se till att sista blir 1.

Fakta

Tidskomplexitet:
Beror på lucksekvens.
Ej sämre än $O(n^2)$

Stabil? Nej

- Quicksort

"Divide and Conquer"

Algoritm

1) Välj ut ett pivotelement och swapa sist i listan

2) Placera alla element <p till vänster i fältet och alla element > p till höger.

3) Flytta in pivot där de två sidorna möts

4) Upprepa (Rekursivt) på den högra och vänstra delen som uppstår.

Fakta

- Tidskomplexitet: $\underline{O(n^2)}$
  Men medelfallet är nästan lika med bästa fallet ($\Omega(n\log(n))$)

- Stabil? Nej (Inte i sin grund)

- Fördel: Jämför tal nära varandra.
  Bra när datamängden är mycket stor.

Val av pivotelement

Ju bättre val desto jämnare uppdelning

1) Elementet längst till höger (ligger redan "åt sidan")
2) Slumpa ett element
3) Medianen av 3 element (Stort chans att det blir lagom).

• Bubblesort

Algoritm

1) Börja från höger
2) Jämför två bredvidliggande element och swapa om den vänstra är större än den högra
3) Upprepa tills hela listan har gåtts igenom
4) Upprepa (1), (2) och (3) tills listan är sorterad.

Fakta

Tidskomplexitet: $O(n^2)$

Stabil? Ja

- Shakersort (Cocktail shakersort)

  Tvåvägs Bubblesort

  Algoritm

  1) Kör Bubblesort höger till vänster
  2) Kör Bubblesort vänster till höger
  3) Upprepa tills sorterad

  Fakta

  Tidskomplexitet: $O(n^2)$
  Stabil? Ja

- Mergesort

"Devide and Conquer"

Tag två sorterade delmängder och sortera ihop dessa.

Algoritm

1) Dela upp mängden tills varje delmängd består av 2 element (eller 3 om det är ojämnt)

2) Jämför dessa element och swapa så minst är till vänster. Gör detta för alla delmängder.

3) Jämför nu de två sorterade delmängderna som ligger bredvid varandra med varandra

4) Upprepa tills vi har ett sorterat element kvar

Fakta

Tidskomplexitet: $O(n\log(n))$

Stabil? Ja (om sammanställningsalgoritmen är väl vald)

Inmatning

Vanliga exempel

Ta in från tangentbordet

```
BufferdReader br = new BufferdReader
(new InputStreamReader(System.in));
String s = br.readLine();
```

eller

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

- Scanner är en klass som hjälper oss med inmatning.

Countsort

Sorteringsalgoritm för tal

Algoritm

1) Dimensionera en array efter värdet på det största elementet (exklusive 0)

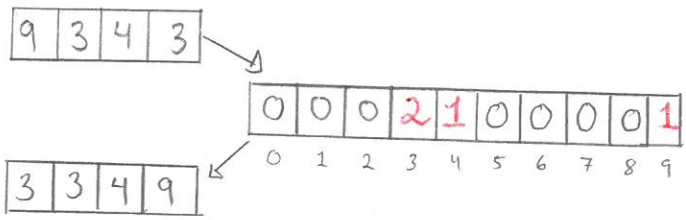
   Om största värdet är 9 får vi storleken 9 + 1 = 10 på arrayen.

2) Sätt alla startvärden till 0.

3) Gå igenom den osorterade listan. För varje påträffat tal så ökar du upp dess värde i count-arrayen.

4) Gå sedan igenom count-arrayen och lägg in talen i en ny sorterad array.
   Har index 4 (talet 3) värdet 2 så finns det 2st 3or.

Exempel



Fakta

Tidskomplexitet: $O(n)$

Stabil? Ja!

Kommentar: Den är bra för mindre begränsade intervall.
Snabb pga att det inte är en jämförelsealgoritm.

