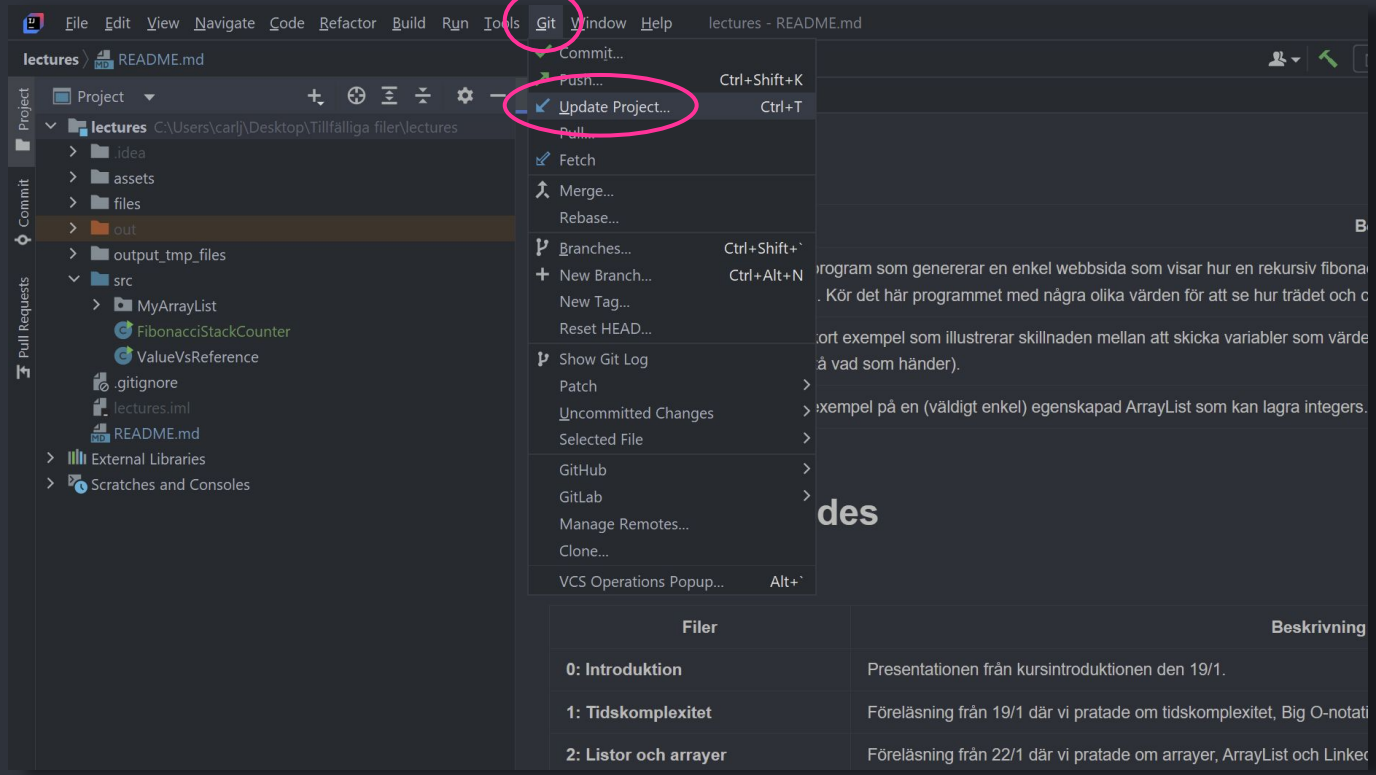
Algoritmer och Datastrukturer

Föreläsning 5

HashMap, TreeMap
och sortering

https://github.com/carljohanj/algo



Innan vi börjar ...

- Kurser på Uppsala Universitet är skyldiga att göra en deltidsav-stämning för att se hur kursen och undervisningen fungerat hittills
- Nu har inte halva kursen gått ännu, men det är bättre att göra dessa lite tidigare så att vi hinner göra förändringar om det skulle behövas
- Bra tillfälle att få feedback om det som fungerar, men även om det finns saker som behöver förbättras
- Ägna 5-10 minuter åt att fylla i den här är ni snälla
- Länk skickas ut som kursanslag

Lite snabb repetition på rekursion ...

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

recursion (n.)

