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

}

Mail:
carl-johan.johansson@im.uu.se

Lite om casting

- Vi tog upp casting i går och varför det kan vara problematiskt

- Vi “bryter någonting” 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
```

- **Upcasting** (subklass till superklass) är okej eftersom **underklasser är instanser av superklassen** de ärver från

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

- Igår 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

[Kodexempel]

Koden finns i klassen Node.java

Exempel på rekursiva data-strukturer: träd, set och grafer

- Andra vanliga rekursiva datastrukturer är **Binära Träd** och **Grafer**
- Vi kommer att prata mer om dessa två om två veckor
- **Set** och **TreeMap** är ytterligare exempel på rekursiva data-strukturer
- **Stacken** (som används för att implementera en callstack) är inte rekursiv i sig men **imiterar ett rekursivt mönster**

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, reversal) **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**

[Kodexempel]

Koden finns i klassen DoubleNode.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 Structure

- 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 en specifik implementation

- 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 outputDirectory);
    void RenderFrames(Path outputDirectory);
    void CreateVideo(Path outputDirectory);
}
```

- 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 adderas till en klass med implements

```
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 {
```

Interface adderas till en klass med implements

Metod som tar en IVideoRenderer:

```
public void RenderVideo(IVideoRenderer videoRenderer, String file, String
                                                              outputDirectory)
{
    videoRenderer.ExtractFrames(file, outputDirectory);
    videoRenderer.RenderFrames(outputDirectory);
    videoRenderer.CreateVideo(outputDirectory);
}
```

I huvudklassen för programmet kan man mata in vilken slags videorenderare man vill:

```
RenderVideo(new NormalVideoRenderer(), “./samples/fish.mp4”, “./outputFiles”);
RenderVideo(new GPUVideoRenderer(), “./samples/fish.mp4”, “./outputFiles”);
RenderVideo(new MultiCoreVideoRenderer(), “./samples/fish.mp4”, “./outputFiles”);
```

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

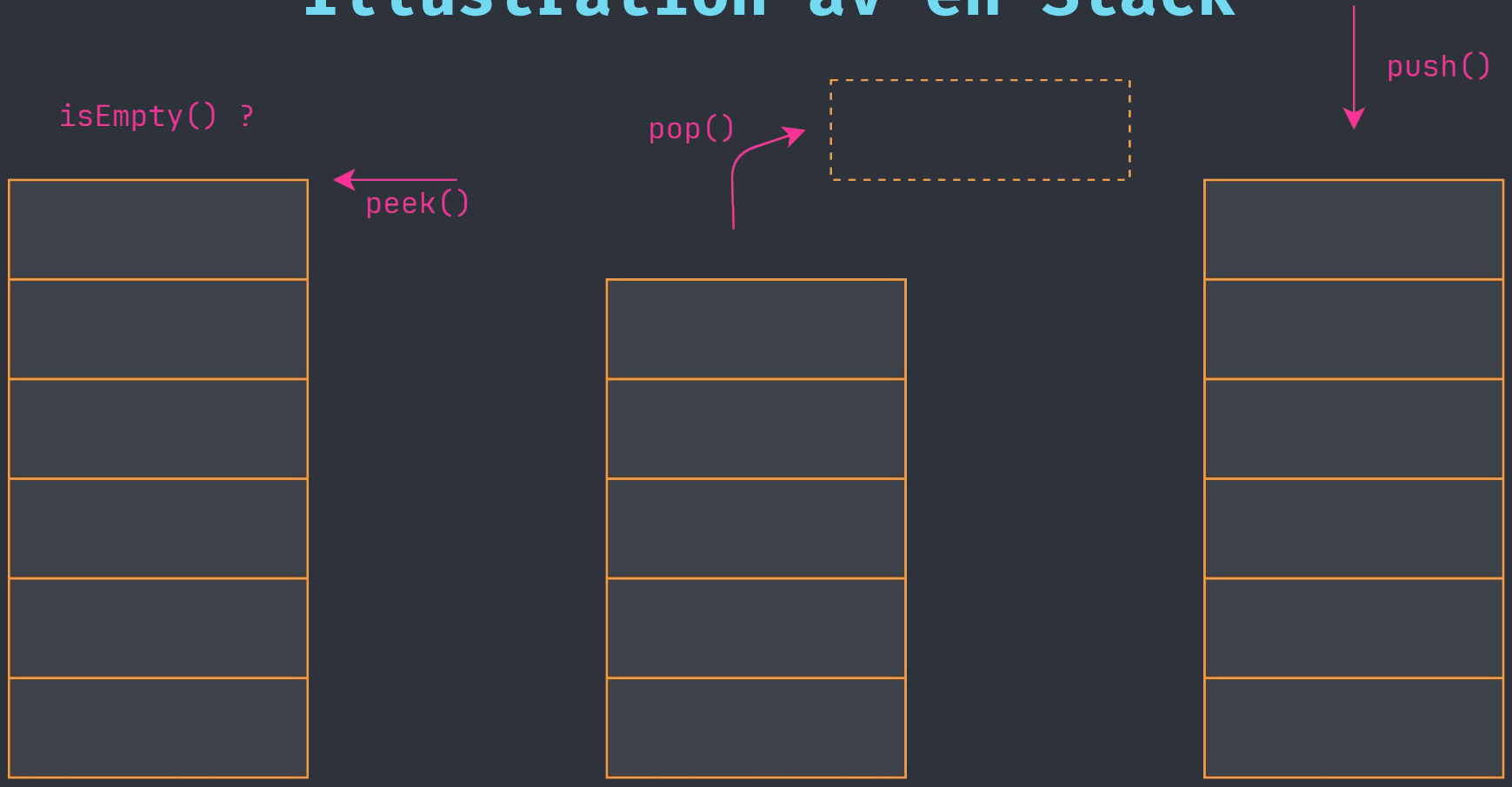
- Vi pratade om **callstacken** i går som lagrar metodanrop och primitiva datatyper medan ett program körs, men det är bara **ett exempel** på en konkret implementation av en stack

