Algoritmer och Datastrukturer

Föreläsning 7

Binära träd och grafer

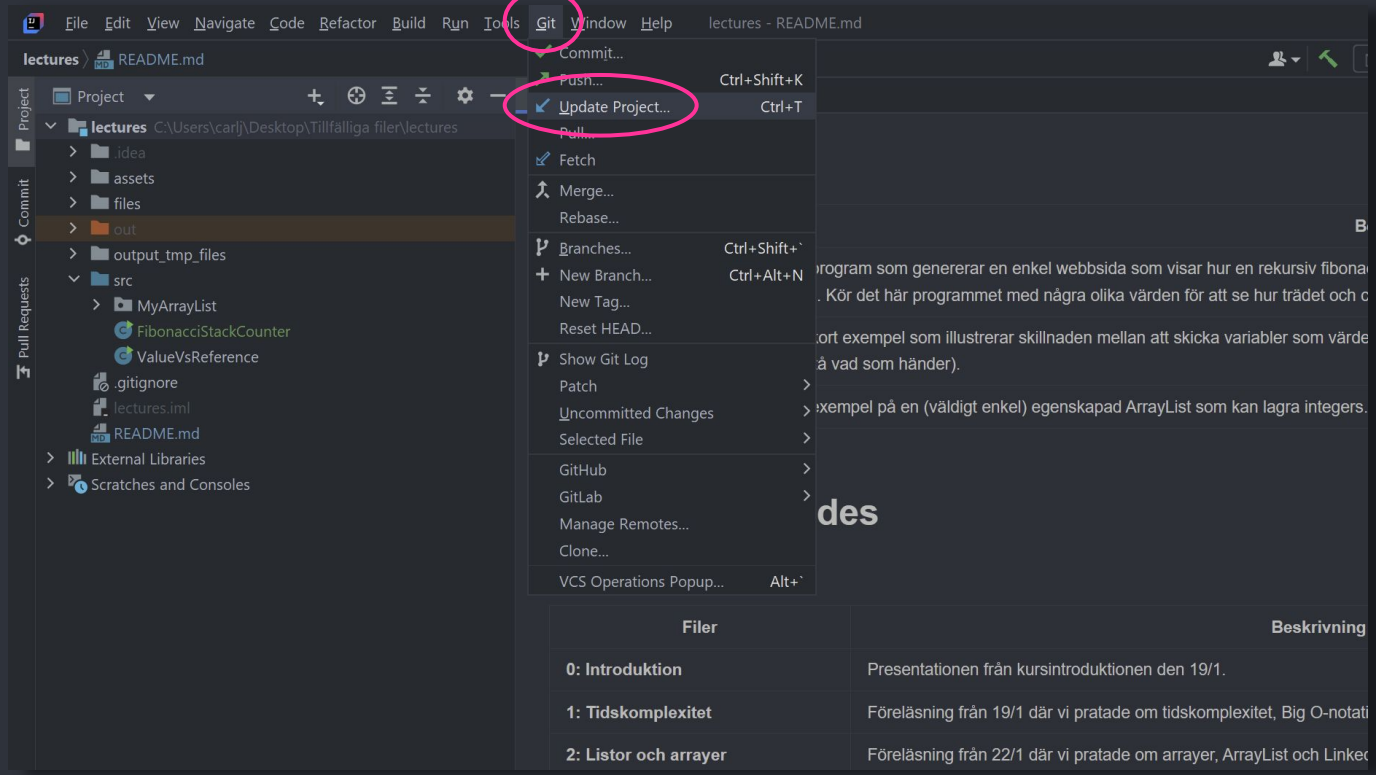
I dag ska vi:

- Studera **binära sökträd**
- Prata lite mer om komplexiteten **O(logn)**
- Lära oss skillnaden mellan **In-**, **Pre-** och **Postorder**
- Fortsätta prata om rekursivitet
- Kolla på **grafer** och hur de fungerar
- Förstå skillnaden mellan **djupet först och bredden först**
- Prata lite om **heapen och garbage collection**

Den här veckan bör ni:

- Göra färdigt **Labb 3 och Labb 4**
- Göra **Dugga 1** (stänger den 15 februari)
- Göra **uppgifterna** under modulen “Vecka 7: Träd och Grafer”
- **Läsa kapitel 8, 11 och 14 i kursboken** (eller de kapitel som handlar om motsvarande saker i Grokking Algorithms)

https://github.com/carljohanj/algo



Lite repetition!

Comparator: att jämföra objekt

```java
public Person findPerson(Path file, Comparator<Person> sortBy) throws IOException
{
    Person target = null;

    for (String line : Files.readAllLines(file))
    {
       Person current = parsePerson(line);

       if (target == null || sortBy.compare(current, target) < 0)
       {
          target = current;
       }
    }

    return target;
}
```

- Vi tittade på Comparator senast: vi kan även spara en Comparator i en variabel och sedan skicka den som argument till en metod!
- Vi kan då skapa sortering av t.ex. filer utifrån användar-behov

```java
public static void main(String[] args)
{
   Comparator<Person> byAge = Comparator.comparing(Person::getAge);
   Comparator<Person> byWages = Comparator.comparing(Person::getSalary);

   Person youngest = findPerson(file, byAge);
   Person bestPaid = findPerson(file, byWages);
}
```

Förstå kod: rekursivitet

Fråga: Vad blir utskriften av anropet beast(666)? **(Tentafråga, VT22)**

```
public static void beast(int n)
{
      if (n == 0)
      {
         return;
      }
      else
      {
         beast(n / 2);
      }

      if (n % 2 > 0)
      {
         System.out.println(1);
      }
      else
      {
         System.out.println(0);
      }
}
```

- Behöver kunna 4 saker för att lösa uppgiften:
  - Förstå rekursivitet
  - Förstå modulo
  - Förstå heltalsdivision
  - Förstå att void-metoder kan returnera tomt (så länge de inte försöker returnera värden: de kan t.ex. inte returnera null!)
- En tom return-sats betyder bara att metodanropet avslutas i förtid

Modulo: resten från heltalsdivision

- Modulo är **resten som blir över vid en division av två heltal**

  Betecknas %

  18 % 7 = 4, eftersom 7 går in jämnt i 18 två gånger (7 * 2 = 14) och vi får 4 som rest

- Operator som är bra att ha koll på eftersom **ett heltal modulo 2 bara kan resultera i ett av två värden: 1 eller 0**

- Att dela valfritt heltal med två ger oss ju antingen en rest, eller ingen rest:

  2 % 2 = 0
  3 % 2 = 1
  4 % 2 = 0
  5 % 2 = 1
  6 % 3 = 0
  osv

Lösning för uppgiften

```
public static void beast(int n)
{
      if (n == 0)
      {
         return;
      }
      else
      {
         beast(n / 2);
      }

      if (n % 2 > 0)
      {
         System.out.println(1);
      }
      else
      {
         System.out.println(0);
      }
}
```

1) beast(666) anropar beast(333)

2) beast(333) anropar beast(166) (333/2 = 166.5, men decimalen ignoreras pga heltalsdivision)

3) beast(166) anropar beast(83)

4) beast(83) anropar beast(41) (decimalen ignoreras)

5) beast(41) anropar beast(20) (decimalen ignoreras)

6) beast(20) anropar beast(10)

7) beast(10) anropar beast(5)

8) beast(5) anropar beast(2) (decimalen ignoreras)

9) beast(2) anropar beast(1)

10) beast(1) anropar beast(0) (decimalen ignoreras)

11) beast(0) exekverar en tom return, och de tidigare metodanropen kan nu gå vidare

Lösning för uppgiften

Metoden:

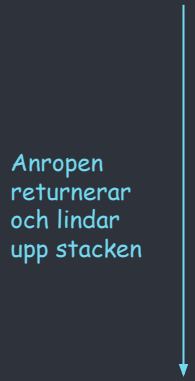
```
public static void beast(int n)
{
      if (n == 0)
      {
         return;
      }
      else
      {
         beast(n / 2);
      }

      if (n % 2 > 0)
      {
         System.out.println(1);
      }
      else
      {
         System.out.println(0);
      }
}
```

Varje metod på stacken har nu enbart den här if-satsen kvar att utföra:

```
if (n % 2 > 0)
{
   System.out.println(1);
}
else
{
   System.out.println(0);
}
```

De har ju redan konstaterat att de inte nådde basfallet (n == 0) och i stället gjort rekursiva anrop. Nu fortsätter de, en efter en, att köras färdigt för att därefter poppas från stacken så att nästa anrop på stacken kan gå vidare, osv.

Lösning för uppgiften



Metodanrop:	Beräkning:	System.out.println():
beast(1)	1 % 2 = 1	1
beast(2)	2 % 2 = 0	0
beast(5)	5 % 2 = 1	1
beast(10)	10 % 2 = 0	0
beast(20)	20 % 2 = 0	0
beast(41)	41 % 2 = 1	1
beast(83)	83 % 2 = 1	1
beast(166)	166 % 2 = 0	0
beast(333)	333 % 2 = 1	1
beast(666)	666 % 2 = 0	0

Svar: Om metoden anropas med beast(666) kommer den att skriva ut: **1010011010**

Det här är för övrigt det binära tal som motsvarar 666!

