Dagens föreläsning: Stackar, köer, abtrakta 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 detaljerat om stacken som datastruktur
- Undersöka vad som händer på callstacken när vi anropar en rekursiv fibonaccialgoritm

}

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

Rekursiva datastrukturer

- Vi har diskuterat 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
- Datastrukturer definierar var och hur data ska lagras medan en algoritm opererar på datastrukturer för att utföra ett arbete
- Vi pratade om algoritmer som anropar sig själva i går, men klasser kan även innehålla instanser av sig själva

[Kodexempel]

Koden finns i klassen **Node**.java

Rekursiv datastruktur: 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 det här
- 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

Träd, grafer, set och kartor

- Andra vanliga rekursiva datastrukturer är Träd och Grafer
- Precis som en länkad lista är de uppbyggda av noder, men de är lite mer komplexa *(Vi kommer att prata mer om dessa två strukturer om två veckor)*
- Set och TreeMap är ytterligare exempel på rekursiva datastrukturer
- Stacken (som används för att implementera en callstack) är inte rekursiv i sig men imiterar ett rekursivt mönster

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 ... det mesta som inte är direktmanipulering av minne eller använder sig av simpla underliggande datastrukturer

ADT.java NotADT.java

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”

NotADT.java

ArrayList.java

Är ArrayList en abstrakt datatyp?

- ArrayList och LinkedList är specifika implementationer av List, som är en abstrakt datatyp

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

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

- 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 det ser ut

- Vi skiljer mellan en abstrakt datatyp och en datastruktur, även om gränsen ibland är suddig

Stacken som ADT

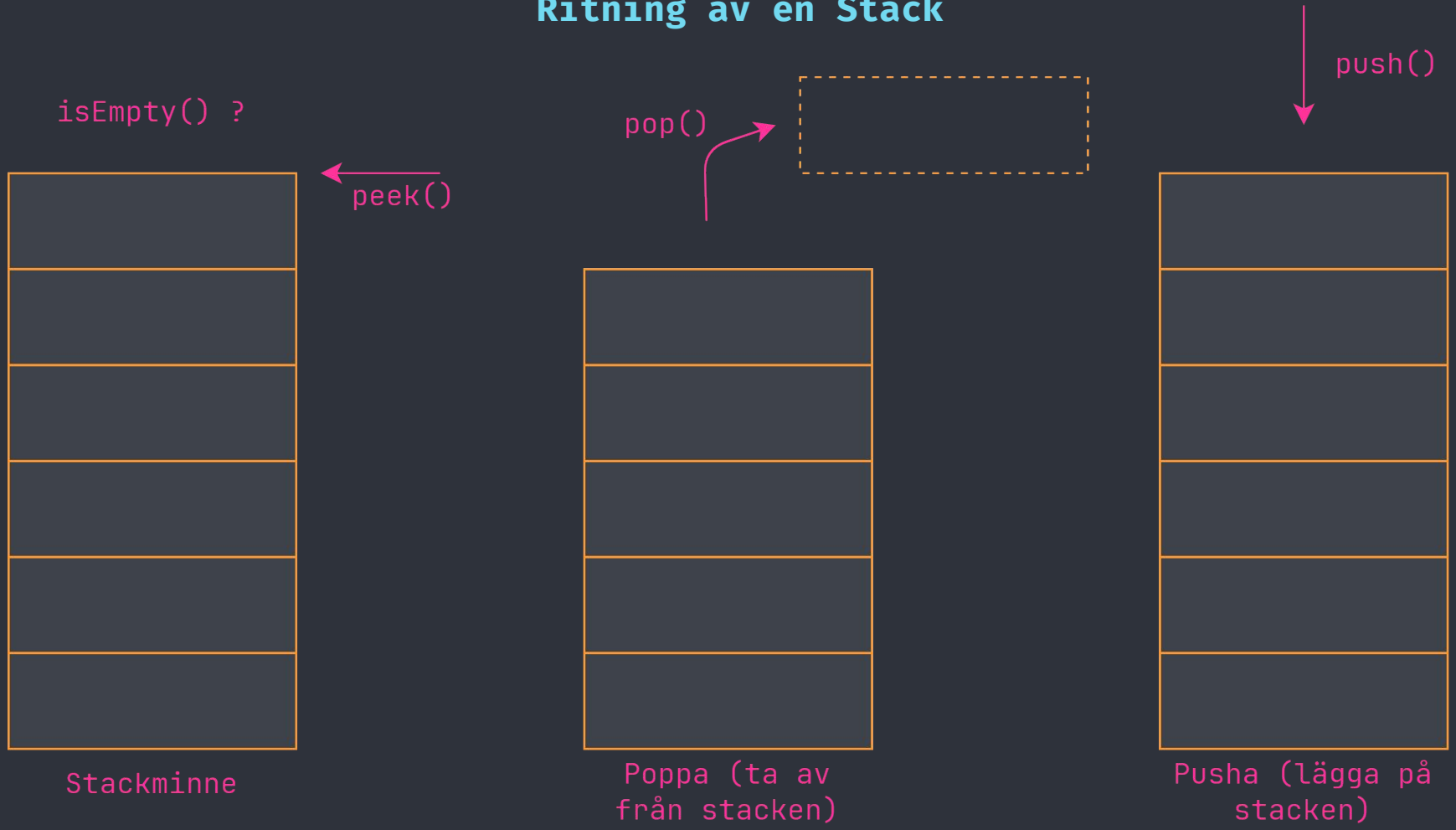
- Vi pratade om callstacken i går som hanterar activation frames för att lagra metodanrop, men det är bara ett exempel på en konkret implementation av en stack

- 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 följaktligen kallas för en stack

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

Ritning av en Stack



[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?