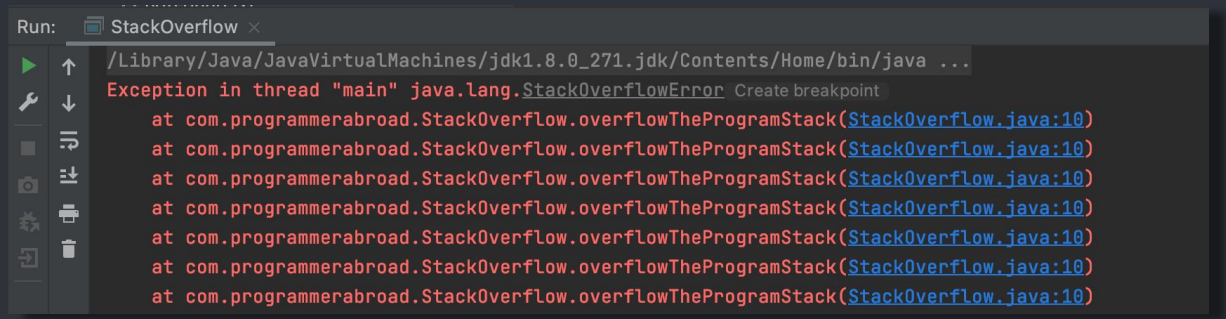
"return, backward movement," 1610s, from Latin *recursionem* (nominative *recursio*) "a running backward, return," noun of action from past-participle stem of *recurrere* "run back" (see **recur**).

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 backtrackingbeteende**

- **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 nextNode;

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

- Innehåller en instans av sig själv
- Går att bygga både köer 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 ($O(2^n)$)
- Går dock att optimisera med dynamisk programmering

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 datastrukturer “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)

Några repetitionsuppgifter!

Repetition: Abstrakt datatyp

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?

Svar: En stack (värdena kommer ut i omvänd ordning)