Vilken tidskomplexitet har koden?

- **O(logn)** eftersom det är ett **exempel på divide-and-conquer:** den halverar n för varje nytt rekursivt anrop
- Eftersom det **bara är ett värde** vi beräknar blir det O(logn)
- **Om vi däremot hade loopat genom en lista** med nummer och frågat: “Vad returnerar beast() när den anropas med vart och ett av de här numren?” skulle det varit O(n*logn): vi skulle då ha anropat beast() n antal gånger

Lite mer om O(logn)

- Eftersom logaritmisk tillväxt är så oerhört liten är det i praktiken **omöjligt att se skillnad** på den och **konstant tid** när vi mäter exekveringstid

- Även när n ökar en miljard gånger går en **divide-and-conquer-algoritm** (t.ex. binär sökning) bara från 3 till 30 operationer för att hämta ut ett värde:

Datamängd n	Array O(1)	Träd O(logn)	Länkad Lista O(n)
10	1	3	10
100	1	6	100
1000	1	10	1000
1000.000	1	20	1000.000
1000.000.000	1	30	1000.000.000

- Vi ska prata mer om datastrukturer som har O(logn) för sökning, såsom träd, under dagen

O(n*logn) är på samma vis väldigt nära O(n)

- Folk ritar ofta kurvan för **O(n*logn)** som halvvägs mellan **O(n)** och **O(n^2),** men det ger en felaktig bild av komplexiteten
- Ta en algoritm med en miljard n som indata som exempel: om tidskomplexiteten är **O(n*logn)** kommer den att utföra **~30 miljarder operationer,** eller 30*n (eftersom $\log_2(10^9) = 30$)
- En algoritm med tidskomplexiteten **O(n^2)** kommer däremot att utföra **en miljard miljard operationer**
- Om vi uppskattar att en dator kan utföra ungefär en miljard operationer per sekund i genomsnitt kommer **O(n*logn)-algoritmen** att kunna köra färdigt på **under en minut**
- **O(n^2)-algoritmen** kommer däremot **att ta flera år (en miljard sekunder)!**

Lite repetition: Abstrakta datatyper och rekursion

- **Abstrakta datatyper** (ADT) är datatyper som definieras av vad de gör och inte hur de ser ut: dvs de **beskrivs av sitt beteende** och inte sin implementation

  Exempel: En lista kan vara både en ArrayList och en LinkedList, en stack går att bygga med både en array och en lista, osv

- **Rekursiva *metoder*** anropar sig själva (ex: binär sökning, rekursiv fibonacci) tills ett basfall har nåtts

- **Rekursiva *datastrukturer*** däremot är datastrukturer som innehåller en instans av sig själva

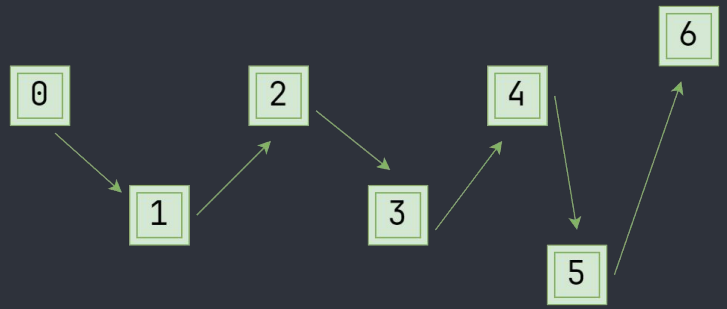
  Exempel: Vi gjorde en **Node**-klass som i sig innehåller en **Node**: en sån här struktur kan vi bygga en länkad lista med

Länkade Listor

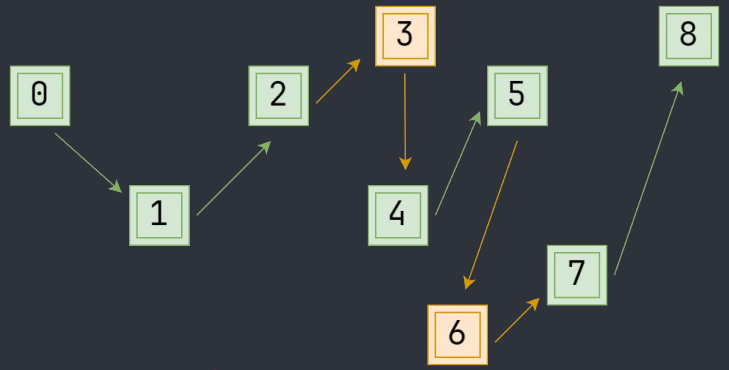
Array:

Allting lagrat i följd i samma minnnesblock

LinkedList:



Samma LinkedList med 2 nya noder:



Ostrukturerad minneslagring

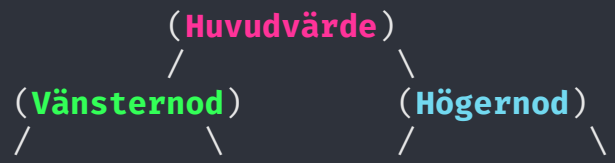
Rekursiva noder

- Vi gjorde en Node-klass för några veckor sedan som innehöll en instans av sig själv. I dag ska vi göra en **dubbelnod**, som innehåller två instanser av sig själv: en som pekar mot föregående nod och en som pekar mot nästa nod

- En sån här datatyp är utmärkt om man vill bygga en **Dubbellänkad Lista**, där ju varje länk i listan har koll på både vad som finns bakom den och framför den i kedjan:

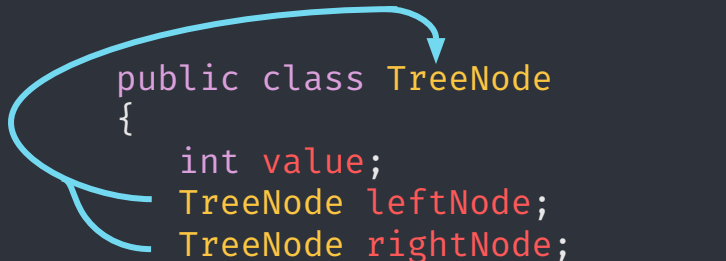
  Node1 ⟷ Node2 ⟷ Node3 ⟷ Node4

- Men den är perfekt för att skapa andra sorters abstrakta datastrukturer också, där man kan behöva lagra värden i vänster- eller högerled kring en huvudnod:



- Ett exempel på en sådan datastruktur är ett **binärt sökträd**

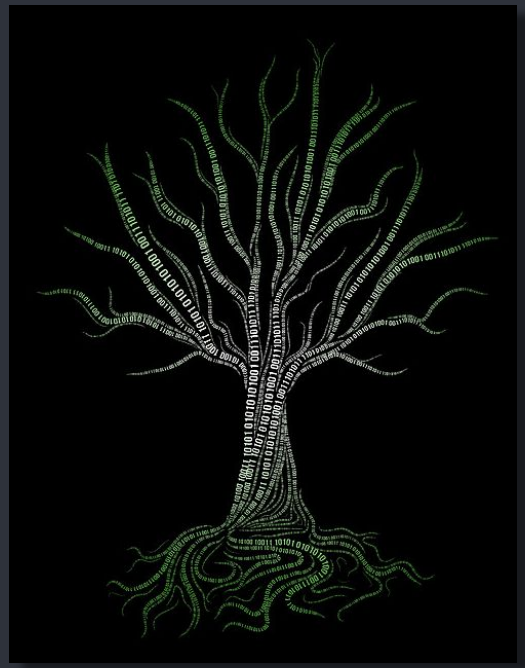
TreeNode: väldigt simpel rekursiv klass



```
public class TreeNode
{
    int value;
    TreeNode leftNode;
    TreeNode rightNode;

    public TreeNode(int value)
    {
        this.value = value;
    }
}
```