- <u>Bucketsort (Bin sort)</u>

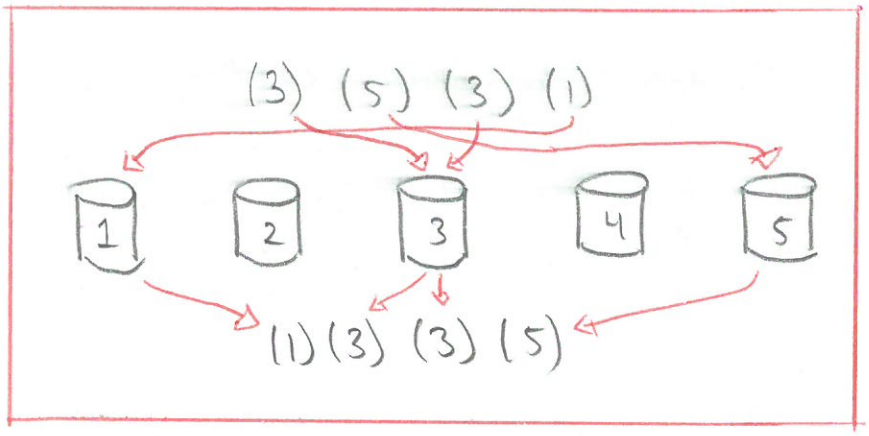
Likt countsort.

Algoritm

1) Skapa likamånga hinkar (buckets) som värdet på det största elementet (+1 om 0 är med)

2) Lägg varje element i en överrensstämmande hink.

3) Börja med hink 1 (index 0) och ta ut värdena & lägg dem bredvid varandra.



Fakta

Tidskomplexitet: Beror på implementation

Stabil? Beror på implementation

- ## Radix sort

Sorterar data med flera delnycklar lexiografiskt.

Algoritm

1) Sortera varje delnyckel med en stabil Bucketsort.

Fakta

Tidskomplexitet: $O(n)$
Stabil? Ja!

Sammanfattning sorteringsalgoritmer

Allmänt

- Det finns både $O(n^2)$ och $O(n\log(n))$
- $O(n^2)$ kan vara bra ibland.

Selectionsort

- Sök minsta elementet i osorterade och lägg i slutet av sorterade.
- Alltid $O(n^2)$ = Alltid långsam
- Fördel att det endast är n byten (i värsta fall)
- Instabil

- Insertionsort
  - Dela upp i en sorterad och en osorterad del.
  - Titta på elementet i den osorterade som är närmast den sorterade och placera in på rätt plats
  - Bästa fall $O(n)$, värsta fall $O(n^2)$
  - Bra på små arrays (<50) där datat nästan är sorterat.
  - Lätt att implementera
  - Stabil!

- Shellsort
  - Börjar med element långt från varandra och nyttjar upprepad insertionsort. Därefter minskas avståndet successivt.
  - Svårt att analysera tidskomplexitet, beror på vald sekvens

    $O(n^2)$, $O(n^{1,5})$, $O(n^{1,3})$, $O(n\log(n))$
  - Instabil!

- Mergesort
  - Ta två delmängder och sortera ihop dessa
  - Optimal algoritm - $O(n\log(n))$
  - Kräver extra minne - $O(n)$
  - Ideal för sortering av internminne
  - Stabil (om bra vald sammanställningsalgoritm)

- Quicksort
  - Välj ett pivotelement och gå från höger och vänster och jämför/swap. När höger & vänster möts har vi två nya delar. Gör om detta på dem. osv.
  - I medelfallet den snabbaste sorteringsalgoritmen $O(n\log(n))$
  - $O(n^2)$ i värsta fallet. Används ej i realtidsapplikationer.
  - Instabil om man implementerar en effektiv algoritm.

- Heapsort
  - Nyttjar en heap för sortering.
  - Mycket snabb - $O(n\log(n))$
  - I praktiken långsammare än Quicksort i medelfallet.
  - Bäst i värstafallet
  - Bra för realtidsapplikationer

- Countsort / Bucketsort
  - Tittar hur många gånger ett tal finns / sorterar genom lådor.
  - "Ej jämförande" algoritm
  - Bra om nyckelmängden är liten och kan avbildas i heltal
  - Snabbare än både Quicksort & Heapsort

- Radixsort
  - Tar hänsyn till lexiografi
  - Bra om det finns flera delnycklar

Urval

Om man tex ska välja ut den anställda som har 3:e högst lön.

