

Dagens tillfälle: Repetition och övningstentor

{

- Repetition av tidskomplexiteter, rekursivitet, sorteringar, maps, hashmap, abstrakta datatyper och grafer
- Genomgång av gamla tentafrågor
- Ni är välkomna att ställa frågor om ni har några!

}

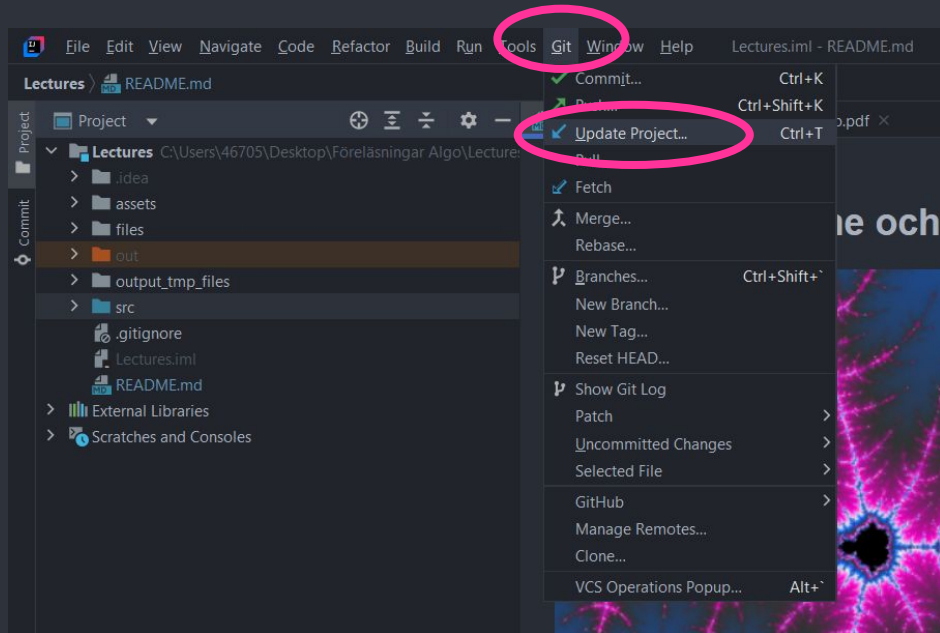
Mail:

carl-johan.johansson@im.uu.se

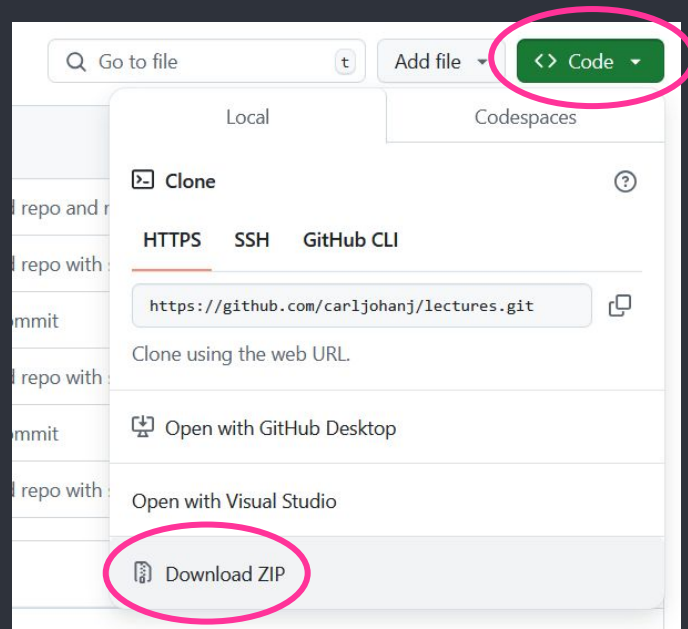
&

lovisa.m.johansson@im.uu.se

Uppdatera repository för att hämta dagens material



Meny > Git > Update project ...

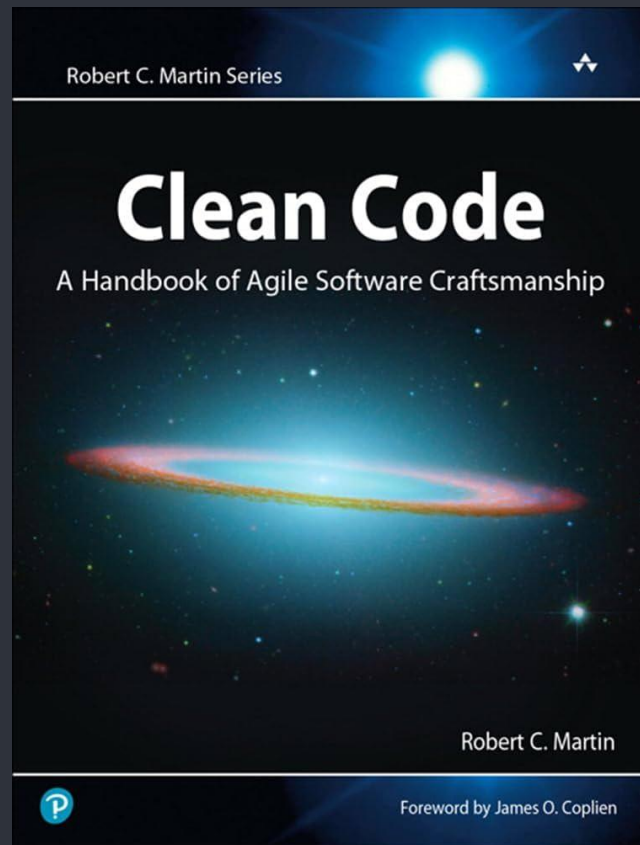


Går också bra att ladda ner en färsk zip från Github:
<https://github.com/carljohanj/lectures>

Boktips: Clean Code

"I was struck by how small all the functions were. I was used to functions in Swing programs that took up miles of vertical space. Every function in this program was just two, or three, or four lines long. Each was transparently obvious. Each told a story. And each led you to the next in a compelling order. That's how short your functions should be!"

