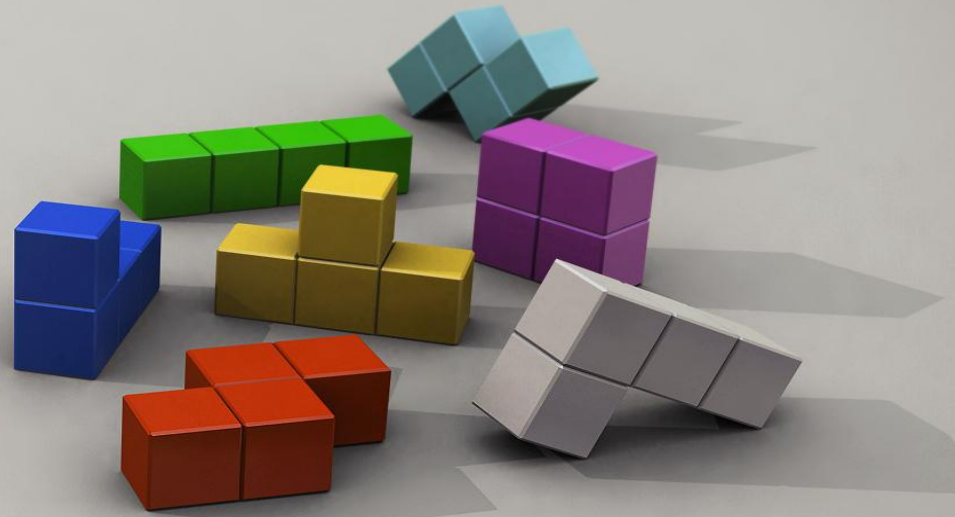
Algoritmer och Datastrukturer

Föreläsning 4



Abstrakta datatyper och generics

Dagens föreläsning: Stackar, köer och abstrakta datatyper

{

Vi ska:

- Kolla på rekursiva **datastrukturer**
- Prata om abstrakta datatyper
- Kika på generiska typer
- Fortsätta prata om rekursivitet
- Lära oss om köer och deras användningsområden
- Prata mer ingående om stacken som datastruktur
- Undersöka vad som händer på callstacken när vi anropar en rekursiv fibonaccialgoritm

}

Försök besvara dessa efter dagen

1) Vad är det för skillnad mellan en stack och en kö?
2) Vad är en ADT för något?
3) Ge två exempel på ADT:er.
4) Vad är en generisk typ (Generics) för någonting? Varför är det bra att känna till?
5) Vad är riskerna med casting?
6) Ge ett exempel på en rekursiv datastruktur och förklara hur den fungerar internt.
7) Vad händer med rekursiva metodanrop när de når ett basfall?
8) Vad är skillnaden mellan en ADT och en implementation av en ADT? Ge ett exempel.
9) Förklara förkortningarna FIFO och LIFO och vad de förknippas med.
10) Vad är en rekursiv datastruktur?

Lite om casting

- Casting (kastande) är typomvandling, och bör användas med försiktighet

- Vi “bryter någonting”, i det här fallet typsystemet, och får sätta på ett gips (“cast” på eng.) för att laga det

Exempel där ett cast är okej:

- **Implicit casting** (widening cast) är okej i Java: man kastar mindre datatyper till större (ex. **en int till en double**)

- **Explicit casting** (narrowing cast) är **riskabelt men ibland nödvändigt**: man kastar en större datatyp till en mindre, t.ex. en double till en int:

```
double num = 10.5;
int convertedNum = (int) num;  //Ok men förlorar data
```

Ett cast på ett brutet ben

- **Upcasting** (subklass till superklass) är i teorin okej eftersom **underklasser är instanser av superklassen** de ärver från, men kan ha sidoeffekter om underklassen t.ex. har metoder som inte finns i superklassen (detta är ofta fallet och **ännu en anledning till att undvika arv!**)

Lite mer om casting

Dålig casting:

– **Narrowing cast** utan explicit deklarering:

```
double num = 10.5;
int convertedNum = num;            //Kompileringsfel!
```

- **Downcasting** (kasta högre objekt till mindre): en Hund är en typ av Djur men Djur är inte en typ av Hund (ger **körtidsfel: ClassCastException**)

- **Casting mellan objekt som delar superklass** (t.ex. Dog och Cat): **livsfarligt** eftersom det tekniskt sett är lagligt

- **Inkompatibla datatyper:** En char är inte en String, eller vice versa:

```
String text = "Hello!";
char letter = (char) text;         //Kompileringsfel!

char letter = 'A';
String text = (String) letter;          //Kompileringsfel!
```