- Quickselect
  - Väldigt lik Quicksort

Algoritm

1) Välj ett pivotelement
2) Välj en av partionerna beroende på sökt värde
3) upprepa

- Går att implementera både in-place och out-of-place / not-in-place.
- Viktigt att välja bra pivotelement

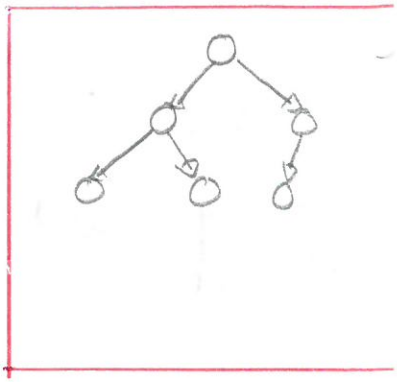
Fakta

Tidskomplexitet

- Värsta fall: $O(n^2)$
- Bästa fall: $O(n)$

Träd

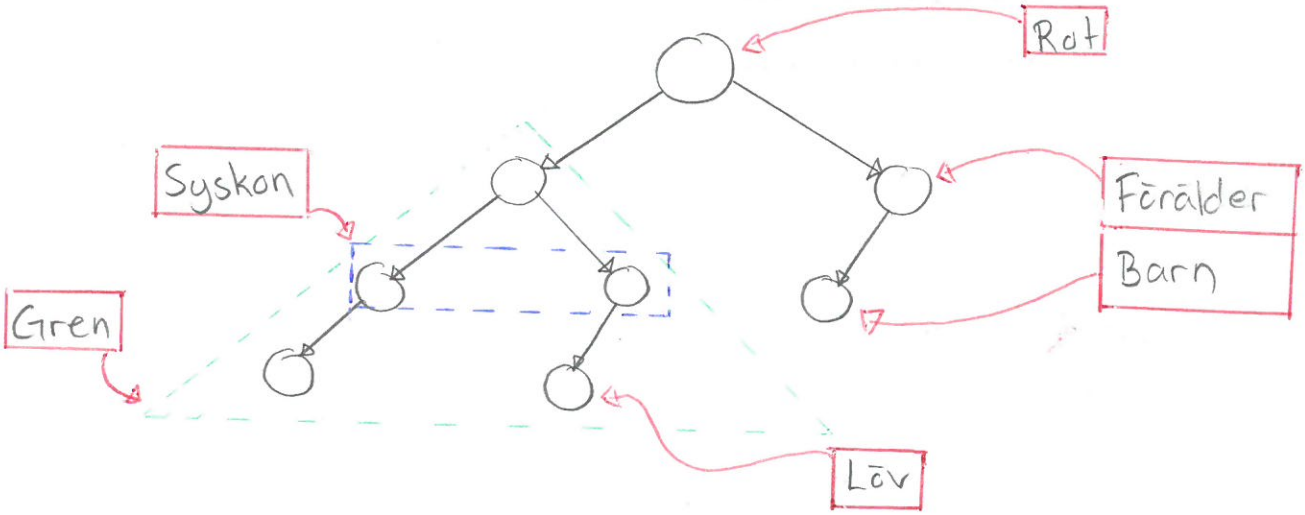
- Är en riktad acyklisk graf (DAG)
  - Riktag graf utan cykler