- 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

Illustration av en Stack



[Kodexempel]

Koden finns i klassen Stack.java

Generiska typer (Generics)

- **Generics** är templateklasser

- Generics innebär att man vid instansiering berättar vad man vill att en datastruktur 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 skapa vår egen generiska klass väldigt lätt:

```
public class Box<Type>
{
   private Type item;

   public Box(Type item) { this.item = item; }
   public Type getItem() { return this.item; }
}
```

[Kodexempel]

Koden finns i klassen StackGeneric.java

Varför vi kastar en generisk array

- 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
{
     private T stack;
     public Stack(int capacity)
     {
        stack = new T[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 = (T[]) new Object[capacity];   //Tillåten operation!
```

Generisk stack

- Med dessa modifikationer kan nu stacken användas för alla datatyper, och kan därmed anses vara **generisk**
- Det finns dock fortfarande ett problem: Stacken kan just nu ta **ALLA** sorters objekt vi matar in i den, men den är ju egt. bara menad att hantera numeriska typer
- Vi kan förlänga vår generiska stack 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: **Integer, Double, Long, Float, Short, Byte**

Generics är en form av polymorfism

- **Generics** är ett exempel på kompileringspolymorfism (**“compile time polymorphism”**)
- När koden kompileras ersätter Java placeholdern **Type** med den faktiska typen som datastrukturen ska innehålla
- Interfaces och abstrakta klasser är ett exempel på körtidspolymorfism (**“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

```
kaffepaus(15);
```

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”** en primitiv typ i ett objekt i stället

- Ni har använt dem varje gång ni instansierat en generisk datastruktur:

```
ArrayList<Double> list = new ArrayList<>(0);
```

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

Om wrapperklasser

- Det här är varför några av er säkert fått komplettering på Java-labbar 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 (*Thomas nämner förmodligen detta nästa vecka*)
- Inget ni behöver arbeta med utöver de användningsområden ni redan känner till: när man ska specificera typer för listor, Maps, osv

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
Kompilatordesign:	Parsa uttryck, hantera symboltabeller, osv

- Allting som kräver att man **backtrackar är fundamentalt lämpat** för stackar, och ur det avseendet liknar de rekursion

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

- En sådan här lista är 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)**

- En dubbellänkad lista innebär att vi har **access till båda ändarna**, och det är där vi vill stoppa in/plocka ut värden

Användningsområden för köer

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

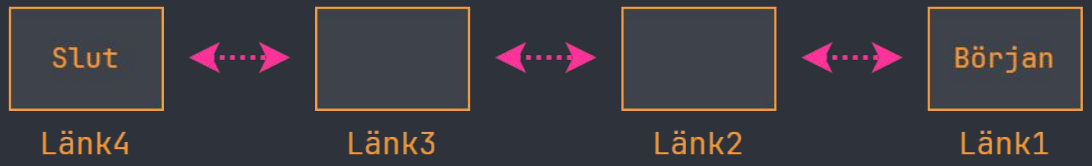
  Exempel: CPU-schemaläggning, utskriftsköer, meddelanden som skickas via Discord, matchmaking i onlinespel, buffertar för videostreaming, och-så-vidare

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

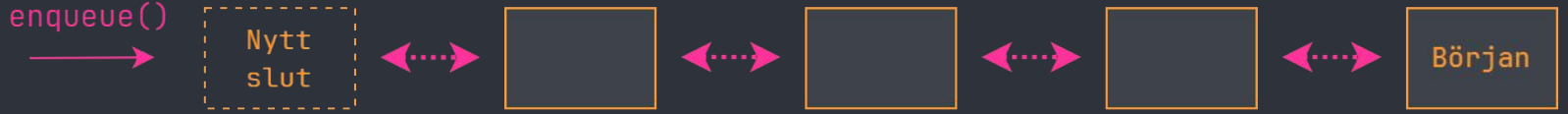
- En **dubbellänkad lista** (Javas LinkedList-klass är en sådan) har tidskomplexiteten **O(1)**, dvs **konstant tid**, för insättning, uthämtning och radering i båda ändarna

Operationer för en kö

isEmpty() ?



Lägga till i kön:



Plocka av från kön:

[Kodexempel]

Koden finns i klassen SimpleQueue.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**

Fibonacci Stack Counter

FibonacciStackCounter.java	Ett program som genererar en webbsida som visar vad som pushas på och poppas från callstacken när en rekursiv fibonaccialgoritm anropas.

Metoden som anropas:

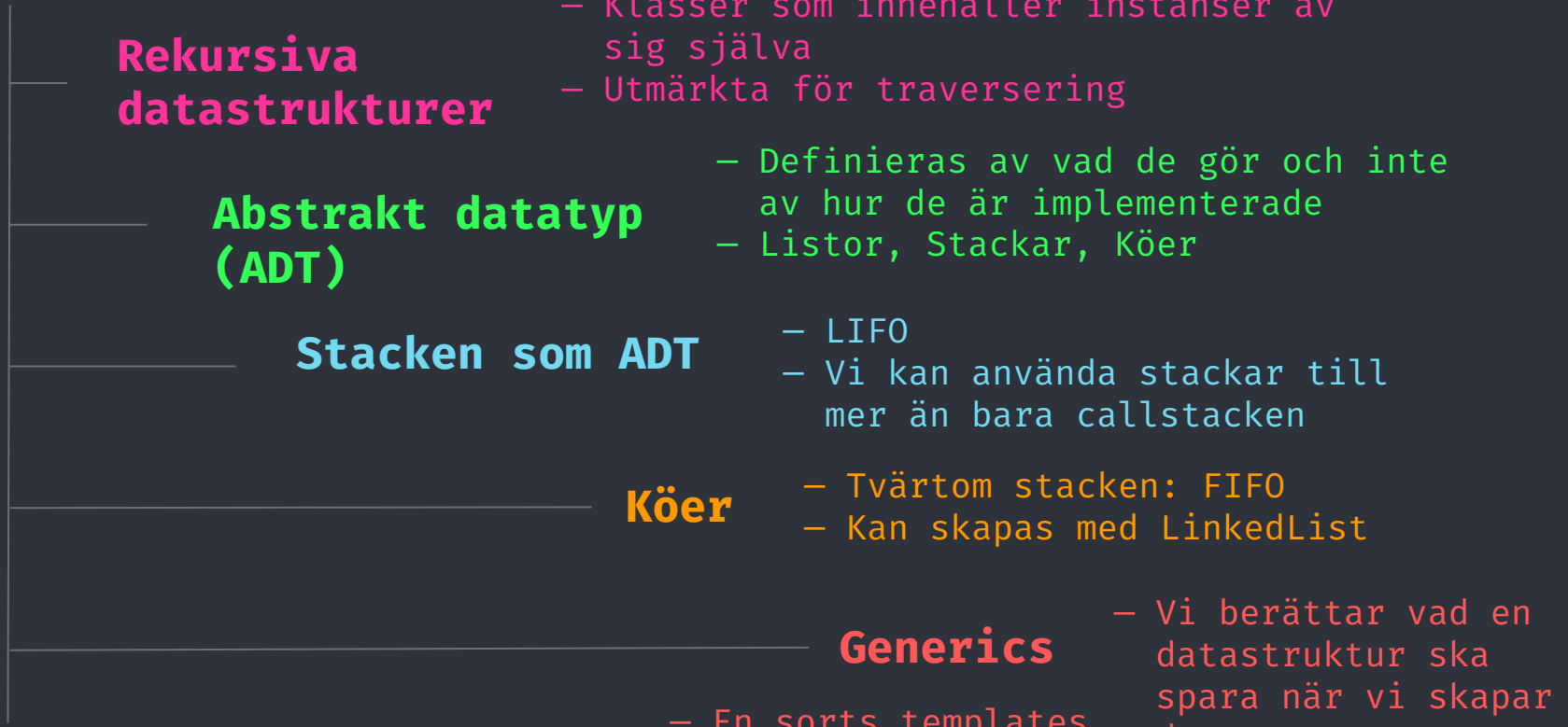
```
public long fib(int n)
{
   if (n ⩽ 1) return n;

   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)

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

{

- Thomas tar över nästa vecka och sen ses vi igen veckan därpå (v. 7). På första tillfället **12/2** kommer vi att:

- 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

- Vi har labbtillfälle i eftermiddag; utsatt tid är till 15 men jag kommer stanna kvar så länge ni vill vara här

- Både jag och Lovisa är på plats om man vill redovisa labbar eller om man har frågor om någonting vi gått genom på kursen hittills (det går bra att ställa frågor om både mina och Thomas bitar)

- Vi rekommenderar att man dyker upp och jobbar med uppgifter även om man inte har några labbar att redovisa

- Algon är en tung kurs, men de som dyker upp på labbtillfällena och ser till att ägna den utsatta tiden åt kursen kommer att klara tentan utan problem