- Vi pratade lite kort om **best practices** senast och tog upp att bra kod bör vara **kort och "clean"**
- En bok som är **ovärderlig** om man vill bli en bättre programmerare är **Clean Code** av Robert Martin



Tumregel för tidskomplexiteter

När vi pratar om “n” menar vi alltid datamängden som vi matar in i algoritmen:

Skickar vi in en lista är n antalet värden i listan, osv

Om tiden:

... är det:

... som betecknas:

Dubblas när datamängden dubblas

Fyrdubblas när datamängden dubblas

Åttadubblas när datamängden dubblas

Dubblas varje gång **n ökar med 1**

Linjär tidskomplexitet

Kvadratisk tidskomplexitet

Kubisk tidskomplexitet

Exponentiell tidskomplexitet

$O(n)$

$O(n^2)$

$O(n^3)$

$O(2^n)$

}

Dessa kallas även för polynom tidskomplexitet
men vi gör skillnad mellan dem eftersom de är
olika grader av hemskhet 🐱 $O(n^2)$ är ofta en giltig komplexitet

Om ni däremot får tidskomplexitet som är $O(n^3)$ (eller värre) bör ni försöka
konstruera en bättre lösning

Tidskomplexitet och platskomplexitet

- **Tidskomplexitet** är ett mått på **hur tiden utvecklas** när indatan i en algoritm ökar
- **Platskomplexitet** är ett mått på **hur minneseffektiv en algoritm är**: hur mycket plats i RAM-minnet upptar den?
- Algoritmer med **dålig platskomplexitet** är ofta **oönskvärda** även om de kan ha bra tidskomplexitet eftersom **minne är en dyr resurs** i programmering
- **Undvik att kopiera** över saker i mellanliggande listor och liknande om möjligt, särskilt när datamängden är stor

ADT: abstrakt datatyp

- En ADT är en Abstrakt DataTyp, dvs en datatyp som **inte definieras av en specifik implementation** (vi menar kod när vi säger implementation) **utan av ett beteende**
- Den kan vara **skriven på flera olika sätt** och använda olika data-strukturer “under huven” så länge den **uppfyller specifika krav** på hur den beter sig
- En **Lista ska till exempelvis vara dynamisk**, till skillnad från en simpel array: vi ska alltid kunna stoppa in fler värden i den utan att få slut på utrymme
- **Exempel:** Både **ArrayList** och **LinkedList** är olika sätt att implementera en Lista (de har sina egna implementationer av de metoder som Javas List-interface säger att de måste innehålla)

Vanliga abstrakta datatyper

ADT	Definition	Implementationer i Javas standardbibliotek:
Lista	Ordnad samling av element	ArrayList, LinkedList
Stack	LIFO: Last in, First Out	Deque
Kö	FIFO: First in, First Out	Queue, PriorityQueue
Träd	Hierarkiskt lagrad data	TreeSet (Red and black tree)
Map	Lagrar nyckel-värde-element	TreeMap, HashMap, LinkedHashMap
Graf	Lagrar relationer mellan data	- (Använd tredjepartsbibliotek som t.ex. Jgraph som finns gratis på Github)

– Det finns fler, men dessa är de som vi gått genom på kursen. Ni behöver inte känna till några andra

Vad är en Map?

- Vi kallar den här datatypen för en Map eftersom **den “mappar” en nyckel** till ett specifikt värde. Med “mappa” menar vi att vi **associerar dem med varandra**: varje värde får en unik nyckel
- Det har alltså **inget att göra** med en **karta**, även om man ibland ser “karta” som svensk översättning av “Map”
- Att “mappa” kommer från matematiken. En matematisk funktion är en formel där vi stoppar in ett värde och får ut ett annat (“F av x är lika med y”):

$$f(x) = y$$

Man säger då att funktionen $f(x)$ ovan **mappar x till en output y**

- På samma vis **mappar vi nycklar till värden i en Map**: vi stoppar in nyckeln x och får ut värdet y

```
hashmap.get(x);    //Returnerar värdet associerat med x
```


HashMap

- En HashMap är en variant av Map (det vill säga: den **implementerar Map-interfacet**) i Java. Det är en datatyp som vi använder för att **lagra nyckel-värde-par** i en underliggande array
- Den hashar först nyckeln: det här innebär att den översätter nyckeln **till en integer** när vi stoppar in den
- Efter att den räknat ut en hashkod **behöver den dock komprimera** det här numret, för det är oftast större än högsta indexplatsen i arrayen
- **Den komprimerar hashkoden** antingen genom att använda en modulo eller att göra en binär bitwise AND-jämförelse med storleken på arrayen (om båda bitarna är 1 blir resultatet 1, annars 0)
- Den **komprimerade hashkoden** blir indexplatsen där den sedan stoppar in nyckel-värde-paret

Vad händer om två insättningar har samma hashkod?