- Det finns en väg till varje nod
- Bra för snabb sökning

Definition

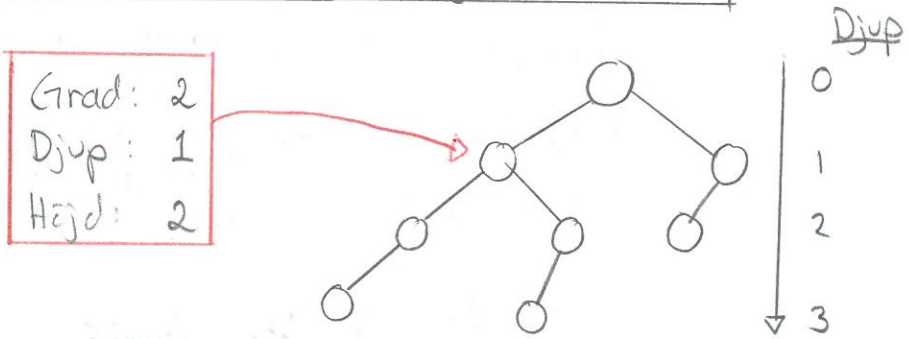
Förälder	Nod med minst 1 barn
Syskon	Noder med samma förälder
Rot	Nod utan förälder
Löv	Nod utan barn
Gren/delträd	Samling noder med gemensam anfader



- Alla träd definieras rekursivt!

- Nodinformation

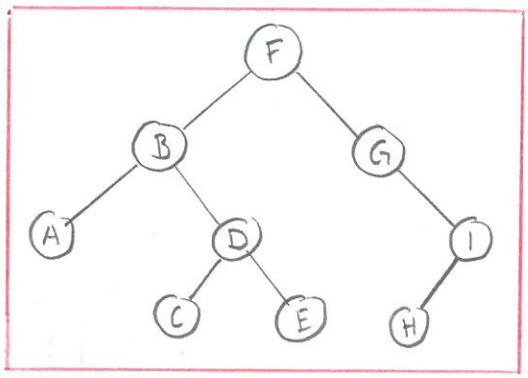
Grad	Antalet barn noden har
Djup	Avstånd från roten
Höjd	Avstånd till lövet längst bort, nedåt



- Trädets höjd = Rotens höjd

- Binärt träd är träd med grad ≤ 2

Traversering

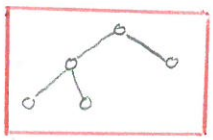


- Preorder (F, B, A, D, C, E, G, I, H)
  - Börja i roten, gå igenom hela vänstra grenen, sen högra.
  - Rot, vänster, höger
- Inorder (A, B, C, D, E, F, G, H, I)
  - Vänster gren gås igenom nedifrån, därefter noden och höger gren
  - Vänster, rot, höger
- Postorder (A, C, E, D, B, H, I, G, F)
  - Vänstra gås igenom nedifrån och upp, sedan högra. Sist roten
  - Vänster, höger, rot
- Levelorder (F, B, G, A, D, I, C, E, H)
  - Ta roten, gå ned ett steg och ta alla syskon från vänster till höger.

- Några fler termer

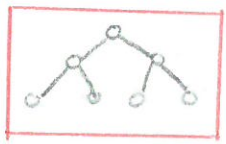
  - Fullt binärt träd

    Samtliga noder har 0 eller 2 barn



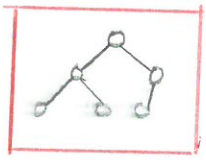
  - Perfekt träd

    Ett fullt träd med alla löv på samma djup



  - Fullständigt binärt träd

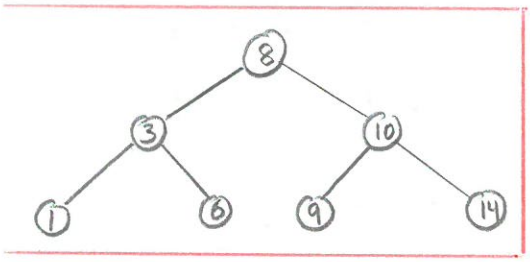
    Ett perfekt träd med skillnaden att den får saknna några av de "högraste" löven längst ned



- Binärt sökträd

  Typ av binärt träd

  - För varje nod gäller:



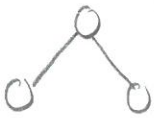
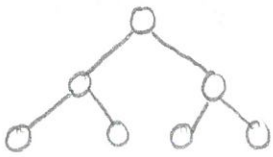
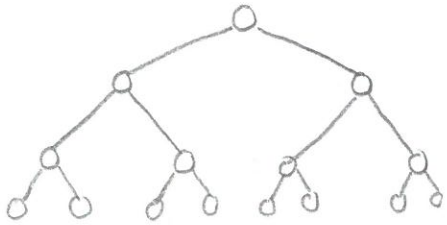
    - Alla element till vänster är mindre än nodens värde
    - Alla element till höger är större än nodens värde.

