# Rekursiva datastrukturer, Generics och abstrakta datatyper

{

Vi ska:

- Prata om abstrakta datatyper
- Kika på Generics
- Studera binära träd
- Fortsätta prata om rekursivitet
- Lära oss hur rekursiva anrop traverserar
- Prata mer detaljerat om stacker och köer
- Kolla på grafer och hur de fungerar
- Förstå skillnaden mellan djupet först och bredden först
- Lära oss skillnaden mellan In-, Pre- och Post-order

}

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

# ADT (Abstract Data Type)

- En abstrakt datatyp är en datastruktur 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, högnivå har fler lager av abstraktion
- Exempel på datastrukturer som är abstrakta: stackar, listor, Köer, kartor ... det mesta som inte är direktmanipulering av minne

# 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. En sträng är egt. bara en pekare till en char-array. I Java är String dock en klass och ingen primitiv datatyp såsom int, double, bool, etc även om den ofta buntas samman med dem

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

ADT.java

StackADT.java

# Stacken som ADT

- Vi pratade om callstacken i går som hanterar activation frames för metodanrop, 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

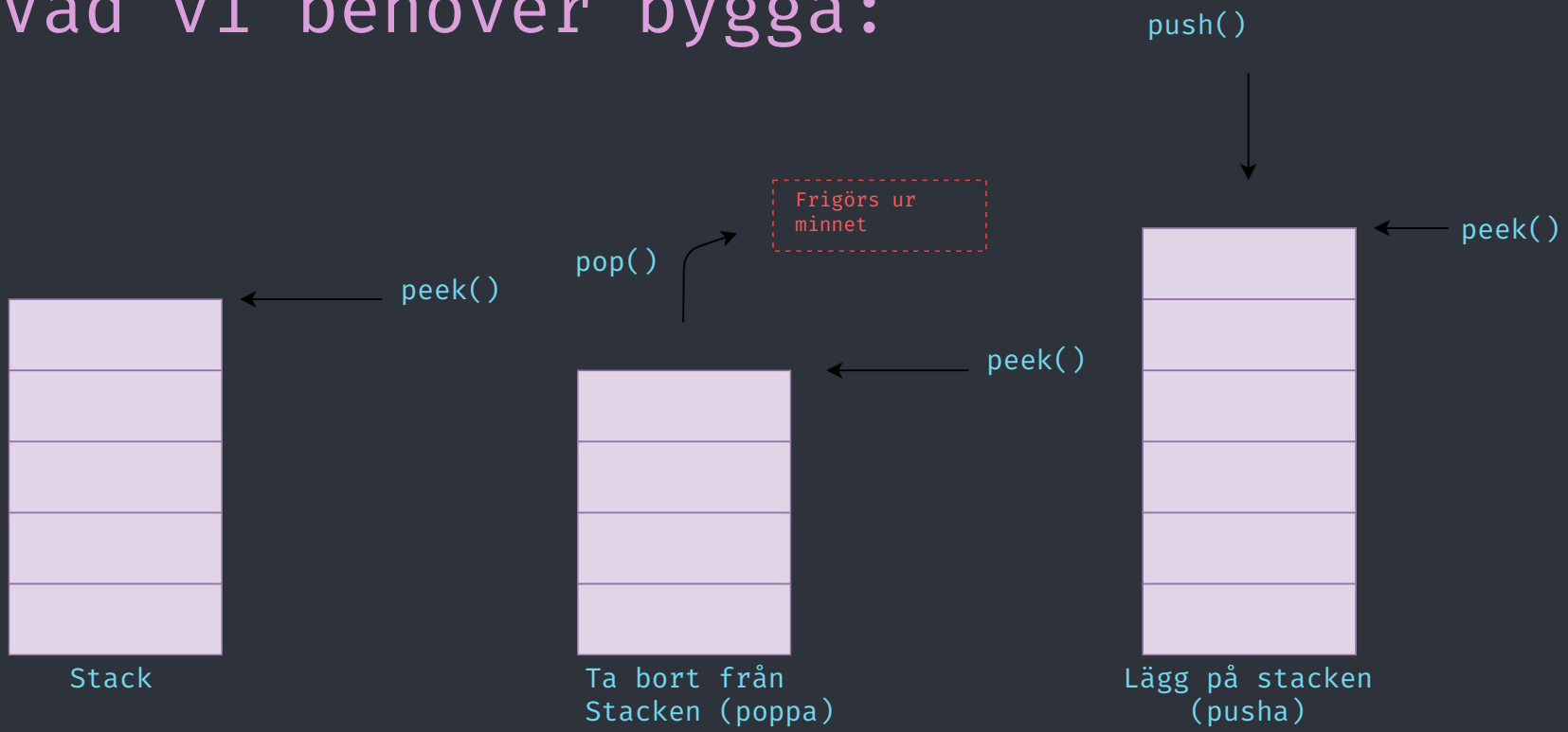
- Alla klasser du skapar som har den här funktionaliteten kan kallas för en stack