Rekursiva datastrukturer

- I måndags pratade vi om **rekursiva algoritmer**, men det finns också **datastrukturer** som är rekursiva till naturen

- Skillnad mellan algoritm och datastruktur: en **algoritm beräknar någonting**, en datastruktur är ett sätt att **organisera data i minnet** på en dator

- Vi såg exempel på metoder som anropar sig själva i går, men **klasser kan även innehålla instanser av sig själva**

  - Man kan tänka på dem ungefär som ryska **matrjosjkadockor:** ihåliga trädockor där varje docka innehåller en mindre version av sig själv

Exempel: Node

- En Node är vad det låter som: en nod-klass

- **En nod representerar ett dataelement i en datastruktur**, såsom listor, grafer, träd, köer, osv

- Tänk på en nod som en container som innehåller:

  1) Data, och
  2) En referens (eller i bland två referens**er**) till andra noder

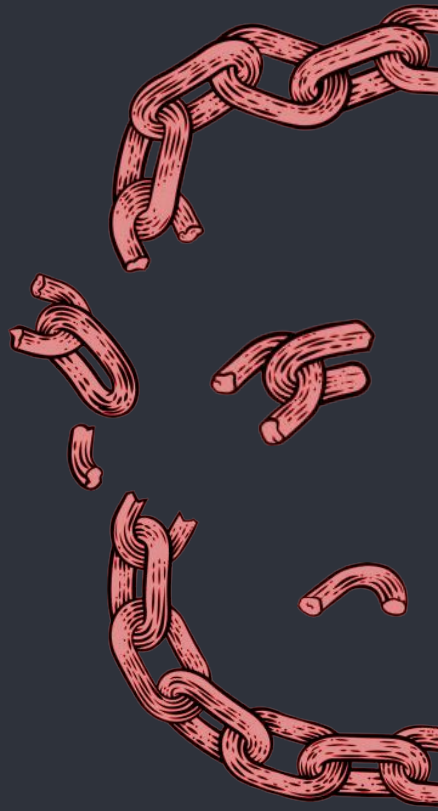
- Låt oss kika på ett exempel!

[Kodexempel]

Koden finns i klassen Node.java

Rekursiv datastruktur: Länkad Lista

- Vi kan använda Node-klassen vi byggde för att göra en länkad lista
- Varje nod innehåller en **instans av sig självt**
- Kedjeliknande struktur som fortsätter tills den når ett basfall (i det här fallet **när vi hittar en nod som är null**, dvs den sista noden i listan som inte blivit kopplad med en annan nod ännu)
- Flera av dess operationer (traversering, reversering) **kan implementeras rekursivt** på grund av detta
- I **dubbellänkade listor** (**Doubly Linked List**) innehåller varje nod två instanser av sig själv: en nod som pekar **mot föregående nod** och en som pekar **mot nästa nod**
- För enkelhetens skull ska vi dock bara göra en singellänkad lista



[Kodexempel]

Koden finns i klassen SingleLinkedList.java

SinglyLinked vs DoublyLinked List

- En **enkellänkad lista** har mindre overhead rent minnesmässigt: varje nod innehåller en referens i stället för två
- Det går snabbare att uppdatera den eftersom man bara behöver uppdatera en nod; om man bara vill traversera listan åt ett håll finns det en viss vinst här
- Om prestanda är viktigt är **dock en länkad lista en dålig data-struktur** att använda från början
- De **flesta länkade listor** som har en standardimplementation, som t.ex. LinkedList-klassen i Java, är dubbellänkade listor
- Länkade listor är **främst användbara** för att bygga abstrakta datastrukturer

ADT: Abstract Data Type

- En **abstrakt datatyp** är någonting som **definieras av sitt beteende** snarare än en specifik implementation
- Till skillnad från t.ex. **en array** finns det **mer än ett sätt** att skapa en abstrakt datatyp på
- **Är högnivåstrukturer:** Samma princip som när man pratar om högnivå- och lågnivåspråk i programmering: **lågnivå** är **nära hårdvaran (dvs minnet)**, **högnivå** har **fler lager av abstraktion**
- Exempel på datatyper som är abstrakta: **stackar, köer, kartor, listor** - det mesta som har beteende som går att generalisera

Datatyper som INTE är abstrakta

- Exempel på datastrukturer som **inte** är abstrakta:

Primitiva typer	int, float, bool
Arrayer	int[], double[], osv
ByteBuffer	I/O-strömmar med direkt byteåtkomst i minnet