Nackdel

Kan ej lagra två element med samma värde

- **Hur snabbt går sökningen?**

  Det går snabbt. I balanserade träd går det på $O(\log(n))$.

n = 3	n = 7	n = 15
		
max 1 "steg"	max 2 "steg"	max 3 "steg"

- **Addering i binära sökträd**

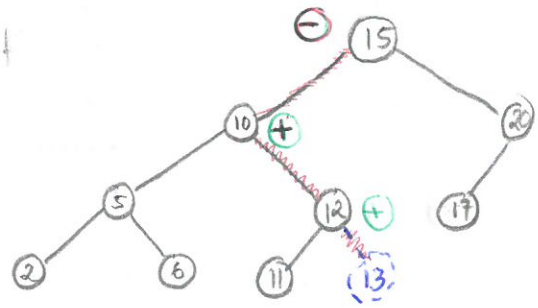
  1) Är värdet mindre än noden?
     => Vänster

  2) Är värdet större än noden?
     => Höger

  3) Finns ej noden?
     => Rätt plats funnen!

Exempel:

Sätt in 13 i trädet



- Borttagning i binära sökträd

Algoritm

1) Sök efter rätt nod på samma sätt som vid addition

2) Sök en ersättare

   a) Om bara ett barn finns, välj det

   b) Annars välj noden längst till vänster i det högra trädet.

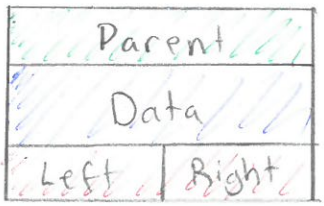
3) Koppla loss ersättaren, ersätt den med dess högra deltråd under sig själv.

4) Sätt in ersättaren på dess nya plats.

- Representation av binära sökträd

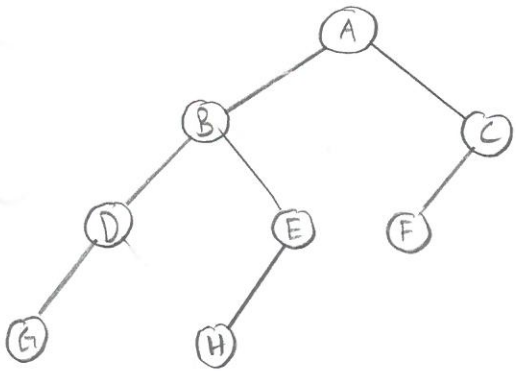
Brukar vanligtvis implementeras som länkade noder.

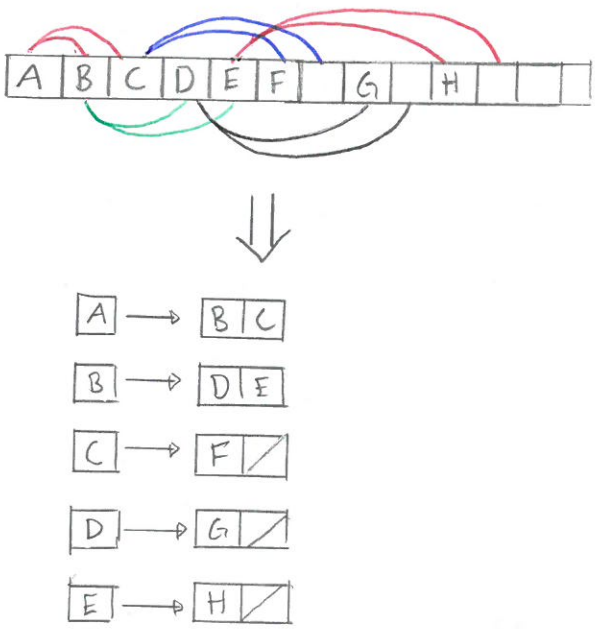
```
class TreeNode {
    TreeNode parent;
    Datatyp data;
    TreeNode Left;
    TreeNode Right;
}
```



- Rekursiva algoritmer är lämpliga då delträd också är träd.