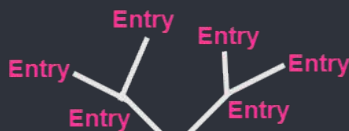
- **Hashkoder är inte unika:** två olika koder kan ha (och producerar ofta) samma index eftersom de komprimeras
- En indexplats i en HashMap kallas för en **“bucket”**: namnet är menat att indikera att den kan innehålla **mer än ett** nyckel-värde-par
- Om en hink bara innehåller ett nyckel-värde-par lagras de **i ett Entry-objekt**
- **Om två eller flera** nyckel-värde-par skulle få samma arrayplats skapar HashMapen **antingen en länkad lista** eller **ett binärt träd** och sorterar in Entry-objekten där i stället
- Sökning/uthämtning **blir då $O(n)$** (för lista) **eller $O(\log n)$** (för träd)
- Därför säger vi att en HashMap **har $O(1)$ i genomsnitt**

Array kontra HashMap








Array

45	108	2455	36	902	96	1045
Index 0	Index 1	Index 2	Index 3	Index 4	Index 5	Index 6
0(1)	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)

Binärt träd



HashMap

						
Index 0	Index 1	Index 2	Index 3	Index 4	Index 5	Index 6
$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$	$O(n)$

Länkad lista

För- och nackdelar med HashMap

- Det här är **varför en HashMap är snabb** på uthämtning: i stället för att loopa genom alla element räknar Java ut ett nummer och **går direkt till det index** i arrayen **som motsvarar det numret**
- En HashMap fungerar på samma vis **som en ArrayList när den blir full**: den skapar då en ny underliggande array, rehashar och kopierar över all sin data i den nya arrayen, och tar sedan bort den gamla
- Det här är kostsamt både tidskomplexitetsmässigt (**$O(n)$ för att kopiera över all data**) och platskomplexitetsmässigt (vi får en dubbelt så stor array)
- Den **sorterar inte data automatiskt** och **bibehåller heller inte insättningsordningen**: hashfunktionen bestämmer var i arrayen som datan hamnar

HashMap eller TreeMap?

- Båda är **utmärkta datatyper** för att lagra **nyckel-värde-par**
- **HashMap** har **$O(1)$ i genomsnitt** för uthämtning och insättning
- **TreeMap** har **alltid $O(\log n)$**
- **HashMap** blir dock mindre effektiv, både tids- och platsmässigt, när den **blir full**, när den **behöver förstora sig själv** och när den **behöver sorteras**
- **TreeMap sorterar allting** man lägger in i den **direkt** medan HashMap kräver sortering efter att den populerats om man vill ha värdena i ordning
- Ofta är det **sorteringsbehovet** man väljer efter: behöver man värden i en viss ordning vill man helst att datatypen **fixar det snabbt vid insättning**

Sortering

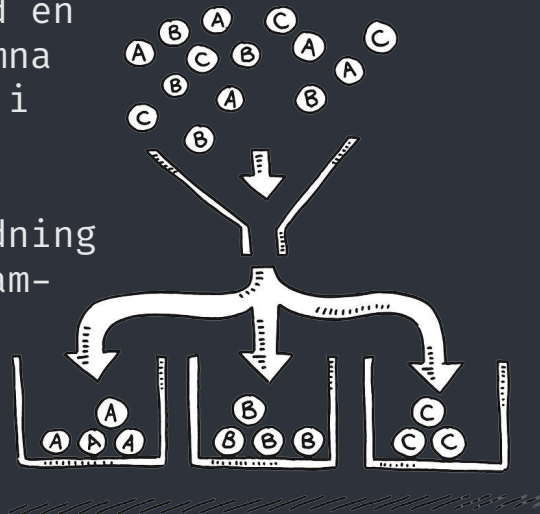
- Sorteringar har **två egenskaper**: de kan vara stabila och in-place
- Med en **stabil sorteringsalgoritm** menar vi något som bibehåller den relativa ordningen mellan objekten

Exempel: [(Alice, 25), (Bob, 30), (Charlie, 25)]

Om vi sorterar den här samlingen efter ålder med en stabil sorteringsalgoritm kommer Charlie att hamna före Bob **men inte Alice** eftersom Alice låg före i originalsamlingen

Instabil sortering garanterar ingen inbördes ordning mellan objekten utöver att de lägsta åldrarna hamnar först

- **In-place innebär** att en samling sorteras utan att sorteringsalgoritmen skapar en ny samling för att lägga datan i



Jämförelse, olika sorteringsalgoritmer

Algoritm	Tidskomplexitet (Bäst/Sämst)	In-Place	Stabil	Används
QuickSort	$O(n \cdot \log n)$ / $O(n^2)$	Ja	Nej	Stora dataset som behöver in-place-sortering
MergeSort	$O(n \cdot \log n)$	Nej	Ja	När stabilitet är viktigt
InsertionSort	$O(n)$ / $O(n^2)$	Ja	Ja	Små eller mestadels sorterade samlingar
BubbleSort	$O(n^2)$	Ja	Ja	Helst inte, men ok för små dataset
SelectionSort	$O(n^2)$	Ja	Nej	Används sällan men föredras över BubbleSort pga färre swaps

- Giltiga algoritmer
- Används främst i lärosyfte

Vad använder Javas automatiska sorteringsfunktioner?

- **Arrays.sort()** använder **MergeSort** för att sortera objekt, men implementationen är egentligen en optimerad algoritm som heter TimSort
- **TimSort** är en **hybrid av MergeSort och InsertionSort**. Om en samling är liten eller delvis sorterad kommer den köra InsertionSort; om den är stor och/eller mestadels osorterad kommer MergeSort köras
- TimSort har $O(n)$ i bästa fall, $O(n \cdot \log n)$ i genomsnitt, och även $O(n \cdot \log n)$ i värsta fallet
- **Collections.sort()** använder **TimSort** för att sortera listor
- **QuickSort** används av **Arrays.sort()** för att **sortera primitiva datatyper** (int, double, osv). Java har en optimerad algoritm kallad för Double-Pivot QuickSort som används

Array eller lista?