Binärt sökträd

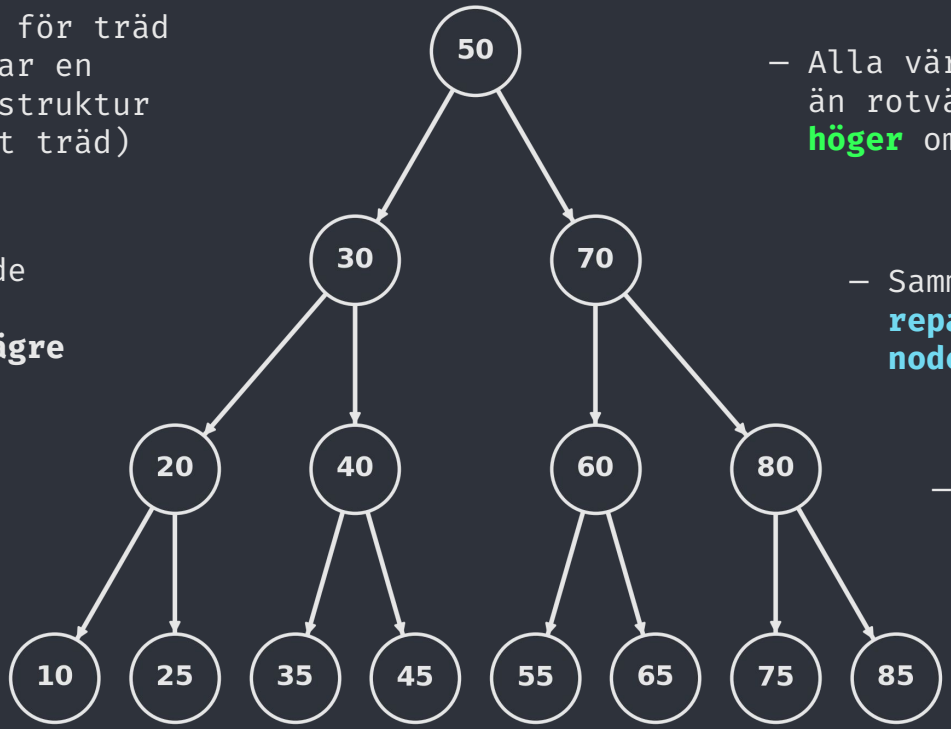
- Ännu ett exempel på en **abstrakt datatyp (ADT)**
- En datastruktur som är rekursiv till naturen eftersom vi bygger den av självrefererande noder (**TreeNode**)
- Sorterar datan åt oss automatiskt när vi stoppar in den
- Sökning i ett sånt här träd är ett exempel på **divide-and-Conquer**, eftersom man kan utesluta halva trädet varje gång: **tidskomplexiteten är därför O(logn)**
- **Ett binärt träd gör bokstavligen talat en binär sökning varje gång det letar efter eller stoppar in ett värde!**
- **Trädet kan ses som en visuell metafor för en binär sökalgoritm**
- Till skillnad från en binär sökning behöver vi dock **inte ha datan sorterad i en array:** trädet är en sorterad datastruktur med ickelinjär minneslagring



Trädstrukturen för ett binärt träd

- Vi kallar dem för träd eftersom de har en trädliknande struktur (uppochnedvänt träd)

- Rotnoden har ett värde
- Alla värden som är **lägre** än det värdet lagras till **vänster** om rotnoden



- Alla värden som är **högre** än rotvärdet lagras till **höger** om rotnoden

- Samma mönster **upprepas i alla undernoder**

- Noder som inte har några undernoder kallas för **löv**

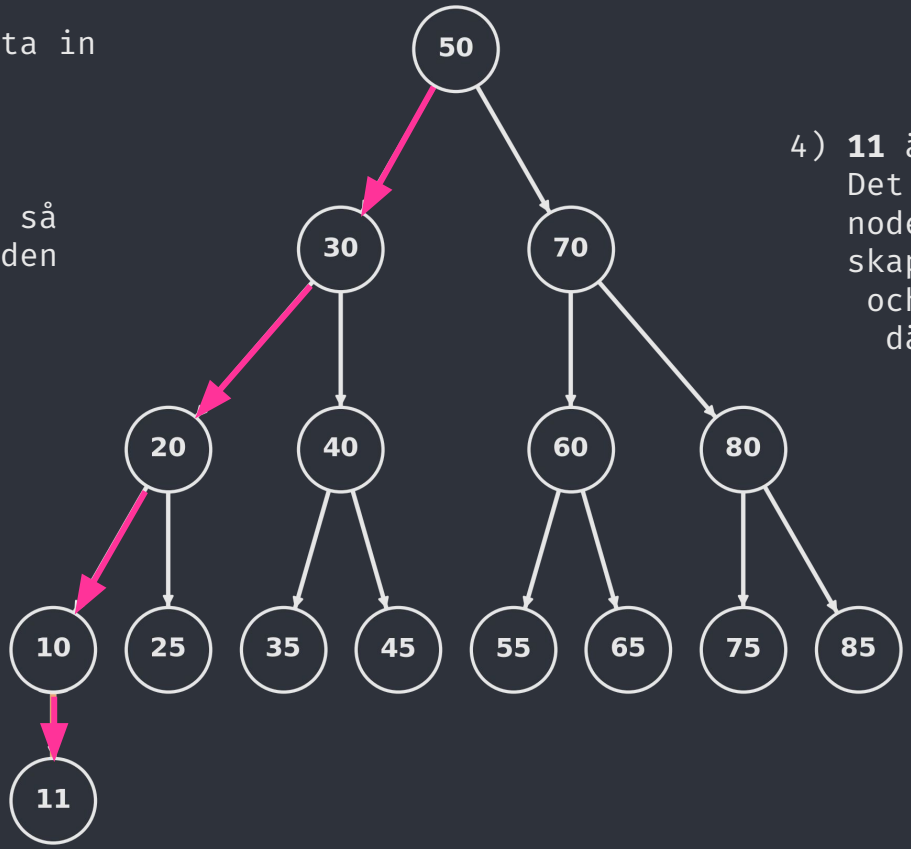
Insättning i trädet

– **Exempel:** Vi vill sätta in värdet 11 i trädet

1) **11** är **mindre** än **50**, så vi kollar vänstra noden

2) **11** är **mindre** än **30**, så vi kollar vänstra undernoden igen

3) **11** är **mindre** än **20**, så vi kollar vänstra under-noden igen



4) **11** är **större** än **10**. Det finns inga under-noder till 10, så vi skapar en högernod och lagrar värdet där

[Kodexempel]

Koden finns i paketet BinaryTree

```
kaffepaus(15);
```

Olika sätt att traversera ett binärt träd

- Utöver att kunna **söka efter specifika värden** behöver vi kunna traversera trädet för att besöka alla noder ibland
- Det finns **olika sätt att besöka** (“traversera”) noderna i ett binärt träd
- De tre vanligaste sätten kallas för **Inorder**, **Preorder** och **Postorder**
- Det går att koda andra fall om man vill men det är inte särskilt vanligt

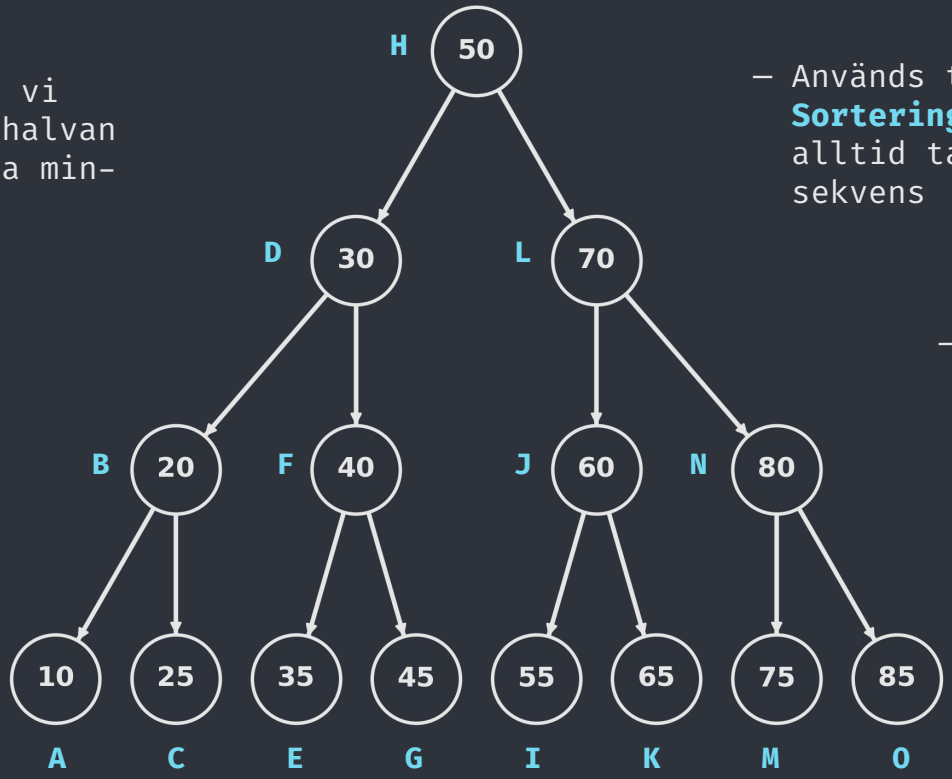
Olika sätt att traversera trädet:
Inorder (Vänster, rot, höger)

– Vi går alltid så långt vi kan längs den vänstra halvan av trädet för att hitta minsta värdet

Därefter:

1. Besök vänstra noden
2. Besök rotnoden
3. Besök högernoden

Outputen blir:
10, 20, 25, 30, 35,
40, 45, 50, 55, 60,
65, 70, 75, 80, 85



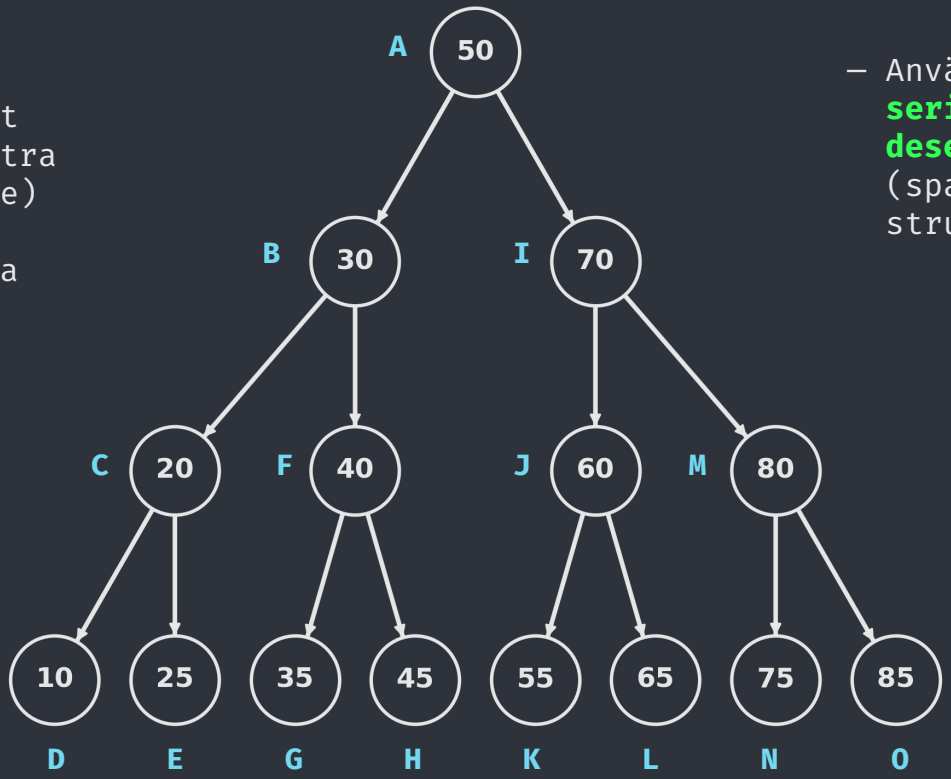
– Används t.ex. till **Sortering:** man får alltid talen i en sekvens

– Kallas inorder eftersom alla värdena **hamnar i rätt ordning**

Olika sätt att traversera trädet:
Preorder (Rot, vänster, höger)

1. Besök rotnoden först
2. Traversera det vänstra underträdet (subtree) först
3. Traversera det högra underträdet

Outputen blir:
50, 30, 20, 10, 25,
40, 35, 45, 70, 60,
55, 65, 80, 75, 85



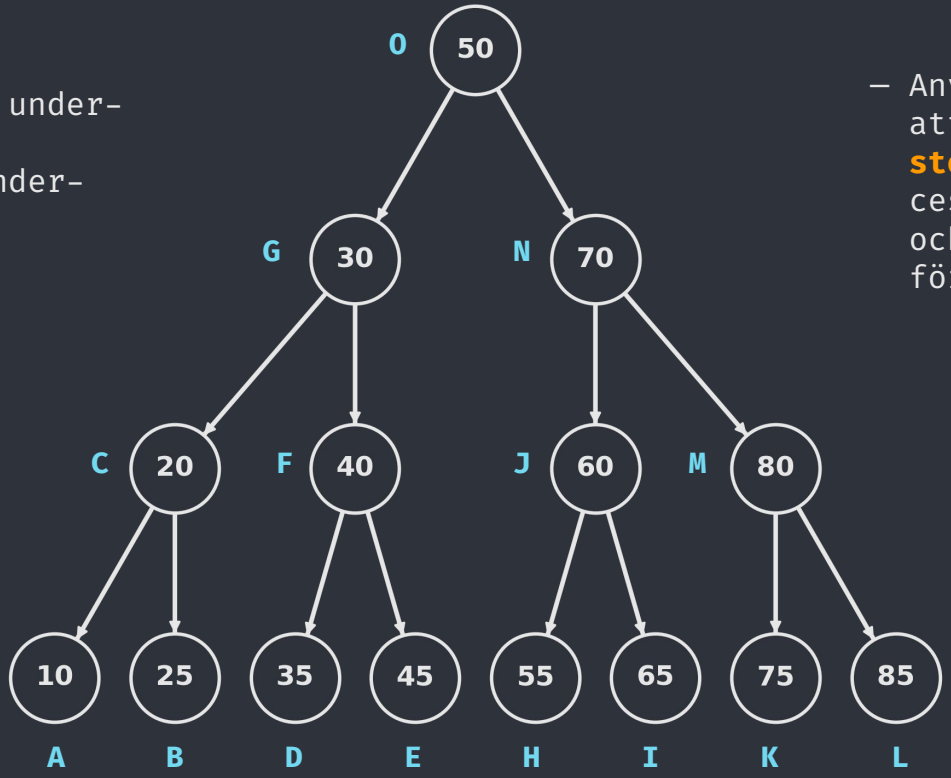
– Används t.ex. till **serialisering** och **deserialisering** (spara och rekonstruera trädet)

– Kallas preorder eftersom **roten besöks före (pre) undernoderna**

Olika sätt att traversera trädet: Postorder (Vänster, höger, rot)

1. Traversera vänstra under-trädet (subtree)
2. Traversera högra under-trädet
3. Besök rotnoden

Outputen blir:
10, 25, 20, 35, 45, 40, 30, 55, 65, 60, 75, 85, 80, 70, 50



– Används t.ex. till att **beräkna filstorlek:** man processerar alla filer och undermappar först

– Postorder eftersom **roten besöks efter (post) undernoderna**

Bra minnesregel: Djupet först

- Notera att vi **alltid går på djupet först** (dvs går så djupt vi kan åt vänster) när vi scannar av trädet
- **Anledningen är att rekursion alltid fungerar enligt principen djupet först** (kom ihåg rekursiv fibonacci och hur den alltid returnerar den vänstra anropskedjan först)
- Med andra ord: **trädtraverseringen**, den fysiska vägen genom trädet, **är alltid densamma**
- Den **enda skillnaden är ordningen som vi besöker noderna** (dvs skriver ut värdena som är lagrade i dem)
- När vi säger “olika sätt att traversera trädet” menar vi ordningen som vi skriver ut värdena i, men det är egt. lite missvisande eftersom **själva traverseringen** av trädet **aldrig förändras**

[Kodexempel]

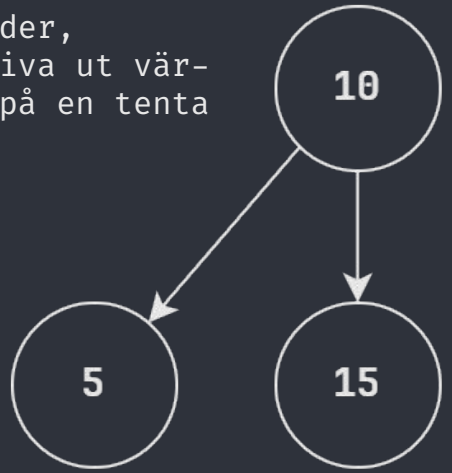
Koden finns i paketet BinaryTree

Bra minnesregel: förstå namnen

- Förstår man innebörden av namnen (Inorder, Preorder, Postorder) kan man lätt klura ur hur man ska skriva ut värdena om det t.ex. skulle komma en sådan uppgift på en tenta

- **Inorder (VRH)** innebär att vi alltid får en sortering: vi **besöker lägsta värdet först,** sedan mittenvärdet, och sist det högsta: **5-10-15**

- **Preorder (RVH)** innebär att vi **besöker roten före undernoderna,** men i övrigt besöker vi undernoderna från vänster till höger som vanligt: **10-5-15**

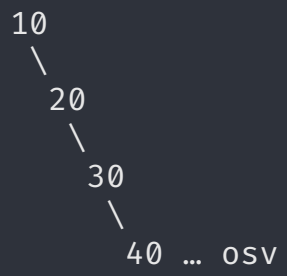


- **Postorder (VHR)** betyder att vi **besöker roten sist,** men precis som i de andra fallen besöks undernoderna från vänster till Höger: **5-15-10**

- **Inget av fallen börjar med att besöka högernoden!**

Obalanserade träd: ett problem

- Ett obalanserat träd har i värsta fallet **O(n)** för uthämtning av data eftersom det kan kollapsa till en länkad lista om man bara går ner längs den ena förgreningen:



- Ett självbalanserande träd, också kallat ett **Red and Black Tree**, har alltid tidskomplexiteten **O(logn)** för uthämtning eftersom det ser till att den här kollapsen aldrig sker

Rödsvarta träd (Red-black tree)

- **Rödsvarta träd** är som sagt självbalanserande: du behöver inte göra någonting! \o/
- Namnet är någonting man valde för att beskriva metaforiskt hur trädet har två typer av noder
- Exempel på datastrukturer i Java med balanserade rödsvarta träd:

TreeMap<K, V>	Lagrar **key-value-par**
TreeSet<E>	Lagrar **element** (dvs objekt men **INTE** primitiva datatyper)