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

ADT.java

StackOverview.java

# Vad vi behöver bygga:



# [Kodexempel]

Koden finns i klassen **Stack**.java

# Generiska typer (Generics)

- Generics är en sorts templates
- 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<>();
```

- Kan vi modifiera vår stackklass så att den blir generisk i stället för att vara låst till integers?

# Generisk stack

- Stacken vi skapat kan nu användas för alla datatyper, och kan därmed anses vara generisk!

- Notera att vi dock måste använda wrapperklassen för den primitiva typ som ska lagras när vi instansierar en generisk datastruktur:

```
Stack<Integer> stack = new Stack<>(0);
ArrayList<Double> list = new ArrayList<>(0);
 ... osv.
```

- Anledningen är att generiska typer kräver objekt. En int är inte ett objekt. Javas lösning är att skapa wrapperklasser som “slår in” en primitiv typ i ett objekt i stället

# Binärt sökträd (BST)

- Ännu ett exempel på en abstrakt datatyp
- En datastruktur som är rekursiv till naturen
- Sökning i trädet är ett exempel på divide-and-Conquer, precis som binär sökning
- Till skillnad från binär sökning behöver vi dock inte ha datan sorterad: trädet sorterar den automatiskt åt oss när vi lägger in den
- Insättningsoperationen har tidskomplexiteten O(logn)

# Trädstruktur

– Kallas träd eftersom de förgrenar sig via noder:

# Lite kort om referenser

- En referens är en variabel som lagrar den första minnesadressen till en datasamling: en objektsinstans, en array, en lista …
- Referensen är 8 byte i minnet på 64-bitarssystem oavsett hur stor datasamling den pekar mot
- I en länkad lista är varje nod sparad på olika platser i minnet, och varje nod innehåller en referens till nästa nod
- Därför har den O(n) för uthämtning: man måste iterera genom hela listan och följa noderna

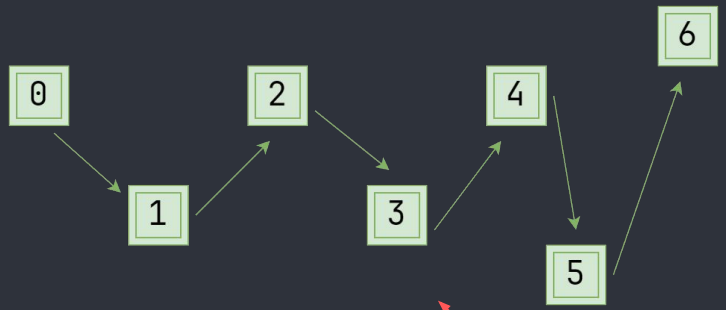
References.java

LinkedLists.java

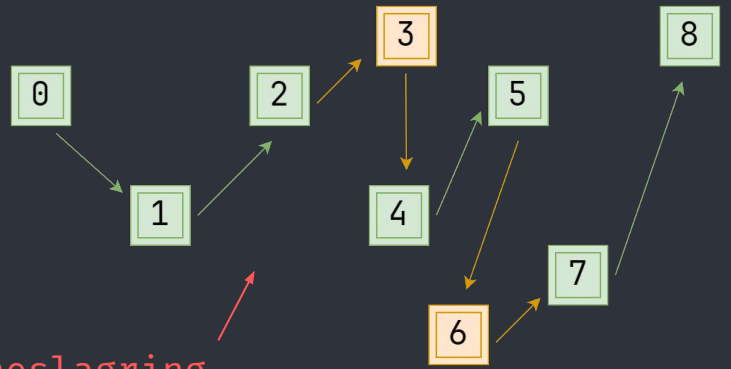
Allting lagrat i följd i samma minnesblock

Array:


LinkedList:




Samma LinkedList med två nya noder:




Ostrukturerad minneslagring

Nummer = indexplatser

References.java

UnbalancedBalanced.java

# Obalanserat vs balanserat träd

- 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 ena förgreningen:

```
10
 \
  20
   \
    30
     \
      40 … osv
```

- Ett balanserat träd, också kallat Red and Black Tree, har tidskomplexiteten O(logn) för uthämtning

# Rödsvarta träd

- Red and Black Trees är 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 klasser i Java med balanserade rödsvarta träd:

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

- En utmärkt datastruktur för stora datasamlingar

# [Kodexempel]

Koden finns i klassen **BinaryTree**.java i zipfilen

RedAndBlackTrees.java

Graphs.java

# Grafer

-

Graphs.java

DepthVsBreadth.java

# Djupet först vs Bredden först

– Grafer

# Kortaste vägen i en graf

- Dijkstras algoritm hittar den kortaste vägen i en graf

  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

Dijkstra.java

Greedy.java

# Giriga algoritmer

- Greedy 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.

-

Greedy.java

GraphExample.java

# Grafexempel

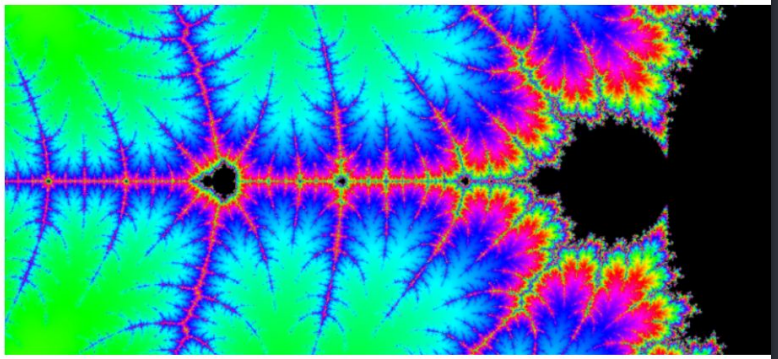
_

# Plocka hem repo från Github:

https://github.com/carljohanj/Recursive_lectures

README

## Rekursion, minne och abstrakta datatyper



### Innehåll

- Kodexempel
- Föreläsningsslides
- Andra filer
- Resurser

### Kodexempel

| | |
|---|---|
| RecursiveFibonacciDynamic.java | En rekursiv fibonaccialternativ som använder en dynamisk programmeringsteknik för att effektivisera algoritmen. |
| RecursiveFibonacciStackCounter.java | Ett program som genererar en webbsida som visar vad som pushas på och poppas från callstacken när en rekursiv fibonaccialgoritm anropas. |
| Stack.java | En implementation av en stack som använder Generics för att man ska kunna lagra olika typer av data i den. |


**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) |

```
return 0;
```

# Nästa vecka: Repetition plus labbtillfälle

{

- Repetera översiktligt vad vi gått genom under kursen hittills, plus lite nya saker
- Vi ska gå genom lösningarna till dugga 1
- Öppen frågestund. Ställ alla frågor som ni varit rädda för att ställa. Det finns inga dumma frågor!
- Laborationstillfälle på eftermiddagen - dyk upp för att jobba med uppgifter. Både jag och Lovisa är på plats om man behöver hjälp

}

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