[Kodexempel]

Koden finns i klassen **StackGeneric**.java

Generisk stack

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

- Det finns dock ett problem: Stacken kan just nu ta ALLA sorters objekt vi matar in i den, men den är 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

Vad är en wrapperklass?

- 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);
```

- Vill man veta mer om varför de behövs kan man googla på “type erasure”, men det är väldigt mycket överkurs

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)!

StackTimeComplexity.java

StackUsage.java

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

StackUsage.java

Queue.java

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), precis som när du står i kö för att köpa lunch: den som står först får mat först

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

Queue.java

QueueUsage.java

Användningsområden för köer

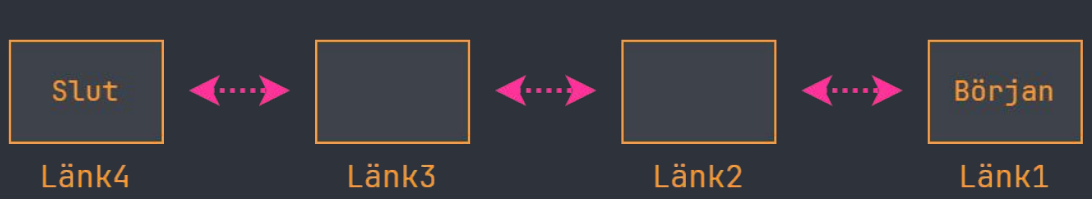
- Köer används ofta i sammanhang där sekvenser är viktiga
  Exempel: CPU-schemaläggning, utskriftsköer, meddelanden som skickas via Discord, matchmaking i onlinespel, buffertar för videostreaming, och-så-vidare

- 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

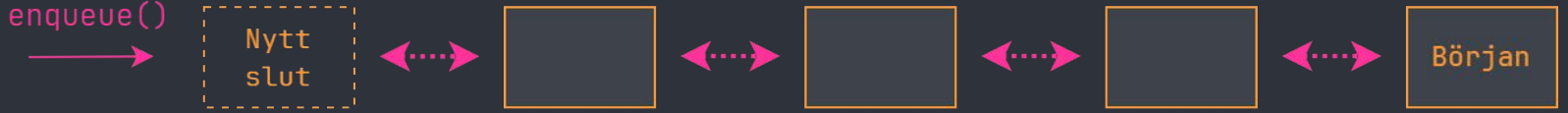
- Vi behöver bara jobba med ändarna på listan i en kö, vilket gör den oerhört tidseffektiv

Ritning av en Kö



isEmpty() ?

Lägga till i kön:



Plocka av från kön:

[Kodexempel]

Koden finns i klassen **SimpleQueue**.java

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 effektivt 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

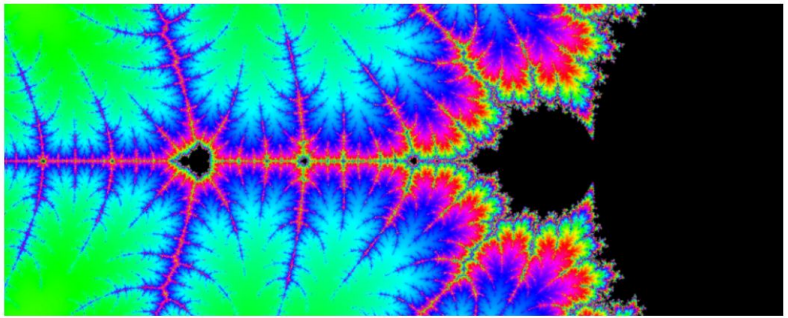
Hur callstacken förändras under körtid

- Vi ska kolla mer ingående på vad som händer med stacken när vi gör rekursiva fibonaccianrop
- I går pratade vi om activation frames och varför stora mängder av metodanrop kan leda till en stacköverfyllnad (ett Stack Overflow)
- Men hur förändras stacken när vi anropar en rekursiv fibonaccialgoritm egentligen?
- För att illustrera det här ska vi använda: FibonacciStackCounter™

https://github.com/carljohanj/lectures

README

Rekursion, minne och abstrakta datatyper



Innehåll

- Kodexempel
- Föreläsningsslides
- Uppgifter och andra filer
- IntelliJ-plugins
- Resurser
  - Länkar

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)
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 tillfälle: Binära träd och grafer

{

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

- Fortsätta prata om rekursivitet
- 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

}

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