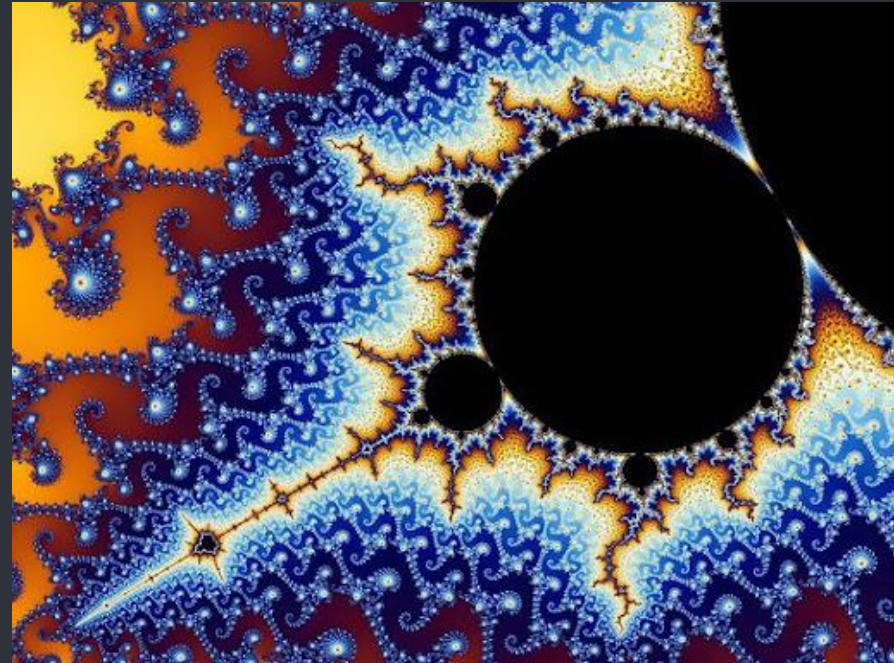
- **Arrays.sort()** är till för arrayer. Den kan sortera arrayer som innehåller:
 - * Primitiva typer: QuickSort används
 - * Objekt: TimSort används
- **Collections.sort()** är till för listor. Den är alltid stabil men aldrig in-place. Exempel på listor som den kan sortera:
 - * ArrayList
 - * LinkedList
- Collections.sort() **funkar bara för listor**. Vill man sortera t.ex. en **HashMap** efter nycklar eller värden brukar man:
 1. **Konvertera** den till en List<Map.Entry<K, V>> och sortera listan
 2. Konvertera den **till en TreeMap**, som då sorterar automatiskt efter nycklar

Rekursivitet

- **Rekursion** är när någonting definieras i termer av sig självt. I programmering menar vi:
 - * **En metod** som anropar sig själv (**rekursiv algoritm**)
 - * **En klass** som innehåller instanser av sig själv (**rekursiv datastruktur**)
- Rekursion finns överallt, inte bara i programmering. Naturen och matematiken är uppbyggda av självrefererande system
- Exempel på rekursiva algoritmer: MergeSort, binär sökning, fibonacci - divide-and-conquer-problem är unikt lämpade för rekursion
- Exempel på datastrukturer som är rekursiva: Länkade listor, binära träd, grafer

Rekursivitet

- **Rekursion använder sig av callstacken:** varje nytt anrop lägger en ny stackframe på callstacken och när en metod **nått basfallet** börjar stacken att lindas upp
- Det är därför rekursiva algoritmer och strukturer **har ett backtrack-ingbeteende**
- **Kan skapa ett Stack Overflow** om de saknar ett basfall eftersom de då fortsätter att lägga på stacken i all oändlighet



Rekursiva exempel

Rekursiv klass:

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

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

- Innehåller två instanser av sig själv
- Går att bygga både träd och länkade listor med