- Strängar är ett specialfall. I Java är **String** en klass och **ingen primitiv datatyp** såsom int, double, bool, etc även om den ofta buntas samman med dem

- En sträng är egt. **bara en klass som lagrar bokstäver i en char-array**, men Java har bara en enda specifik implementation av en sträng

- Folk bråkar mycket om detta, men en kompromiss har ibland varit att kalla den en **“simpel abstrakt datatyp”**

Interface som kravlista

- **Abstrakta datatyper** beskrivs ofta av ett interface

- Ett interface är ett sorts kontrakt som beskriver vilka metoder en klass måste innehålla. Det innehåller metodhuvuden men inga metodkroppar (dvs ingen implementation). **Exempel:**

```
public interface IVideoRenderer
{
      void extractFrames(Path videoPath, Path file);
      void renderFrames();
      void createVideo();
}
```

- Alla klasser som implementerar **IVideoRenderer** måste innehålla de här tre metoderna, men är fria att bestämma själva hur koden inuti dem ska se ut och kan innehålla fler metoder utöver dessa

- Alla klasser som implementerar IVideoRenderer **anses vara av typen** IVideoRenderer

Interface

Metod i programmet som tar en IVideoRenderer som argument:

```
public void renderVideo(IVideoRenderer videoRenderer, String file)
{
    videoRenderer.extractFrames(file);
    videoRenderer.renderFrames();
    videoRenderer.createVideo();
}
```

När renderVideo() anropas kan vi mata in en av tre möjliga video-renderarklasser **som allihop implementerar IVideoRenderer men fungerar helt olika inbördes:**

```
renderVideo(new CPUVideoRenderer(),   "./samples/test.mp4");
renderVideo(new GPUVideoRenderer(),   "./samples/test.mp4");
renderVideo(new MultiCoreRenderer(), "./samples/test.mp4");
```

Några vanliga exempel på klasser i Java som implementerar olika interface

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

```
public class HashMap<K, V> extends AbstractMap<K, V>
        implements Map<K, V>, Cloneable, Serializable {
```

```
public final class Scanner implements Iterator<String>, Closeable {
```

```
public class Random implements RandomGenerator, java.io.Serializable {
```

Är ArrayList en abstrakt datatyp?

- Nej. **ArrayList** och **LinkedList** är båda ***specifika implementationer*** av **List**, som är en abstrakt datatyp som beskrivs av **Javas List-interface**

- En ledtråd är att vi kan skapa båda som instanser av List< >:

```
List<Integer> list = new ArrayList<>();
List<Integer> list = new LinkedList<>();
```

- Dessa implementationer kan dock bara se ut på ett vis, och därför anses de inte vara abstrakta. En ADT berättar ***hur någonting fungerar men inte hur koden ser ut***

- **List** är med andra ord datatypen, **ArrayList** är den specifika implementationen av den

Stacken som ADT

- Vi pratade om **callstacken** i måndags som lagrar metodanrop och primitiva datatyper medan ett program körs, men det är bara **ett exempel** på en konkret implementation av en stack

- Stackar definieras inte utifrån specifik kod **utan utifrån hur de fungerar:** **LIFO** (Last In First Out), **push()-**, **pop()-** och **peek()**-operationer, och-så-vidare

- **Alla klasser** du skapar som har den här funktionaliteten **kan följaktligen kallas för en stack**

- Man kan t.ex. göra en implementation av en stack både med en array och med en länkad lista

- För professionellt bruk använder man dock Javas inbyggda klasser

ArrayDeque

- I Java finns det en gammal datastruktur som heter just Stack, men den är märkt som deprecated och används inte längre förutom i legacykod som måste underhållas (den använder Vector, som också är märkt som deprecated)

- Modern Java använder en klass som heter **ArrayDeque** i stället om man behöver en stack:

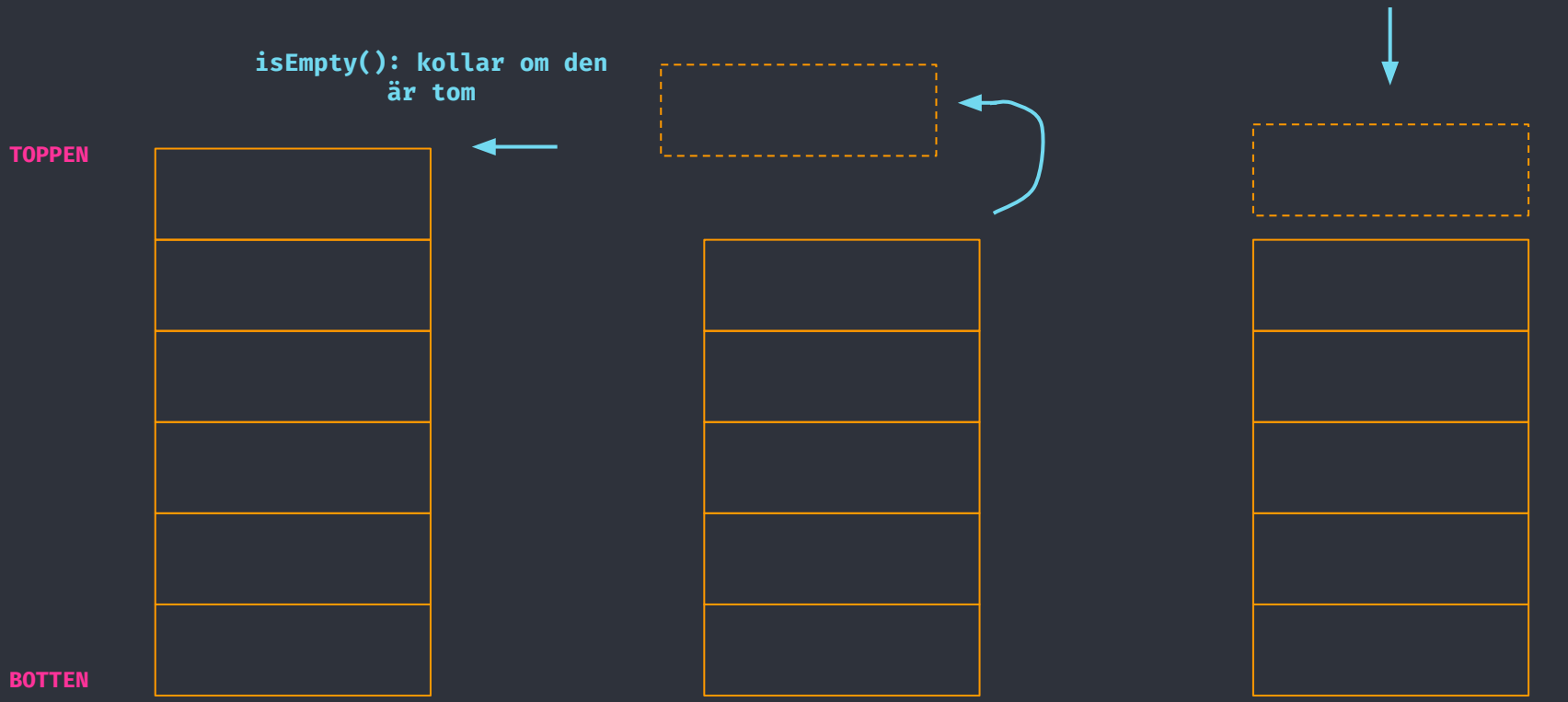
```
import java.util.ArrayDeque;

ArrayDeque<Integer> stack = new ArrayDeque<>();
```

- "Deque" står för "Double-ended Queue" och är **ett interface för köer** i Java, precis som List är ett interface för listor

- "Array" eftersom den använder en array för att implementera kön

- Vi ska prata mer om vad köer är för något, men först ska vi göra vår egen stack

Vad vi ska bygga:



[Kodexempel]

Koden finns i paketet MyStack i repositoryt

```
kaffepaus(15);
```

Generiska typer (Generics)

- **Generics är templateklasser:** deras fält har inga fasta datatyper, utan vid instansiering berättar man i stället vad man vill att datastrukturen ska spara för sorts data

- **Ni har redan använt såna här flera gånger** när ni t.ex skapat en ArrayList och specificerat typen inom < >:

```
ArrayList<Integer> numberList = new ArrayList<>();
ArrayList<String> stringList = new ArrayList<>();
```

- Vi kan modifiera vår länkade lista så att den blir generisk i stället om vi vill, och då kan vi spara vilken datatyp som helst i den

- Allt vi behöver är att addera **en så kallad typ-parameter** till klassen som ser ut så här: < >

- Inuti typparametern lägger vi till en template som sedan ersätts med själva datatypen som vi anger när vi skapar en instans av listan:

```
public class SingleLinkedList<Type>
```

[Kodexempel]

Koden finns i paketet GenericLinkedList i repositoryt

Generisk lista begränsad till numeriska typer

- Med dessa modifikationer kan nu listan användas för alla datatyper, och kan därmed anses vara **generisk**