- Representera binära sökträd som array





- Indexering av noder

Root	0
Vänster barn	$2*i+1$
Höger barn	$2*i+2$
Förälder	$(i-1)/2$ (Avrunda nedåt)

i = Förälderns index

i = Eget index

ADT Prioritetskö

En prioritetskö är en samling "nycke-värde" par med följande metoder:

- insert(prioritet, värde)

  Associerar en nyckel med ett värde

- removeMin();

  Tar bort och returnerar elementet med högst prioritet.

- Nycklen kan vara vad som helst:
  - Tal
  - Datum
  - Bokstäver

- Mindre värde på nycklen => Högre prioritet

- Antingen gör man en egen, eller så kan man använda Javas inbyggda PriorityQueue.

2 sätt att göra det på

1	`PriorityQueue<Song> pQ = new PriorityQueue(new SongComparator());`

2	`PriorityQueue<Song> pQ = new PriorityQueue<Song>();`
	⇓
	`class Song implements Comparable {`
	`....`
	`}`

- För att använda Object som nycklar <u>måste</u> interfacet <u>Comparable</u> implementeras

- Alternativt att skapa en motsvarande "SongComparator"

- Går även att sortera med en prioritetskö:
  1) Skapa en prioritetskö & addera in alla objekten.
  2) Skapa en ny sortedList och loopa igenom hela kön & lägg över dem i prio-ordning i den nya listan. Ta sedan bort prio-listan.

- Implementation

Finns två sätt att implementera:

1) Osorterad lista.
   - Blir som en selection sort när vi väljer ut.
   - $O(n)$

2) Sorterad lista
   - Blir som en insertionsort när vi ska lägga in
   - $O(n)$

- Vilken väljer man?

1) Osorterad lista OM många insättningsoperationer jämfört med remove();

2) Sorterad lista OM många remove(); jämfört med insättningar (?)

Heap - $O(\log(n))$

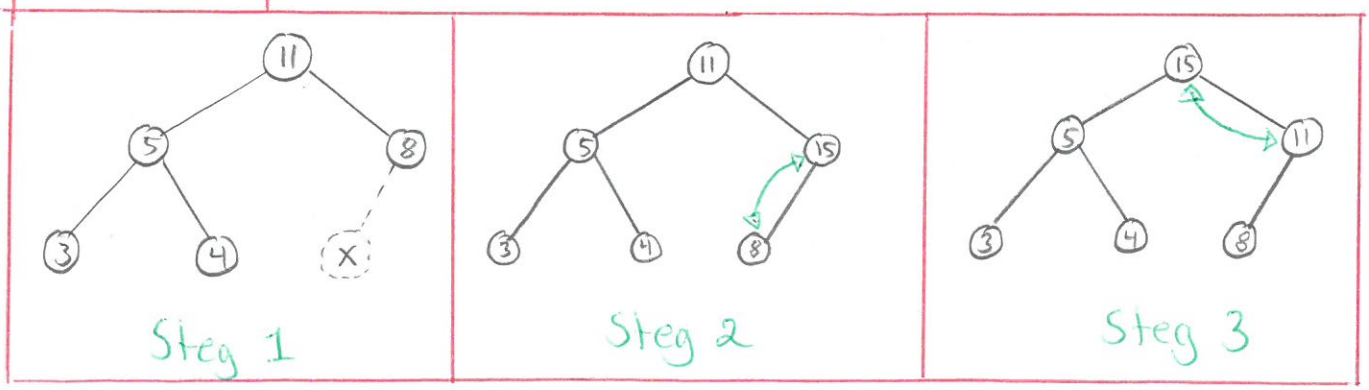
- Max-heap
  1) Varje nods värde är större eller lika med värdena i nodens barn
  2) Trädets grenar ska vara så jämna som möjligt i antal. Om det inte går fylls den nedersta nivån på från vänster.