Rekursiv algoritm:

```
public long fibonacci(int n)
{
    if(n <= 1) { return n; }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

- Gör två nya anrop till sig själv om den inte når basfallet
- Just rekursiv fibonacci är en väldigt ineffektiv algoritm eftersom den växer exponentiellt
- Går dock att optimisera med dynamisk programmering

Algoritmdesigntechniker

- Det finns lite olika algoritmdesigntechniker man kan använda. Designtechnik och “paradigm” syftar till samma sak: det är bara namn som beskriver ett beteende. Några vanliga exempel är:

Teknik	Definition:	Exempel:
Brute force	Testar alla möjliga kombinationer (ineffektiv men ibland nödvändig)	Nästlade for-loopar
Divide-and-conquer	Delar upp ett problem i delproblem (naturligt lämpade för rekursion)	MergeSort, Quick-Sort, Binär sök
Greedy	Tar alltid det lokalt bästa beslutet (Ändrar sig aldrig)	Dijkstras algoritm
Backtracking	Utforskar alla möjliga vägar och kan ångra felaktiga beslut (mer effektiv än en brute force-approach)	Djupet-först i graf och pathfinding

Grafer

- **En graf** är en datastruktur som man använder **när relationerna** mellan datan **är lika viktiga** att lagra **som datan själv**
- Precis som träd, listor, stackar och köer är det **en abstrakt data-typ** som kan skapas på olika vis: den definieras utifrån vad den gör och inte hur den är implementerad
- Grafer är **inte särskilt värdefulla för att lagra linjär data**: det vill säga primitiva datatyper (typ integers), objekt, nyckel-värde-par, och liknande. Man använder dem bara **om det finns meningsfulla relationer** mellan dessa nummer eller objekt

Att rita en graf

– Genom att rita smart blir det lättare att besöka noderna i en viss ordning

– Om A har en koppling till B men B inte har någon koppling till A betyder det att riktningen på kanten pekar mot B

– Om både A och B har en koppling till varandra så är riktningen åt båda håll, alltså ingen alls eftersom de tar ut varandra

A → B → D

B → A → C → E

C → B → F

D → A

E → B → F

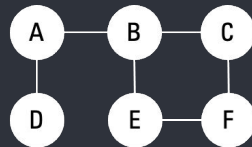
F → C → E

– Börja med att rita alla noder:



– Nästa steg är att rita kanterna

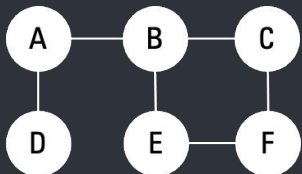
– Eftersom exempelvis A har en kant till B och B har en kant till A så finns här ingen riktning att utläsa



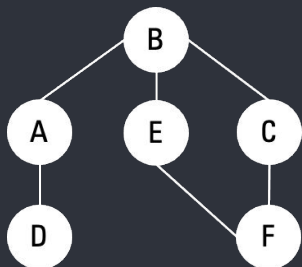
Anpassa grafen

- Sätt utgångsnoden högst upp
- Ordna noderna i hierarkisk ordning (ex. alfabetisk) om möjligt

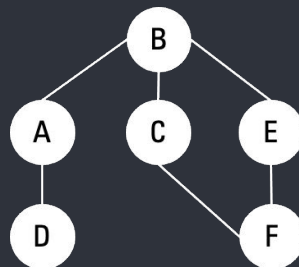
Grafen från början:



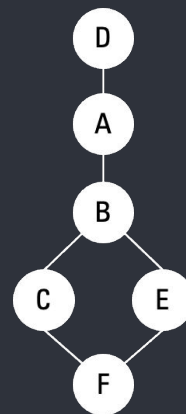
Grafen efter du fått veta att du ska utgå från noden B:



Grafen efter att du fått veta att vägen till andra noder lagras i bokstavsordning:



Grafen efter att du fått veta att du ska utgå från en annan nod, nämligen D:



Genomgång av gamla tentamenfrågor

ADT (VT23)

Fråga: Du har en okänd datastruktur och du vill ta reda på vilken datastruktur det är. För att undersöka saken lagrar du i tur och ordning följande data i din datastruktur:

2, 4, 6, 7, 10, 12, 10, 8, 6, 4, 2

När du hämtar ut data från din datastruktur får du, i tur och ordning:

2, 4, 6, 8, 10, 12, 10, 7, 6, 4, 2.

Vilken datastruktur är det?

Lösning: ADT (VT23)

Kandidater: Lista, HashMap, Träd, Kö, Stack

Beteende: Vi tycks få ut datan i exakt omvänd ordning mot hur vi stoppade in den.

Listor lagrar värden i den ordning vi stoppar in dem. Träd sorterar datan automatiskt åt oss. En HashMap lagrar ingenting i en särskild ordning alls. Det kan inte vara någon av dessa.

En kö fungerar enligt principen FIFO: **First in, first out.** Vi borde med andra ord fått ut värdena i samma ordning som vi stoppade in dem.

En stack däremot fungerar enligt LIFO-principen: **Last in, first out!** Eftersom vi staplar ("stackar") saker på hög måste vi börja ta av dem från toppen igen när vi vill hämta ut dem.

Svar: Den abstrakta datatypen **borde vara en stack**, eftersom vi får ut värdena i omvänd ordning.

ADT (HT23)

Fråga: Följande operationer görs på en inledningsvis tom kö:

`enqueue(5)`, `enqueue(8)`, `dequeue()`, `enqueue(4)`, `enqueue(3)`, `size()`

Vilket data får vi när vi nu utför operationen `dequeue()`?

- En kö fungerar enligt principen **FIFO: First In, First Out**
- Operationerna gör följande: `enqueue()` lägger på kön, `dequeue()` hämtar ut från kön, `size()` ger oss hur många element kön innehåller men förändrar inte kön
- Vi lägger 5 och 8 i kön. Vi tar sedan bort det första elementet (5). 8 är nu först i kön. Vi lägger till 4 i kön, och sen 3. Kön innehåller 8, 4 och 3. Vi anropar `size()` som ger oss storleken.

Svar: Kön innehåller 8, 4, 3. Om vi hämtar ut ett värde från den kommer vi att få ut 8 eftersom det nu är först i kön.

Grafer (VT24)

Fråga: Utifrån grannlistan (Adjacency List) nedan:

A → B → D

B → A → C → E

C → B → F

D → A

E → B → F

F → C → E

1. I vilken ordning kommer noderna i grafen nedan att besökas om du söker **enligt djupet först (Depth First)** och utgår från noden B (om alla noder lagrar vägen till andra noder i bokstavsordning)?

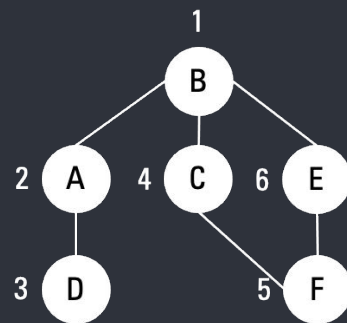
Lösning: Grafer (VT24)

Depth First från noden B

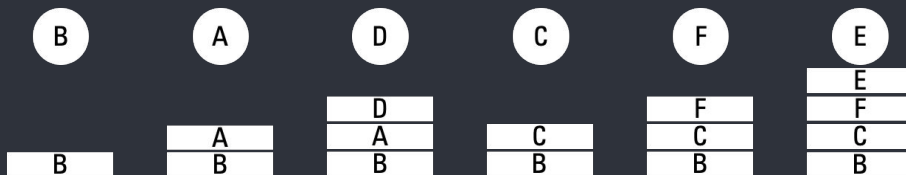
Depth first besöker noderna som en stack fungerar; enligt principen LIFO: **Last in, first out**.

Beteende:

1. Starta på B (lägg B på en stack), lägg den närliggande noden som är först i alfabetisk ordning på stacken (A)
2. Gå till A, lägg den närliggande noden som inte besökts och är först i alfabetisk ordning på stacken (D)
3. Gå till D, poppa D eftersom den inte har några obesökta närliggande noder
4. Gå tillbaka till A, poppa A eftersom den inte har några obesökta närliggande noder
5. Gå tillbaka till B som har fler obesökta närliggande noder
6. Gå till C som är näst på tur i alfabetisk ordning
7. Gå till F som är en obesökt nod näst på tur i alfabetisk ordning
8. Gå till E som är en obesökt nod näst på tur i alfabetisk ordning
9. Eftersom E inte har några obesökta närliggande noder så poppar vi noderna från stacken tills den är tom



Svar: BADCFE



Grafer (VT24)

Fråga: Utifrån grannlistan (Adjacency List) nedan:

A → B → D

B → A → C → E

C → B → F

D → A

E → B → F

F → C → E

2. I vilken ordning kommer noderna i grafen nedan att besökas om du söker **enligt bredden först (Breadth First)** och utgår från noden B (om alla noder lagrar vägen till andra noder i bokstavsordning)?

Lösning: Grafer (VT24)

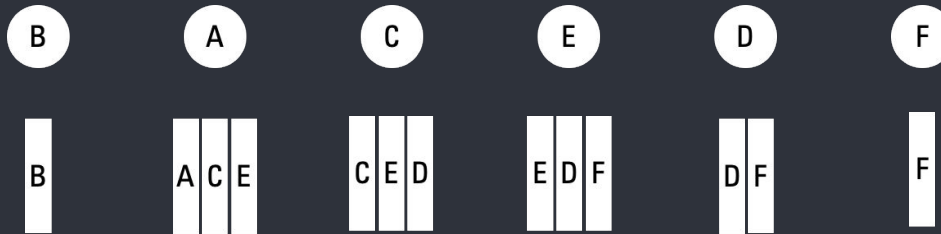
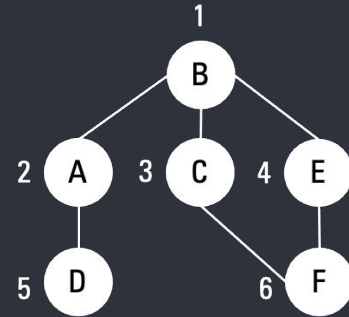
Breadth First från noden B

Breadth first besöker noderna som en kö fungerar; enligt principen FIFO: **First in, first out**.

Beteende:

1. Starta på B (ställ B på en kö), ställ de närliggande noderna i alfabetisk ordning på kö (A, C, E)
2. Gå till A som är näst på tur i alfabetisk ordning, ställ närliggande noder som inte besökts i alfabetisk ordning på kö (D)
3. Gå till C som är näst på tur i alfabetisk ordning, ställ närliggande noder som inte besökts i alfabetisk ordning på kö (F)
4. Gå till E som är näst på tur i alfabetisk ordning, ställ närliggande noder som inte besökts i alfabetisk ordning på kö (F)
5. Gå till D som är näst på tur i kön
6. Gå till F som är näst på tur i kön

Svar: BACEDF



Grafer (VT24)

Fråga: Utifrån grannlistan (Adjacency List) nedan:

A → B → D

B → A → C → E

C → B → F

D → A

E → B → F

F → C → E

3. I vilken ordning kommer noderna i grafen nedan att besökas om du söker enligt djupet först (Depth First) och utgår från noden D (om alla noder lagrar vägen till andra noder i bokstavsordning)?

Lösning: Grafer (VT24)

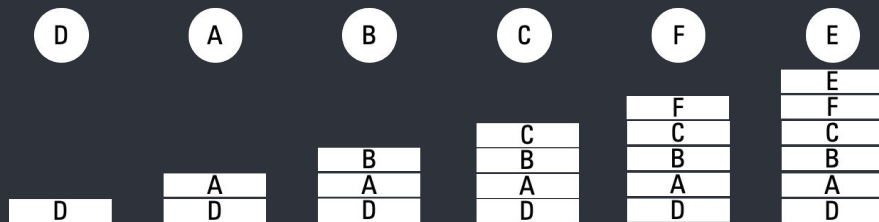
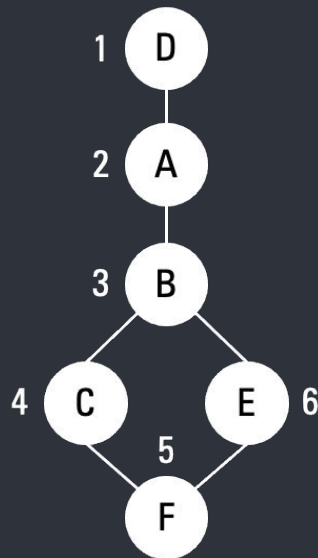
Depth first från noden D

Depth first besöker noderna som en stack fungerar; enligt principen LIFO: **Last in, first out**.

Beteende:

1. Starta på D (lägg D på en stack), lägg den närliggande noden som är först i alfabetisk ordning på stacken (A)
2. Gå till A, lägg den närliggande noden som inte besökts och är först i alfabetisk ordning på stacken (B)
3. Gå till B, lägg den närliggande noden som inte besökts och är först i alfabetisk ordning på stacken (C)
4. Gå till C, lägg den närliggande noden som inte besökts och är först i alfabetisk ordning på stacken (F)
5. Gå till F, lägg den närliggande noden som inte besökts och är först i alfabetisk ordning på stacken (E)
6. Eftersom E inte har några obesökta närliggande noder så poppar vi noderna från stacken tills den är tom

Svar: DABCFE



Förstå kod: tidskomplexitet

Fråga: Vilken tidskomplexitet har koden? (VT22)