- Det finns dock fortfarande ett problem: **Listan kan just nu ta ALLA sorters objekt vi matar in i den,** men vi vill kanske bara att den ska hantera numeriska typer

- Vi kan i så fall förlänga vår generiska lista med ett **interface** för att skapa ett kontrakt som bestämmer vilken typ av data som ska accepteras

- I Java är t.ex. **Number** en abstrakt klass i paketet **java.lang**. Det är en superklass för alla wrapperklasser för numeriska typer

- Om vi låter vår typparameter (obs! Inte själva klassen!) ärva från Number skapar vi en regel som säger att listan bara kommer acceptera numeriska värden:

```
public class GenericLinkedList<Type extends Number>
```

Numeriska typer

Primitiva typer: **short**, **int**, **long**, **float**, **double**, **byte**, **char**

Wrapperklasser: **Short**, **Integer**, **Long**, **Float**, **Double**, **Byte**, **Character**

När vanlig precision inte räcker lagras nummer som strängar och kan då teoretiskt vara hur stora som helst:

BigInteger (för heltal)

BigDecimal (för flyttal)

Både BigInteger och BigDecimal implementerar Number-interfacet i Java, precis som de primitiva typerna, trots att de lagrar strängar och inte bitrepresentationer av tal.

Generics är en form av polymorfism

- Generics är ett exempel på kompileringspolymorfism (**“compile time polymorphism”**)
- När koden kompileras ersätter Java placeholdern **Type** (eller T, eller vad vi nu väljer att kalla den; namnet spelar ingen roll) med den faktiska typen som datastrukturen ska lagra
- Interface och abstrakta klasser däremot är ett exempel på körtids-polymorfism (**“runtime polymorphism”**)
- Under körtid kommer en klass eller metod att exekveras annorlunda beroende på vad den är för något
- **Compile time polymorphism** är snabbare än **runtime polymorphism** eftersom vi redan löst vilken sorts typ en klass ska vara när vi kompilerar
- **Runtime polymorphism** är dock mer flexibel eftersom vi inte behöver känna till typen när vi kompilerar

Om vi vill göra en array generisk behöver vi vara lite kreativa

- Problemet är att Java inte tillåter generisk instansiering med **new** på grund av hur språket är uppbyggt

- Den här koden ger kompileringsfel (“Generic array creation is not allowed in Java”):

```
public class Stack<Type>
{
      private Type stack;
      public Stack(int capacity)
      {
          stack = new Type[capacity];   //Otillåten operation!
      }
}
```

- Object är rootklassen som **ALLA** klasser implicit ärver från i Java; därför kan vi kasta vår templatetyp till ett Object (**upcasting**):

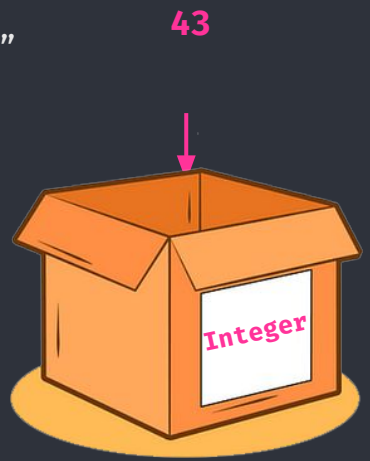
```
stack = (Type[]) new Object[capacity];    //Tillåten operation!
```

Om wrapperklasser

- Några av er har kanske noterat att vi skriver “Integer” och inte “int” inom <> när vi instansierar en generisk typ
- **Generiska typer kräver objekt**, men en int är en **primitiv typ** och inte en klass. Javas lösning är att skapa wrapperklasser som **“slår in”** (även kallat för “boxing”) en primitiv typ i ett objekt i stället
- När man sedan vill plocka fram värdet igen behöver man då “unboxa” det, dvs packa upp det ur lådan
- Ni har använt wrappertyper varje gång ni instansierat en generisk datastruktur:

```
ArrayList<Double> list = new ArrayList< >();
```

- Listan unboxar dessa värden automatiskt åt oss när vi hämtar ut dem



Överkurs: Vill man veta mer om varför de behövs kan man googla på **“type erasure + Java”**, men det är överkurs och inget ni förväntas kunna. Ni behöver bara förstå att det finns en skillnad mellan en wrapperklass och en primitiv typ

Wrapperklasser

- Det här är varför några av er säkert fått komplettering på Javalabbar där ni t.ex. skrivit **Double** i stället för **double** när ni deklarerat variabler

- En primitiv typ går att utföra matematiska/logiska operationer på direkt medan en wrapperklass behöver hämta ut värdena med metoder