- Min-heap
  1) Varje nods värde är mindre eller lika med värdena i nodens barn
  2) Likadan som för max-heap.

- Insättning i Max-heap
  1) Lägg in noden sist i heapen
  2) Om noden är större än sin förälder => swap
  3) Upprepa (2) tills föräldern är större

(Fungerar likadant för min-heaps, fast du då tittar om föräldern är mindre eller inte)

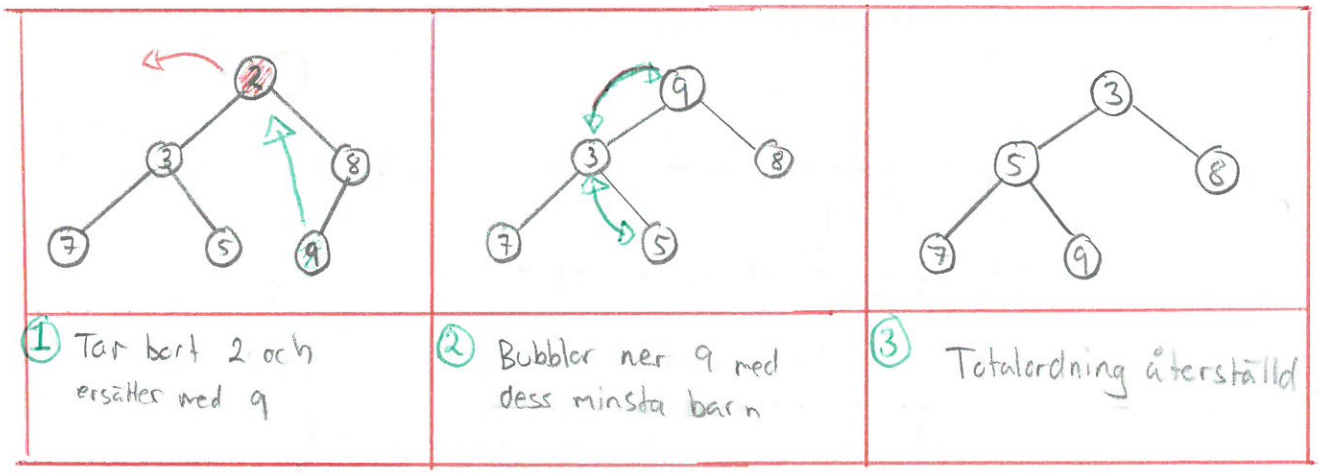
Sätt in X=15



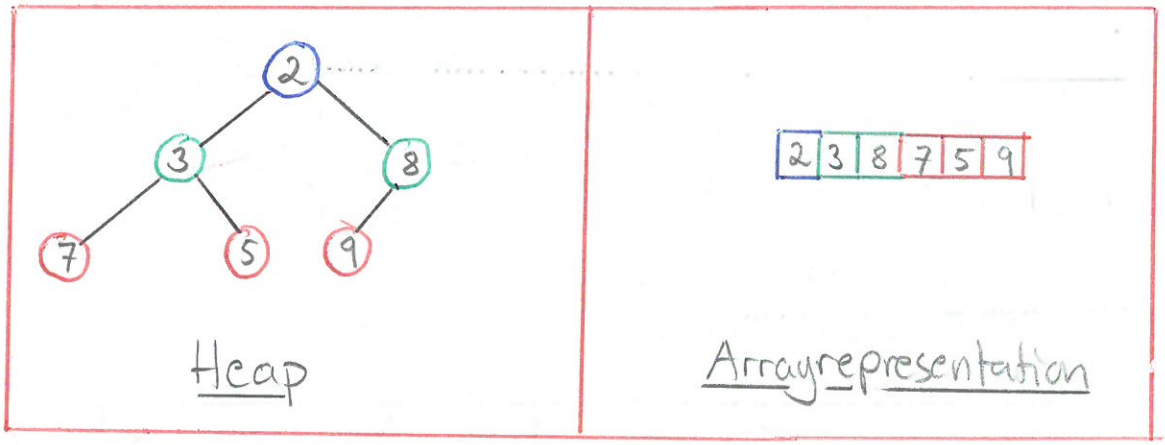
- Borttagning ur Min-heap

1) Ta bort root-noden och ersätt den med den sista noden.