```
public static int compute(int n)
{
    if (n < 10)
        return n;
    else {
        int number = 0;
        for (int i = 1; i < 1000; i = i * 2) {
            number += i;
        }
        return number;
    }
}
```

- Oavsett hur stort n är kommer loopen bara att gå max 10 varv
- **Metoden har därför tidskomplexiteten $O(1)$:** antalet operationer som utförs förändras inte i proportion till indatan (utom vid väldigt små värden, men vi bryr oss ju bara om när n är stort)

Förstå kod: rekursivitet

Fråga: Vad blir utskriften av anropet `beast(666)`? (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 var en rad börjar och slutar
- * Förstå implicit casting

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) (egt 166.5, men decimalen ignoreras)
- 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 rekursiva anropen börjar nu återvända (så kallad **stack unwinding**)

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, och körs färdigt för att därefter poppas från stacken.

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?

Fråga: Vilken tidskomplexitet har `beast()`? (VT22)

- **$O(\log n)$** 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 $O(\log n)$
- 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 $O(n \cdot \log n)$: vi skulle då ha anropat `beast()` n antal gånger

Skriva kod (VT23)

Du ska:

1. Implementera metoden så den fungerar enligt JavaDoc-kommentaren.
2. Ange din lösnings tidskomplexitet i en eller flera kommentarer (där n är längden på a).

Förutom att vara korrekt ska din implementation ska vara så effektiv som möjligt (hellre $O(n \cdot \log n)$ än $O(n^2)$ osv). Du väljer om du vill skriva funktionaliteten själv eller om du vill använda datastrukturer och/eller algoritmer från Javas API:er. Koden behöver inte vara exakt korrekt syntaxmässigt (om du t ex inte kommer ihåg vad en metod heter). Det viktiga är att vi kan se att du förstår hur metoden kan implementeras.