- Finns också skillnader i **vilken sorts sorteringsalgoritmer** de aktiverar

- Inget ni behöver arbeta med utöver de användningsområden ni redan känner till: när man ska specificera typer för listor, Maps, osv

- Är dock bra att förstå att **Integer är ett objekt medan int är en primitiv typ**

- Primitiva typer representerar nummer direkt i bitform och går därför att beräkna:

```
int number1 = 200;                  //Lagras som 11001000 i bitform i minnet
Integer number2 = 200;        //En referens till ett objekt som innehåller en int
```

Tidskomplexitet för stacken som ADT

- Eftersom vi använder en underliggande array är vår implementation av en stack väldigt effektiv
- **push()**, **pop()** och **peek()** har **O(1)** i tidskomplexitet, dvs konstant tid
- I en array tar det lika lång tid att hämta ut värdet på index-plats **array[1]** som på plats **array[1000000]**
- Det här är också varför callstacken är **så snabb**: allting går på konstant tid med direktminnesåtkomst (direct memory access)!

Användningsområden för stackar

- Vi känner redan till att **stacken som ADT** används för att skapa callstacken inom programmering, men stackar finns överallt:

Webbläsare:	Framåt/bakåt-knapparna använder stackar
Ordbehandlare och IDE:	Ångra (Ctrl+Z) och upprepa (Ctrl+Y) implementeras genom att pusha och poppa saker på/från stackar
Versionshantering:	Historik kan rullas tillbaka om en commit blir dålig: poppa allt fram till dess bara. Gör man en soft reset sparas poppade commits i en redo-stack så länge.
Kompilatordesign:	Parsa uttryck, hantera symboltabeller, osv

- Allting som kräver att man **backtrackar är fundamentalt lämpat** för stackar, och ur det avseendet liknar de rekursion även om de inte är rekursiva i sig själva

Köer: som stackar, fast tvärtom

- Vi nämnde en **dubbellänkad lista** förut, där **varje nod innehåller två noder**, en som pekar bakåt och en som pekar framåt:

  Node1 ⟷ Node2 ⟷ Node3 ⟷ Node4 ⟷ Node5 ⟷ Node6 ⟷ Node7

- En sådan här struktur är också perfekt för att skapa en kö (**Queue**)

- Till skillnad från stacken, som fungerar enligt principen **LIFO (last in first out)**, är en kö **FIFO (first in, first out)**

- Ofta vill vi kunna köa både data och processer. Ett datorprogram vill undvika konflikter när det körs: **för det mesta utförs instruktioner en åt gången på CPU:n** (flerkärnig programmering är svårt och vinsterna är inte uppenbara om man inte har väldigt mycket data som kan delas upp i deljobb på ett tydligt vis)

Beteende för en kö

- **Fungerar precis som kön på ICA:** de som kommer först och står längst fram får hjälp först, resten får vänta tills alla före dem har betalat för sina varor

- **Det som legat lagrat längst hämtas med andra ord ut först**

- Köer bryr sig inte om vad som finns i mitten: de är bara intresserade av att kunna hämta ut data (från början av kön) och att kunna lägga till ny data (i slutet av kön)

- En dubellänkad lista brukar vara ett populärt val när man bygger köer av den här anledningen: den har redan en referens till den första och sista länken, så uthämtning och insättning går på O(1) i tid

- Däremellan finns det ingen poäng i att datan ligger lagrad sekventiellt (som i en array), och den länkade listans fragmentariska natur är därför en fördel

Användningsområden för köer

- Köer används ofta i sammanhang **där sekvenser är viktiga**

  Exempel: CPU-schemaläggning, utskriftsköer, meddelanden som skickas via Discord, matchmaking i onlinespel, buffertar för videostreaming, och-så-vidare. Allting med en prioritetsordning använder en kö. Utan köer skulle streamad musik hacka och Netflix skulle visa alla bildrutor i oordning.

- Inom algoritmdesign används de ofta för att hålla koll på traversering i träd och grafer

- En **dubbellänkad lista** (Javas LinkedList är en sådan) har tidskomplexiteten **O(1)**, dvs **konstant tid**, för insättning, uthämtning och radering i båda ändarna

Kö i Java: ArrayDeque (igen!)