- En utmärkt datastruktur för stora datasamlingar som behöver traverseras ofta

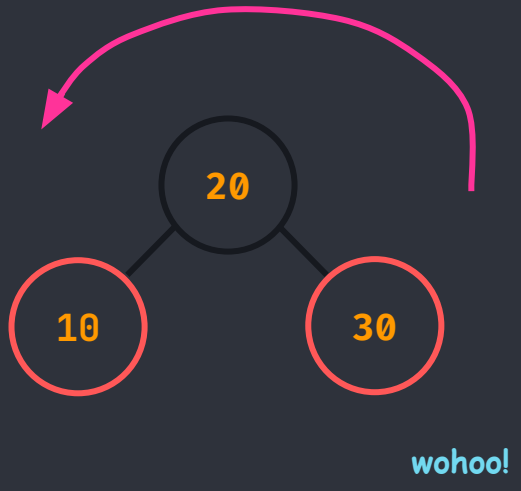
Rödsvarta träd: ombalansering

Fyra regler för ett rödsvart träd:

* Varje nod är röd eller svart
* Roten är svart
* Ingen röd nod får ha en röd undernod
* Varje ny nod vi lägger in **börjar som en röd nod**

1. Vi lägger in värdet 10 i roten
2. Vi lägger in värdet 20 i högerled
3. Vi lägger in värdet 30 i högerled
4. Vi har nu en länkad lista 😢

Rödsvarta träd: ombalansering



Om ett träd bryter mot någon av reglerna efter insättning:

* Roteras trädet (O(logn) i värsta fall)
* Noderna färgas om
* Länkar om eventuella undernoder (O(1) eftersom de bara är referenser som uppdateras)

1. Trädet roterar åt vänster så att 20 blir rotnoden och färgas om
2. 10 blir vänsternoden och färgas om
3. 30 stannar där den är och behåller sin färg
4. Vi har nu ett balanserat träd! \o/

Tidkomplexiteten för binära träd

	Balanserat träd	Obalanserat träd
Sökning:	O(logn)	O(n)
Insättning:	O(logn)	O(n)
Radering:	O(logn)	O(n)

Divide-and-conquer:
Vi halverar trädet varje gång vi gör ett val att följa en förgrening, precis som en binär sökning halverar samlingen varje gång vi kollar om mittvärdet är större eller mindre än måvärdet.

Är traversering av alla noder O(n) eller O(n*logn)?

- Intuitionen säger att det **borde vara O(n*logn):** om det tar O(logn) att hitta ett värde i ett träd bör det ta O(logn) gånger O(n) att hitta alla värden, eller hur?
- **Fel!** Tidskomplexiteten för att traversera ett **träd är O(n)**
- Anledningen är att vi **inte gör en sökning** för varje värde när vi traverserar trädet
- Jämför med en länkad lista: att **iterera genom hela listan tar O(n)** eftersom vi bara följer noderna, oavsett faktumet att en sökning efter ett enskilt värde också är O(n)
- **Trädtraversering funkar likadant:** vi gör inte en sökning efter varje nod - vi följer bara referenser från en trädnod till en annan

Lunch(60);

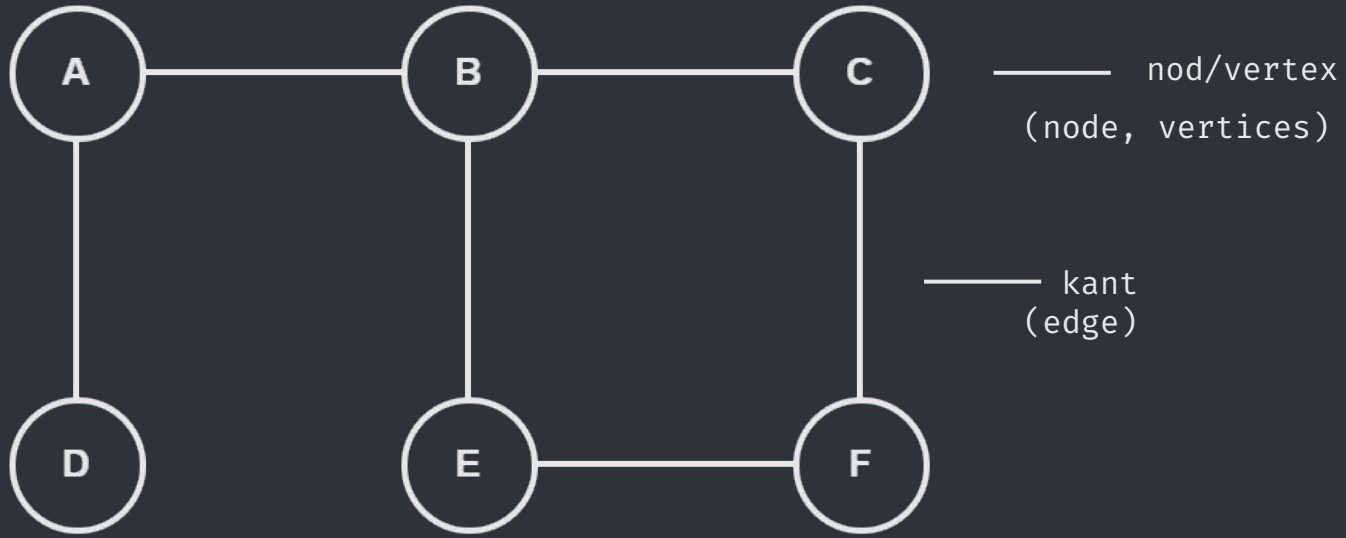
Vi ses i D22 13.15


Grafer

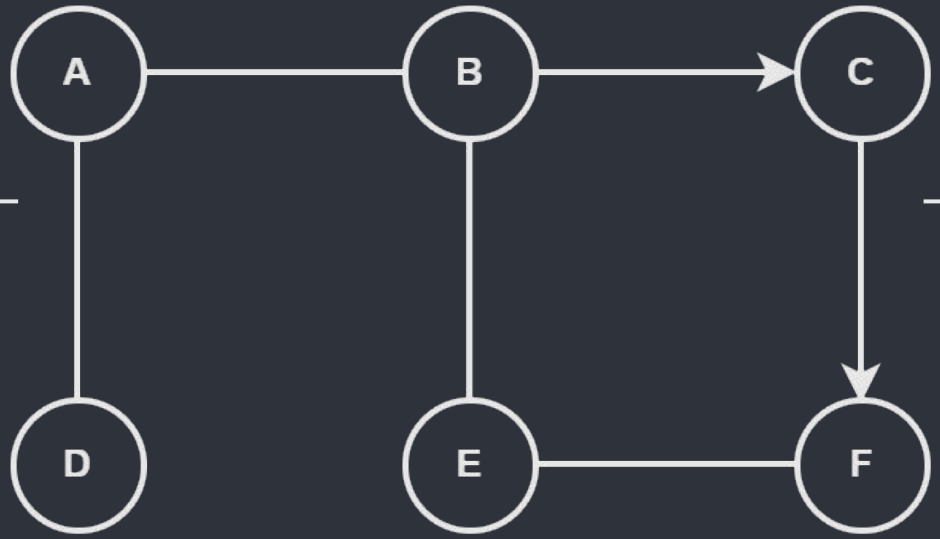
Grafer

- En graf är en **abstrakt datatyp (ADT):** den har ingen egen datastruktur utan använder andra såsom arrayer, listor eller maps för att simulera ett beteende

- En graf representeras av noder (cirklar) som ansluts till varandra med kanter (linjer):



Grafer



Oriktad kant
(undirected edge)

Det finns en väg
från A till D och
från D till A

Riktad kant
(directed edge)

Det finns en väg
från C till F men
inte från F till C

Användningsområden

- Grafer används överallt där förhållanden, anslutningar och vägar existerar mellan någonting. **Exempel:**

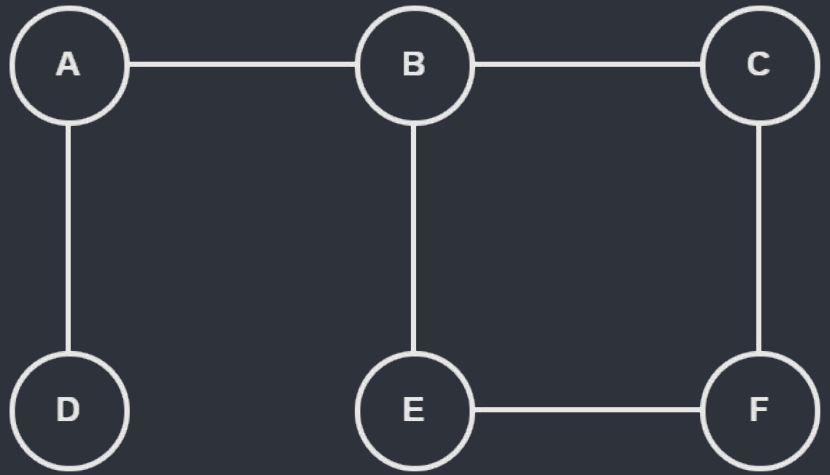
  Sociala nätverk: Förhållanden mellan människor
  GPS-system; Kortaste vägen mellan två orter
  Datornätverk: Routing och överföring av data