```
/**
 * Returns true if there is one or more duplicate elements in an unsorted array.
 * If a contains [ 2, 5, 3, 2 ] the method will return true since it contains
 * duplicate elements with value 2.
 *
 * @param a an unsorted array
 * @return true if there are duplicates in the array, false if not
 */
public boolean isDuplicateElements(int[] a) { }
```

Lösning: Sortera arrayen

- Börja med att fundera på vilket det **lättaste sättet** är **att kolla kopior**: ganska snabbt inser man att det är **om man har en sorterad samling**
- Om **två tal intill varandra** är **samma tal** innehåller samlingen kopior, och vi kan returnera true direkt i loopen, och är då klara. Annars går loopen färdigt och vi returnerar false.
- **Vi tar in en array** i metoden, så vi kan använda `Arrays.sort()` för att sortera den (använd Javas inbyggda API:er - **gör det inte svårare** för er själva än ni behöver!)
- **`Arrays.sort()`** använder QuickSort som **är $O(n \cdot \log n)$** både i bästa fallet och i genomsnitt. Värsta fallet är **$O(n^2)$** , men den är optimerad för att undvika det
- Vi kan nu fylla i metoden:

Lösning: Sortera arrayen

```
import java.util.Arrays;

/**
 * Returns true if there is one or more duplicate elements in an unsorted array.
 * If a contains [ 2, 5, 3, 2 ] the method will return true since it contains
 * duplicate elements with value 2.
 *
 * @param a an unsorted array
 * @return true if there are duplicates in the array, false if not
 */
public boolean isDuplicateElements(int[] a)
{
    Arrays.sort(a);

    for (int i = 0; i < a.length - 1; i++)
    {
        if(a[i] == a[i+1]) return true;
    }

    return false;
}
```

Lösning: Sortera arrayen

```
public boolean isDuplicateElements(int[] a)
{
    Arrays.sort(a);                // O(n*logn)

    for (int i = 0; i < a.length - 1; i++) // O(n)
    {
        if(a[i] == a[i+1]) return true;    // O(1)
    }

    return false;                  // O(1)
}
```

$$O(n \cdot \log n) + \cancel{O(n)} + \cancel{O(1)} + \cancel{O(1)} = O(n \cdot \log n)$$

Alternativ lösning: ett träd

```
import java.util.TreeSet;

/**
 * Returns true if there is one or more duplicate elements in an unsorted array.
 * If a contains [ 2, 5, 3, 2 ] the method will return true since it contains
 * duplicate elements with value 2.
 *
 * @param a an unsorted array
 * @return true if there are duplicates in the array, false if not
 */
public boolean isDuplicateElements(int[] a)
{
    TreeSet<Integer> tree = new TreeSet<>();

    for (int number : a)
    {
        if(!tree.add(number)) return true; //Om vi inte kan lägga till numret
                                           //finns det redan i trädet
    }

    return false;
}
```

Alternativ lösning: ett träd

```
public boolean isDuplicateElements(int[] a)
{
    TreeSet<Integer> tree = new TreeSet<>();           //O(1)

    for (int number : a)                               //O(n)
    {
        if(!tree.add(number)) return true;           //O(logn)
    }

    return false;                                     //O(1)
}
```

$$O(1) + (O(n) * O(\log n)) + O(1) = O(n * \log n)$$

- Träd sorterar alltid direkt vid insättning, så vi vet direkt om värdet redan finns eller ej
- Vi använder TreeSet och inte TreeMap eftersom vi bara har vanliga heltal: maps är för nyckel-värde-par

Kommentera och förstå kod (VT24)

Du ska:

Utifrån koden nedan:

1. Ändra namn på metoden och variabler så de får tydliga och informativa namn. (2 p)
2. Färdigställa JavaDoc-komentaren så den blir tydlig, informativ och korrekt. (2 p)
3. Ange vilken tidskomplexitet metoden har i bästa och värsta fall (där n är längden på l). Du ska även motivera varför metoden har denna tidskomplexitet. (2 p)

```
/**
 *
 * @param
 * @return
 */
public LinkedList<Integer> method(LinkedList<Integer> l) {
    Stack<Integer> s = new Stack<>();
    for (int i : l) {
        s.push(i);
    }
    LinkedList<Integer> a = new LinkedList<>();
    While (!s.empty()) {
        a.add(s.pop());
    }
    return a;
}
```

Kommentera och förstå kod (VT24)

Du ska:

Utifrån koden nedan:

1. Ändra namn på metoden och variabler så de får tydliga och informativa namn. (2 p)
2. Färdigställa JavaDoc-komentaren så den blir tydlig, informativ och korrekt. (2 p)
3. Ange vilken tidskomplexitet metoden har i bästa och värsta fall (där n är längden på l). Du ska även motivera varför metoden har denna tidskomplexitet. (2 p)