2) Jämför och swapa med nodens minsta barn. Gör detta tills totalordningen är återställd



- Arrayrepresentation av Heap



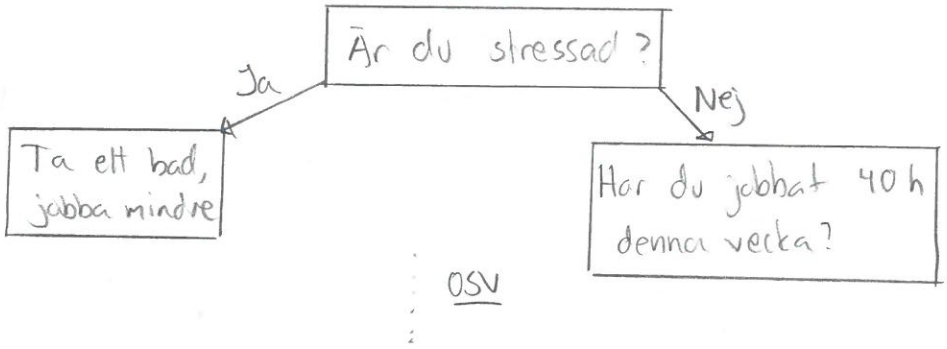
- Heapen är "medelbra" på både insert och remove.

Insert() - $O(\log(n))$

Remove() - $O(\log(n))$

Beslutsträd

Fungerar som ett binärt träd där löven är utfall



- Hälften av alternativen kan sållas bort vid varje val.
- För n utfall blir trädet som kortast log(n) i höjd.
- Antalet utfall = Antalet permutationer!

n = 2	n = 4	n = 5
(a,b)	(a,b,c,d)	(a,b,c,d,e)
ab ba	abcd bacd ⋮	abcde bacde ⋮
Tot: 2st	Tot: 24st	Tot: 120st

- Sortering som beslutsträd

  n element att sortera
  => n! permutationer

Sammansatta nycklar

Nycklar behöver inte vara enskilda heltal, kan bestå av flera delnycklar:

Punkter i plan	(x, y)	2st
Datum	år, månad, dag	3st
Strängar	Varje enskild char	Beror på längd

- Lexiografisk ordning

  Olika delnycklar har olika signifikans
  - Först gäller första delnyckeln
  - Sen gäller andra delnyckeln osv.

  Exempel på detta är en alfabetisk ordlista där första bokstaven har prio 1, andra prio 2 osv.

Grafik

Fönster ser olika ut på olika operativsystem. Detta tar swing hand om.

- Swing

  Ett uniformt sätt att skapa GUI

Swing

JFrame:

Används för att få upp ett fönster

JButton:

Knapp

JList:

Lista

- Grapics

  Ses som en pensel associerad till en viss yta.

  - setColor, setPen, setBrush, setFont
  - drawLine
  - drawRect, fillRect
  - drawOval, fillOval
  - drawImage
  - drawString

- JComponent
  - Dimension getSize()
    Ger komponentens nuvarande storlek
  - void setPerferredSize(Dimension dim)
    Sätt komponentens önskade storlek
  - void repaint()
    Be om utritning

Komplexitet

- Rumskomplexitet

  Mäter hur mycket extra minne algoritmen allokerar förutom indatat.

  - int[] tmp = new int[1000000] => $O(1)$
  - int[] tmp = new int[n.length*2] => $O(n)$
  - int[] tmp = new int[n.length * n.length] => $O(n^2)$