- Grafdatabaser används ofta i sammanhang **där förhållandet mellan data är lika viktigt som datan i sig själv** (t.ex. Neo4J)

- Spotify, Facebook, Twitter, osv använder alla olika former av grafer för att **modellera förhållanden mellan användare** och förutspå vem du kanske vill följa baserat på de du redan följer, vad du vill lyssna på, osv

Grannmatris (Adjacency Matrix)

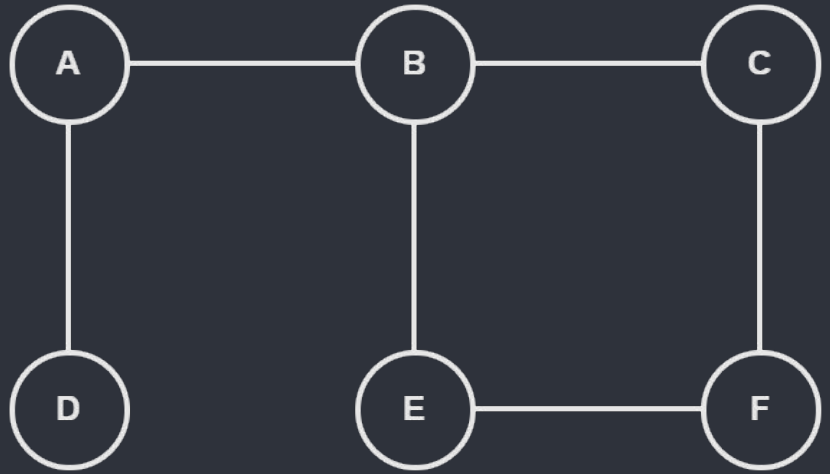
- Ett sätt att beskriva relationerna i en graf
- Använder sig av en 2D-array för att lagra relationerna
- 1 betyder att det finns en kant mellan noderna; 0 = ingen kant

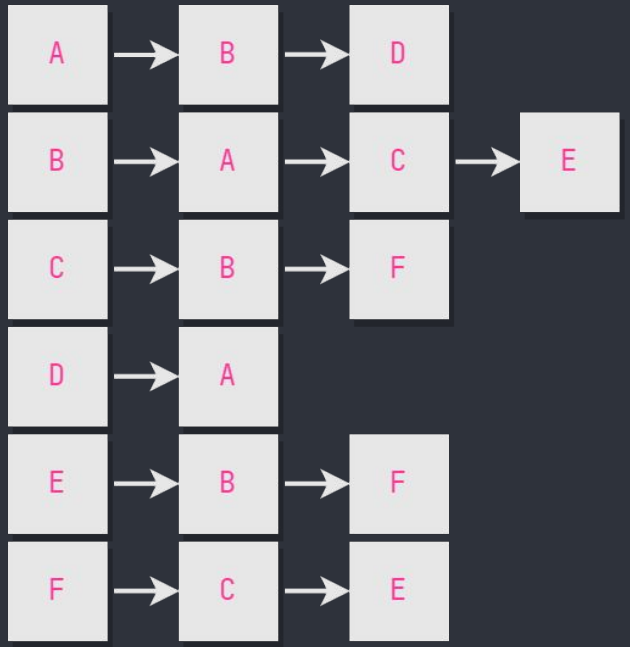


	A	B	C	D	E	F
A	0	1	0	1	0	0
B	1	0	1	0	1	0
C	0	1	0	0	0	0
D	1	0	0	0	0	0
E	0	1	0	0	0	1
F	0	0	0	0	1	0

Grannlista (Adjacency List)

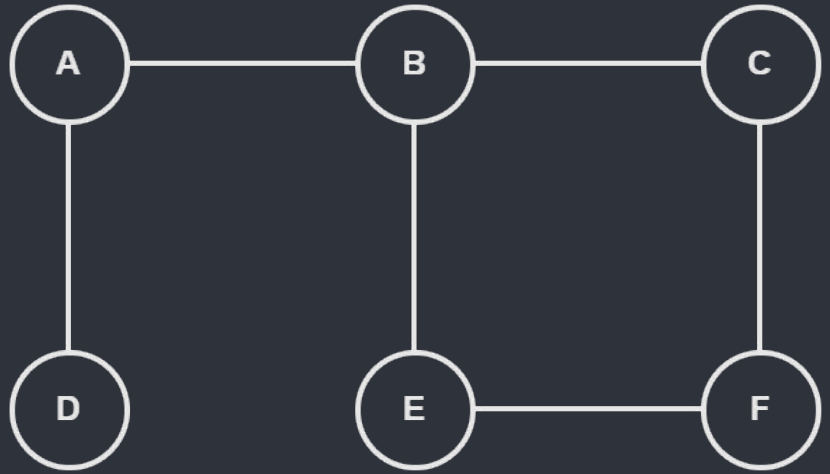
- Ytterligare ett sätt att lagra relationerna i en graf
- Använder en HashMap eller en Lista för att lagra varje nods grannar

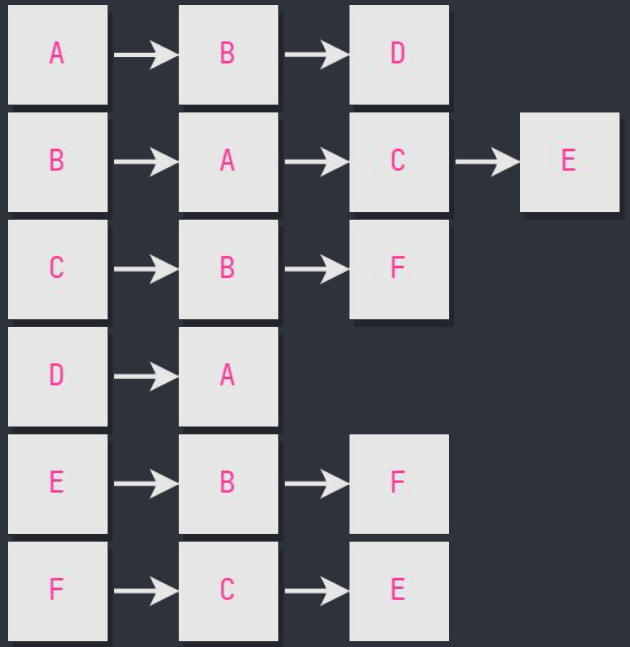




Grannlista (Adjacency List)

- Ytterligare ett sätt att lagra relationerna i en graf
- Använder en HashMap eller en Lista för att lagra varje nods grannar





När används vad?

- Vi sa att en 2-dimensionell array är vanlig att lagra grannmatriser i: nack-delen med den här typen av lagring är att om grafen innehåller få kanter mellan de flesta av noderna så kommer vi slösa väldigt mycket utrymme på att lagra nollor

- Varje cell i ett 2-dimensionellt rutnät besvarar frågan: **“Finns det en kant från vertex i till vertex j?”**

- Innebär att platskomplexiteten blir $O(n^2)$ där n är antalet noder
  (uttryckt i grafkomplexitet säger vi egentligen $O(V^2)$ där v = antalet vertexar)

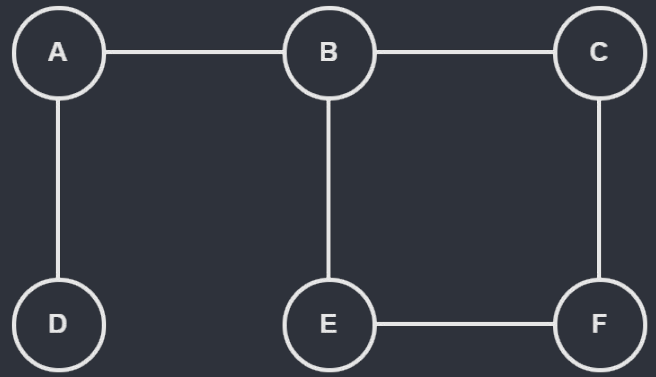
- Det här innebär att om det finns **1000 noder** måste 2d-arrayen ha **en miljon celler**. Finns det **tiotusen noder** måste den lagra **100 miljoner celler**, även om de flesta av dem är 0 (saknas en kant mellan de flesta av noderna)

- Om grafen innehåller få relationer mellan varje nod är det därför bättre att använda en grannlista, som då har platskomplexiteten O(V+E)

- Eller översatt till n: ungefär O(n + m), där n är antalet noder och m är antalet kanter som existerar