```
/**
 *
 * @param originalList
 * @return
 */
public LinkedList<Integer> method reverseList(LinkedList<Integer> l originalList) {
    Stack<Integer> s stack = new Stack<>();
    for (int i item : l originalList) {
        stack.push(item);
    }
    LinkedList<Integer> a reversedList = new LinkedList<>();
    while (!stack.empty()) {
        reversedList.add(stack.pop());
    }
    return reversedList;
}
```


Kommentera och förstå kod (VT24)

Du ska:

Utifrån koden nedan:

1. Ändra namn på metoden och variabler så de får tydliga och informativa namn. (2 p)
2. **Färdigställa JavaDoc-komentaren så den blir tydlig, informativ och korrekt. (2 p)**
3. Ange vilken tidskomplexitet metoden har i bästa och värsta fall (där n är längden på l). Du ska även motivera varför metoden har denna tidskomplexitet. (2 p)

```
/**  
 * Reverse the order of items in a list  
 * @param originalList the original list  
 * @return the list in a reversed order  
 */  
public LinkedList<Integer> reverseList(LinkedList<Integer> originalList) {  
    Stack<Integer> stack = new Stack<>();  
    for (int item : originalList) {  
        stack.push(item);  
    }  
    LinkedList<Integer> reversedList = new LinkedList<>();  
    while (!stack.empty()) {  
        reversedList.add(stack.pop());  
    }  
    return reversedList;  
}
```

Kommentera och förstå kod (VT24)

Du ska:

Utifrån koden nedan:

1. Ändra namn på metoden och variabler så de får tydliga och informativa namn. (2 p)
2. Färdigställa JavaDoc-komentaren så den blir tydlig, informativ och korrekt. (2 p)
3. **Ange vilken tidskomplexitet metoden har i bästa och värsta fall (där n är längden på l). Du ska även motivera varför metoden har denna tidskomplexitet. (2 p)**

```
/**
 * Reverse the order of items in a list
 * @param originalList the original list
 * @return the list in a reversed order
 */
public LinkedList<Integer> reverseList(LinkedList<Integer> originalList) {
    Stack<Integer> stack = new Stack<>(); //O(1)
    for (int item : originalList) { //O(n)
        stack.push(item); //O(1) item läggs alltid högst upp
    }
    LinkedList<Integer> reversedList = new LinkedList<>(); //O(1)
    while (!stack.empty()) { //O(n)*O(1) = O(n) kolla om stacken är tom för varje n
        reversedList.add(stack.pop()); //O(1) + O(1) = O(1) poppa det översta elementet från
        stacken O(1) och lägg till i det på slutet av den länkade listan O(1) eftersom en länkad
        lista i Java som standard är dubbellänkad
    }
    return reversedList; // O(1)
} // sammanlagt O(1) + O(n*1) + O(1) + O(n*1) + O(1) = O(n)
```

Välja sorteringsalgoritm 2 (VT24)

Du ska:

utveckla en applikation som behöver sortera listor med stora mängder data. Listorna är i ca 50 % av fallen redan sorterade men för att vara säker behöver vi varje gång antingen sortera om listorna eller kontrollera om de är osorterade och i så fall sortera dem. I de fall listorna är osorterade är det maximalt 15 % av elementen som inte är i ordning. Utifrån scenariot:

1. Väljer du att varje gång sortera listorna eller istället att kontrollera om de är osorterade och i så fall sortera dem? (5 p)
2. Vilken sorteringsalgoritm väljer du? (5 p)

För båda frågorna behöver du motivera ditt svar genom att resonera om dina val jämfört med andra möjliga val. I ditt resonemang vill vi att du hänvisar till tidskomplexitet. Om du gör antaganden vill vi att du tydligt beskriver dessa.

Välja sorteringsalgoritm 2 (VT24)

Du ska:

utveckla en applikation som behöver sortera listor med stora mängder data. Listorna är i ca 50 % av fallen redan sorterade men för att vara säker behöver vi varje gång antingen sortera om listorna eller kontrollera om de är osorterade och i så fall sortera dem. I de fall listorna är osorterade är det maximalt 15 % av elementen som inte är i ordning. Utifrån scenariot:

1. Väljer du att varje gång sortera listorna eller istället att kontrollera om de är osorterade och i så fall sortera dem? (5 p)

Kolla om varje lista är sorterad en gång först för att sedan **spara in på 50% av sorteringarna.**

2. Vilken sorteringsalgoritm väljer du? (5 p)

Välja sorteringsalgoritm 2 (VT24)

Du ska:

utveckla en applikation som behöver sortera listor med stora mängder data. Listorna är i ca 50 % av fallen redan sorterade men för att vara säker behöver vi varje gång antingen sortera om listorna eller kontrollera om de är osorterade och i så fall sortera dem. I de fall listorna är osorterade är det maximalt 15 % av elementen som inte är i ordning. Utifrån scenariot:

1. Väljer du att varje gång sortera listorna eller istället att kontrollera om de är osorterade och i så fall sortera dem? (5 p)

Kolla om varje lista är sorterad en gång först för att sedan spara in på 50% av sorteringarna.

2. Vilken sorteringsalgoritm väljer du? (5 p)

Insertionsort har vinst när **stor mängd data är sorterad** så den väljs. Mergesort skulle i och för sig passa bra när det är mycket osorterade data men presterar sämre vad gäller tid ($O(n \log n)$) och är inte in-place. Insertion Sort har $O(n)$ i värsta fall när elementen redan är sorterade eftersom den bara flyttar element när elementet som undersökts är felplacerat. I detta fall kommer det i värsta fall att hända för 15 procent av elementen vilket medför att Insertion Sort kommer att prestera nära $O(n)$ eller som mest $O(n + k)$ där $k \approx 0,15n$. Att den är in-place, betyder att den inte behöver någon ytterligare datastruktur under sorteringen vilket sparar minne.