- Java använder samma datastruktur för både stack och kö
- Om du använder push() och pop() har den stackfunktionalitet; om du använder queue() och deque() kan den betraktas som en kö
- **Om du skulle mixa beteenden så går dina program sönder**
- Javas svar är att man för köer borde programmera mot Queue-interfacet som då gömmer allt som inte är rena kömetoder från autocomplete - men de går fortfarande att anropa och använda!
- Ett annat problem är att det inte finns något Stack-interface: bara Queue och Deque

```
Queue<Integer> queue = new ArrayDeque<>(); //Kö som går att missbruka
Deque<Integer> stack = new ArrayDeque<>(); //Kö som cosplayar en stack
Deque<Integer> queue = new ArrayDeque<>(); //Kö som inte är en kö
```

- Ett av många exempel på saker som knakar och skaver i Java: i modern programmering beter man sig inte så här

[Kodexempel]

Koden finns i klassen IntQueue.java

Stackar och köer: effektiva abstrakta datatyper

- En anledning till att både **stackar** och **köer** är så vanligt förekommande inom datorvetenskap är just att de är så tidseffektiva
- Om de är korrekt implementerade bör de **alltid ha O(1)** för insättning och uthämtning
- De är **bara intresserade av ändarna** på den underliggande datastruktur som de lagrar sin data i: de behöver inte sökning, sortering och liknande operationer som ofta har **O(logn)** eller **O(n)** i tidskomplexitet
- **De är lättviktiga:** de behöver inte några komplicerade strukturer för att lagra nycklar, hashfunktioner och liknande som krävs i **Trees** och **HashMaps**

Övningar: generalisera stacken och kön

- Förslag på övning: försök att använda generics så att **även stacken och kön vi skapade blir generiska**
- Testa även att göra ett eget interface och implementera det (med extends, trots att det är ett interface!) för den generiska typparametern
- Det här innebär att alla objekt ni försöker spara i stacken/kön måste implementera detta interface!

Fibonacci Stack Counter

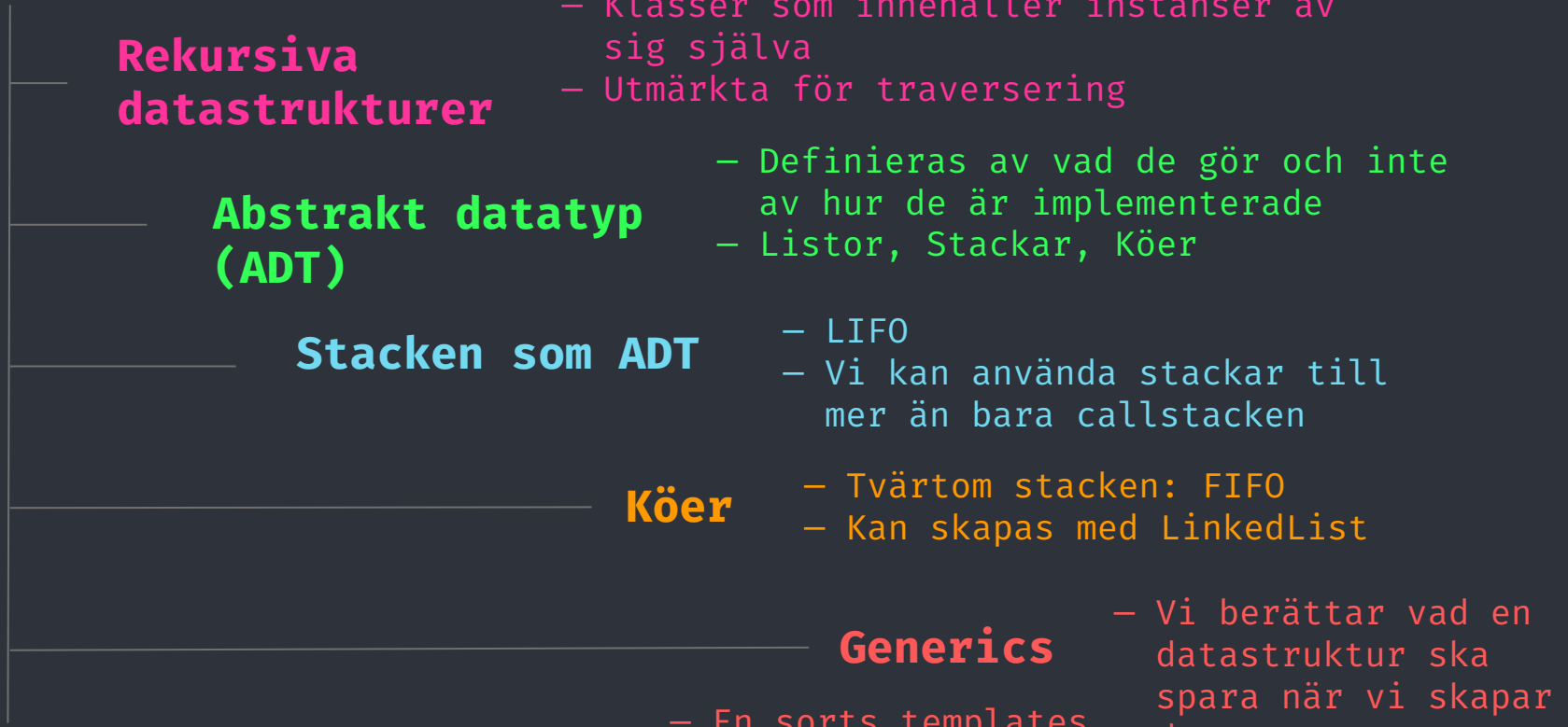
Metoden som anropas:

```
public long fib(int n)
{
   if (n ⩽ 1) { return n; }

   return fib(n-1) + fib(n-2);
}
```