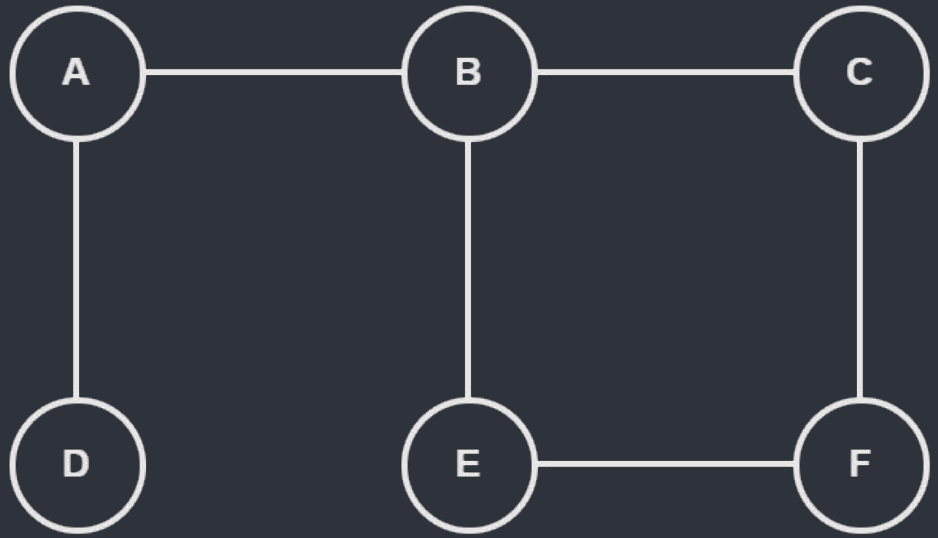
DFS implementerad med en stack

1. Lägg A på stacken. **Stacken innehåller: A**
2. Besök A. A har en kant till B. Lägg B på stacken. **Stacken innehåller: A, B**
3. Besök B. B har en kant till C. Lägg C på stacken. **Stacken innehåller: A, B, C**
4. Besök C. C har en kant till F. Lägg F på stacken. **Stacken innehåller: A, B, C, F**
5. Besök F. F har en kant till E. Lägg E på stacken. **Stacken innehåller: A, B, C, F, E**
6. Besök E. E har inga kanter till nya noder så vi poppar den från stacken. **Stacken innehåller: A, B, C, F**
7. Besök F igen. F har inga fler kanter till obesökta noder. Poppa F. **Stacken innehåller: A, B, C**
8. Besök C igen. C har inga fler kanter till obesökta noder. Poppa C. **Stacken innehåller: A, B**
9. Besök B igen. B har inga fler kanter till några obesökta noder. Vi poppar B. **Stacken innehåller: A**
10. Besök A igen. A har en kant till D. Vi lägger D på stacken. **Stacken innehåller: A, D**
11. D har inga kanter till obesökta noder, så vi poppar den från stacken, och återvänder till och poppar sedan A. **Stacken är nu tom.**



Bredden Först (Breadth First Search)

- Implementeras med hjälp av en **Kö**
- Besöker först alla noder som A har en kant till, och sedan alla noder som de noderna har en kant till, och-så-vidare
- Backtrackar inte, så den är (normalt) iterativt implementerad

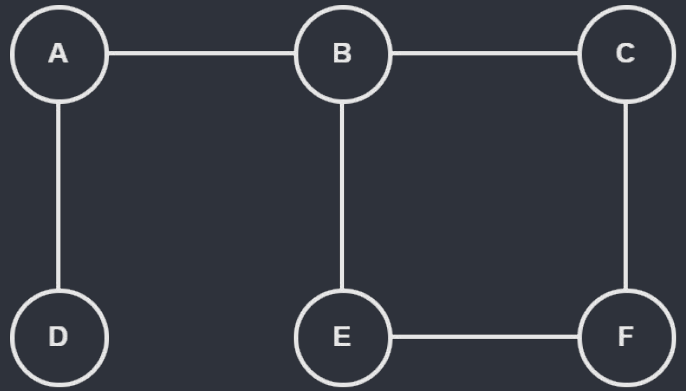


BFS med start från A, om alla noder lagrar vägen till andra noder i bokstavsordning:

A B D C E F

BFS implementerad med en kö

1. Lägg till A i kön. **Kön innehåller: A**
2. Besök A. A har en kant till B och D. Lägg till B och D i kön. Ta bort A från kön. **Kön innehåller: B, D**
3. Besök B. B har en kant till C och E. Lägg till C och E i kön.Ta bort B från kön. **Kön innehåller: D, C, E**
4. Besök D. D har inga kanter till andra noder. Ta bort D ur kön. **Kön innehåller: C, E**
5. Besök C. C har en kant till F. Lägg till F i kön. Ta bort C ur kön. **Kön innehåller: E, F**
6. Besök E. E har inga kanter till noder vi inte besökt. Ta bort E ur kön. **Kön innehåller: F**
7. Besök F. F har inga kanter till noder vi inte besökt. Ta bort F ur kön. **Kön är nu tom.**



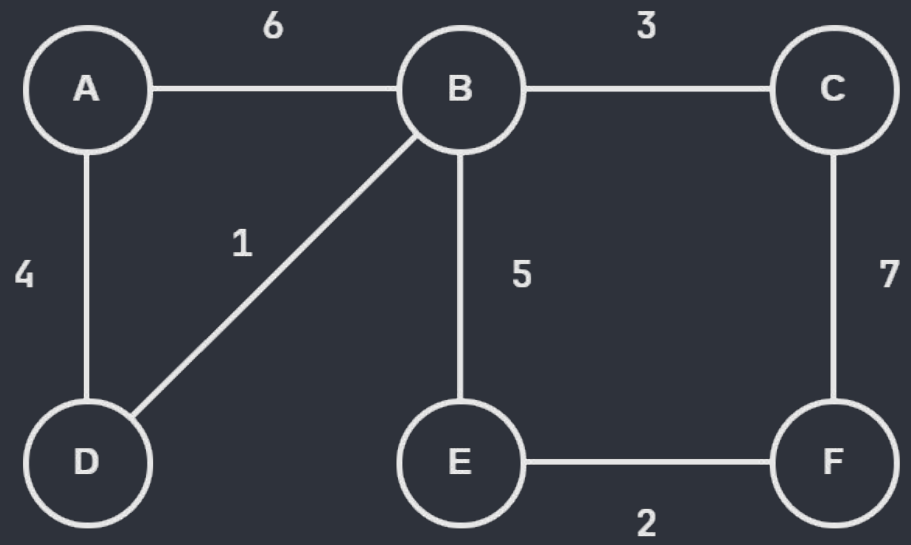
Dijkstras algoritm: Kortaste vägen i en graf

- **Dijkstras algoritm** hittar den kortaste vägen i en graf
- Namngiven efter **Edsger Dijkstra** som uppfann den. Ett av 1900-talets största intellekt; fick en **Turing Award** (motsv. till Nobelpris)

  Exempel: Netflix använder ett nätverk av servrar, CDN (Content Delivery Network), för att lagra filmer. När du ansluter till tjänsten och spelar upp en film har de en algoritm som **hittar närmaste geografiska serverplatsen** för att minimera latens
- Är exempel på **girig (greedy)** algoritm: dessa letar **alltid** efter minst belastning
- Idén är att **lokalt kortaste vägen = globalt kortaste vägen**
- Funkar dock bara så länge kantvikten inte är negativ

Dijkstras algoritm

- Kan implementeras med en **Prioritetskö**
- Siffrorna representerar kantvikten (Edge Weight)



Distanstabell för noden A

Nod	Kostnad	Via
B	5	D
C	8	B
D	4	A
E	10	B
F	12	E

Grafer har inte konventionell tids-komplexitet

- Grafer har **två variabler,** noder och kanter, vilket gör dem mer komplexa än arrayer, listor och liknande datastrukturer
- Det **går därför inte** att prata om dem i termer som O(n), för det finns inget enskilt n-värde som definierar grafen
- Man brukar säga att **komplexiteten är O(V+E)**, där V är antalet noder (Vertices) och E är antalet kanter (Edges)
- Det här är dock ingenting vi tar upp på kursen och inget ni förväntas kunna: vårt mål är bara att skapa en konceptuell förståelse för grafer

Giriga (greedy) algoritmer

- Vi nämnde att **Dijkstras algoritm** är girig, men giriga algoritmer är inte bara ett koncept som gäller för grafer

- Det är skillnad mellan **greedy** och **djupet först:** giriga algoritmer backtrackar aldrig. När de tagit ett beslut håller de fast vid det

- Problem som är lämpade för att lösas rekursivt är för det mesta omöjliga att lösa med giriga algoritmer eftersom de **aldrig ångrar sig**

- **Föreställ er en labyrint:** Dijkstras algoritm kommer ge oss den kortaste vägen från Start till Mål så länge vi alltid kan röra oss fritt mellan olika rum

- Om labyrinten har hinder (återvändsgränder, krokodilgropar, zombier, etc) **kommer den dock att misslyckas,** för den kan inte återvända när den tagit ett beslut att gå till ett rum där den visar sig vara fast

Exempel: en girig myntväxlare

- Säg att vi vill programmera en maskin som **ger tillbaka växelmynt** när någon har handlat
- Vi vill att den ska ge ut så få mynt som möjligt för att inte irritera kunderna
- En **girig algoritm** skulle kunna hjälpa oss att programmera maskinen: Om den ska ge ut 20 spänn i växel vill vi exempelvis att den delar ut två 10-kronor; inte 4 st 5-kronor
- Vi vill med andra ord att den **alltid ska välja det största möjliga myntet** först, och sen det näst största möjliga, osv

[Kodexempel]

Koden finns i klassen GreedyCoinChange.java

Vad händer om valörerna ändras så att de inte längre följer regelbundna mönster?

- Säg att vi i stället har tre sorters mynt som är värda 4, 3 och 1 kr och vi behöver ge ut växelpengar som motsvarar 6 kr

- Hur kommer myntväxlaren att bete sig?

Svar:

- Eftersom den är girig **kommer den att välja myntet som är värt 4 kr** först och sedan två stycken 1-kronor (4 + 1 + 1)

- Den **optimala lösningen** är dock att ge ut 2 st 3-kronorsmynt

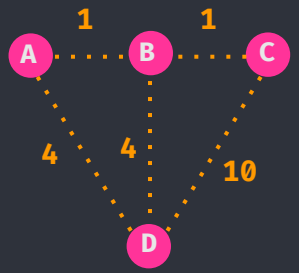
- Det här är anledningen till att giriga algoritmer t.ex. inte kan lösa **The Travelling Salesman**-problemet: i verkligheten är inte alltid det lokalt bästa valet samma sak som det bästa valet ur ett större perspektiv

Travelling Salesman

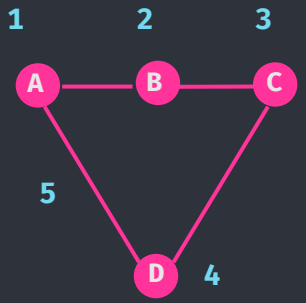
- Ett av datorvetenskapens mest ökända problem: saknar fortfarande lösning
- Givet ett antal städer och avstånden mellan dem: hur hittar du den kortaste möjliga vägen för att besöka dem alla och sen återvända till staden där du började?
- För 10 städer finns det ~362 000 möjliga kombinationer
- För 15 städer finns det **80 miljarder möjliga kombinationer**
- Tidskomplexiteten är O(n!)
- Man kan optimera mindre dataset med dynamisk programmering och liknande men det finns ingen universell optimering av problemet
-

Travelling Salesman

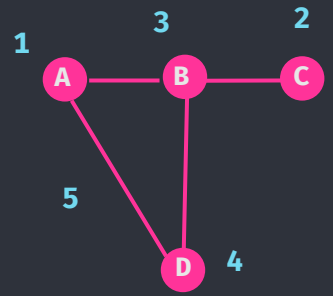
	A	B	C	D
A	0	1	4	4
B	1	0	1	4
C	4	1	0	10
D	4	4	10	0



Städernas placering



Girig algo väljer



Optimal väg

Heapen

- Vi fokuserade på **callstacken** senast, men **heapen** är lika viktig att förstå när vi vill skapa oss en bild av hur minne fungerar i Java

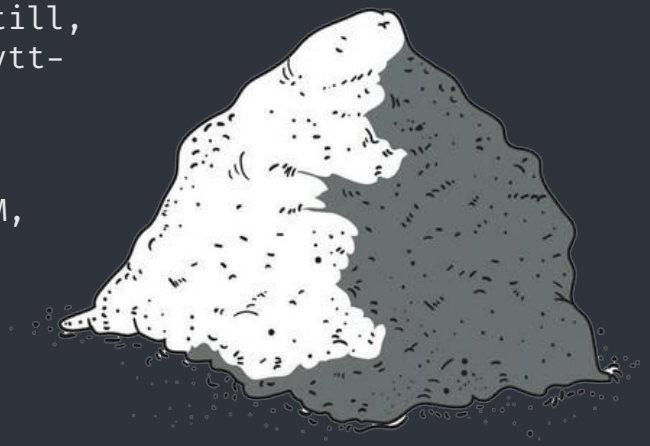
- **Alla objekt** lagras på heapen; stacken håller enbart referenserna till dem

- Vi nämnde tidigare exempel på vad grafer är bra till, såsom vägvisningsappar och sociala nätverk, men ytterligare ett användningsområde är inbyggt i Java självt: heapen

- Heapen i sig självt är bara ett minnesblock i RAM, dvs själva utrymmet är linjärt och inte en graf

- **Heapminnet kan dock visualiseras som en graf** med objekt som noder, där referenser formar kanter mellan dem

- **Garbage collection** utnyttjar detta för att frigöra minne



Hur garbage collectorn fungerar

- Den ser **varje objekt** i minnet **som en nod i en graf,** medan referenser mellan objekten ses som kanter som förbinder dem

- Rötterna (GC Roots) är en samling av speciella objekt som håller koll på **referenser som existerar när programmet startar:** stack-variabler, statiska variabler, trådar och liknande

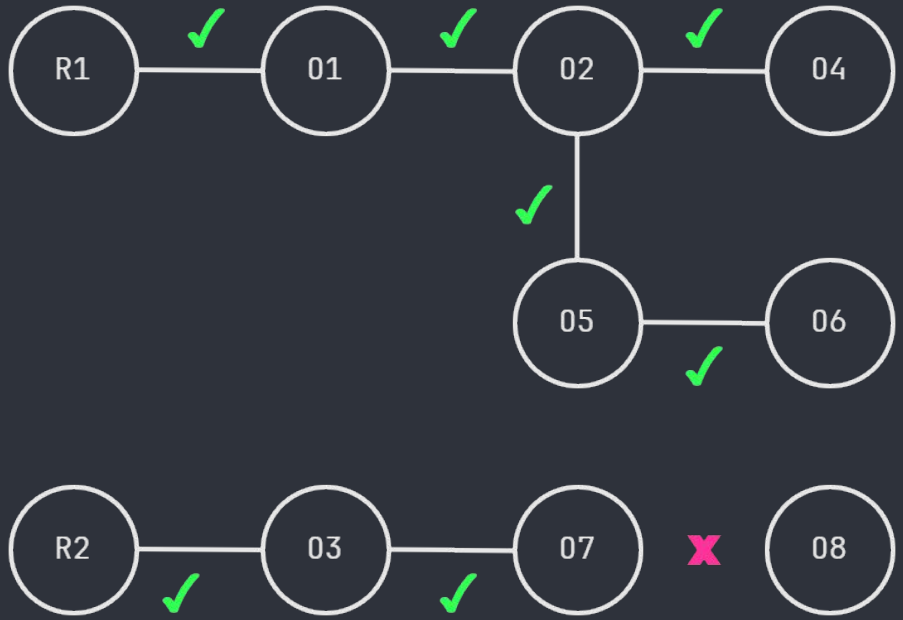
- Så länge en nod (dvs ett objekt) är ansluten till en rot anses den vara vid liv och behålls i minnet

- **När vi poppar en stackframe tar vi bort en kant** mellan två noder, och när garbage collectorn försöker traversera grafen på nytt kan den inte längre nå den yttersta noden

- Den märker då den här noden som “död”, varpå den sedan kommer att tas bort i nästa cykel

Hur objekt elimineras från heapen

– Garbage collectorn vet att objekt 8 finns, men den kan inte längre nå det genom traversering från Rot 2 efter att kanten har försvunnit



R: Rötter (GC Roots)
O: Objekt

– O8 märks för borttagning och kommer att raderas i nästa cykel

Varför just en graf?

- Varför använder garbage collectorn **en graf i stället för** att t.ex. stoppa alla referenser i **en lista** när någonting poppas från stacken och sen radera allt som finns i den listan?

- Anledningen är att **minneshantering inte bara handlar om** att hålla reda på variabler, utan även komplexa **förhållande mellan dem**

- Det kan till exempel finnas **flera referenser till ett objekt** i flera olika stackframes: om vi tar bort objektet bara för att en frame poppas kommer vi att **radera en bit levande minne** som fortfarande används!

- Just såna här grejer är grafer bra på att modellera: **förhållanden och relationer,** där vi behöver veta hur saker hänger ihop i flera olika led

Vad för sorts algoritm använder garbage collectorn?

- Varken rekursiv eller girig: även om den “imiterar” ett rekursivt mönster, precis som stacken gör, så är den **inte självrefererande**
- Den är **inte heller girig** eftersom den kan backtracka: den tar inte bort objekt så fort den hittar ett som inte går att nå, utan märker det för radering och fortsätter sedan
- I stället lagrar den objekt som den ska besöka **i en kö eller en stack,** och processerar dem sedan iterativt
- Algoritmen den använder kallas för **Mark-and-Sweep** och är en variant på **djupet-först-sökning** (DFS) i grafer som vi gick genom tidigare

Andra ställen där Mark-and-sweep används

- **Mark-and-sweep-algoritmer** är värdefulla i system som producerar "skräp", dvs resurser som behöver städas upp för att undvika belast-Ning för systemet. **Exempel:**

  Datornätverk: Brutna anslutningar, oanvända resurser och liknande som **belastar servrar och bandbredd**

  Unreal Engine: Objekt som sprites, assets och partiklar behöver städas upp periodiskt när de inte används: när du **skjuter en space invader** i ett spel vill vi inte ha kvar spriten för den i minnet längre

  Operativsystem: Processer som inte längre används avslutas för att **frigöra RAM-minne**

- **Graftraversering är vanlig** i många sammanhang som man inte reflek-terar över normalt, och därför är det bra att ha lite koll på