`Fibonacci(6) = Fibonacci(5) + Fibonacci(4)` `fib(A) = fib(B) + fib(Q)`

Vad som händer i programmet	Vad stacken innehåller	Hur det rekursiva trädet ser ut
Pushar fib(6) med ID A på stacken.	A	A: fib(6)
Pushar fib(5) med ID B på stacken.	A, B	├── B: fib(5)
Pushar fib(4) med ID C på stacken.	A, B, C	│ ├── C: fib(4)
Pushar fib(3) med ID D på stacken.	A, B, C, D	│ │ ├── D: fib(3)
Pushar fib(2) med ID E på stacken.	A, B, C, D, E	│ │ │ ├── E: fib(2)
Pushar fib(1) med ID F på stacken.	A, B, C, D, E, F	│ │ │ │ ├── F: fib(1)
fib(1) med ID F returnerar värdet 1.	A, B, C, D, E, F	│ │ │ │ └── G: fib(0)
Poppar F från stacken.	A, B, C, D, E	│ │ │ └── H: fib(1)
Pushar fib(0) med ID G på stacken.	A, B, C, D, E, G	│ │ └── I: fib(2)
fib(0) med ID G returnerar värdet 0.	A, B, C, D, E, G	│ │ ├── J: fib(1)
Poppar G från stacken.	A, B, C, D, E	│ │ └── K: fib(0)
fib(2) med ID E returnerar värdet 1.	A, B, C, D, E	│ └── L: fib(3)
Poppar E från stacken.	A, B, C, D	│ ├── M: fib(2)
Pushar fib(1) med ID H på stacken.	A, B, C, D, H	│ │ ├── N: fib(1)
fib(1) med ID H returnerar värdet 1.	A, B, C, D, H	│ │ └── O: fib(0)
Poppar H från stacken.	A, B, C, D	│ └── P: fib(1)
fib(3) med ID D returnerar värdet 2.	A, B, C, D	└── Q: fib(4)
Poppar D från stacken.	A, B, C	├── R: fib(3)
Pushar fib(2) med ID I på stacken.	A, B, C, I	│ ├── S: fib(2)
Pushar fib(1) med ID J på stacken.	A, B, C, I, J	│ │ ├── T: fib(1)
fib(1) med ID J returnerar värdet 1.	A, B, C, I, J	│ │ └── U: fib(0)
Poppar J från stacken.	A, B, C, I	│ └── V: fib(1)

Sammanfattning av förmiddagen



Rekursiva datastrukturer

- Klasser som innehåller instanser av sig själva
- Utmärkta för traversering

Abstrakt datatyp (ADT)

- Definieras av vad de gör och inte av hur de är implementerade
- Listor, Stackar, Köer

Stacken som ADT

- LIFO
- Vi kan använda stackar till mer än bara callstacken

Köer

- Tvärtom stacken: FIFO
- Kan skapas med LinkedList

Generics

- Vi berättar vad en datastruktur ska spara när vi skapar den
- En sorts templates

Nästa tillfälle: Binära träd, grafer, garbage collection och undantagsfel

{

- Fortsätta prata om rekursiva datastrukturer
- Lära oss om binära sökträd
- Kolla på grafer och hur de fungerar
- Förstå skillnaden mellan djupet först och bredden först
- Lära oss skillnaden mellan **In-**, **Pre-** och **Post-**order
- Prata lite mer ingående om heapen och garbage collection
- Diskutera säkerhet och vad som händer när undantag kastas i ett program

}

Labbtillfälle i eftermiddag 13.15 - 15

- Kom och redovisa labbar, jobba med uppgifter, ställ frågor, osv!