Repetition: Tidskomplexitet

– Vad blir tidskomplexiteten?

```
public LinkedList<Integer> reverseList(LinkedList<Integer> originalList)
{
      Stack<Integer> stack = new Stack<>(originalList.size());       // O(n) +

      for (int item : originalList)                                  // O(n) *
      {
            stack.push(item);                                        // O(1) +
      }

      LinkedList<Integer> reversedList = new LinkedList<>();         // O(1) +

      while (!stack.empty())                                         // O(n) *
      {
            reversedList.add(stack.pop());                           // O(1) +
      }

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

Repetition: Tidskomplexitet

- Räkna samman faktorerna. Vi får:

  $O(n) + O(n) * O(1) + O(1) + O(n) * O(1) + O(1)$

- Vilket stryker de minsta koefficienterna:

  $O(n) + O(n) * \cancel{O(1)} + \cancel{O(1)} + O(n) * \cancel{O(1)} + \cancel{O(1)}$

- Vilket blir:

  $O(n) + O(n) + O(n) = 3\ O(n)$

- Vilket är samma som:

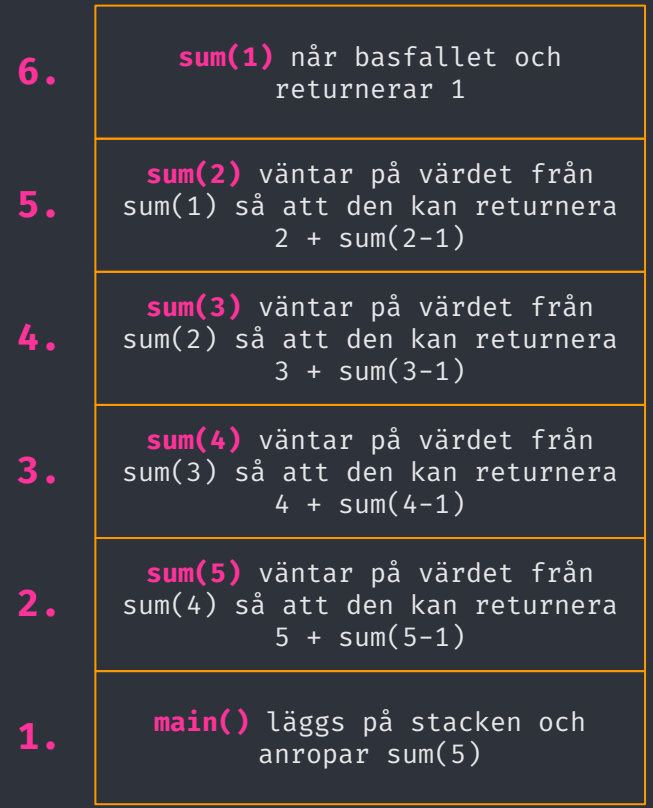
  $\cancel{3}\ O(n)$

Svar: O(n)

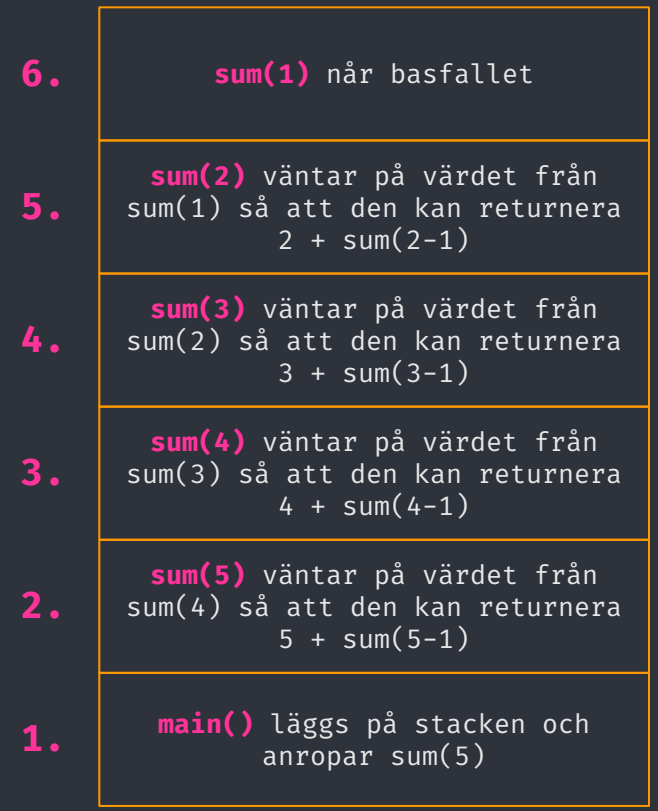
Repetition: Rekursion

– Vad returnerar metoden om den anropas med sum(5)?

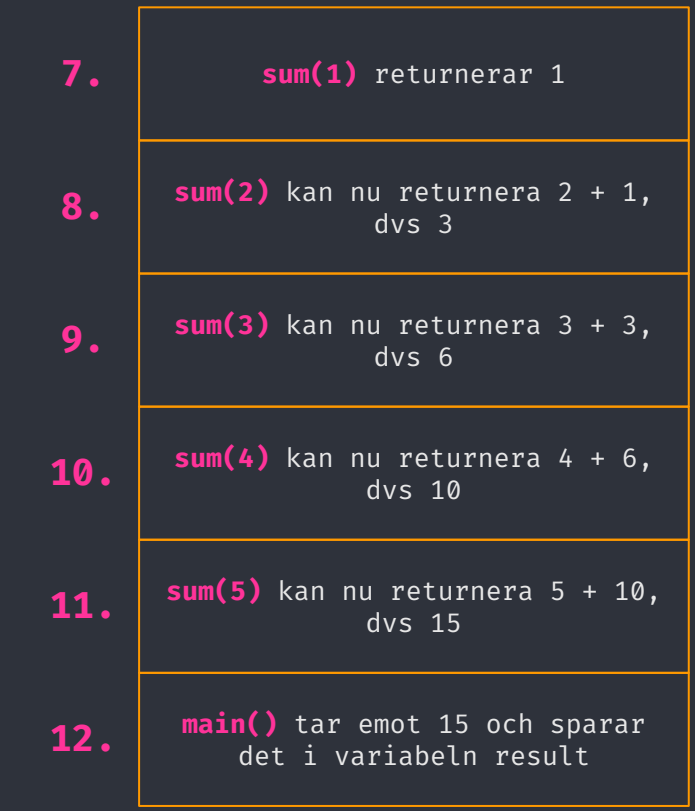
```
public static int sum(int n)
{
   if (n == 1) //Basfall
   {
     return 1;
   }

   return n + sum(n - 1);
}


//Metoden anropas så här:

public static void main(String[] args)
{
   int result = sum(5);
}
```



Callstack



Callstack: vad som läggs på stacken



Callstack: hur anropen returnerar

Repetition: Rekursion

- Stacken lindas upp (så kallad stack unwinding) i takt med att metodanropen returnerar
- En efter en kommer alla frames som vi lagt på stacken att poppas och returnera värden till föregående metod som nu kan köras färdigt
- sum(5) kommer att returnera 15
- Det finns sedan inte mer kod i main(): metoden kommer att köras färdigt och poppas från callstacken, och programmet avslutas

Svar: sum(5) returnerar 15

Maps: nyckel-värde-datatyper

- När vi pratar om maps på engelska menar vi inte kartor på svenska, utan **datastrukturer som mappar en nyckel till ett värde**

- Kommer från matematiken, där vi stoppar in ett värde i en funktion och får ut ett annat:

  `f(x) = y` **“F av x är lika med y”**

- När vi skapat en sådan funktion säger vi att vi har mappat x till y. Precis likadant fungerar en Map: när vi vill leta upp något stoppar vi in en nyckel och får ut ett värde som är associerat med den. Den är i praktiken en tabell.

- Vi kan använda sökord (nycklar) för att hitta information (värden) i den här tabellen.

Exempel: namn associerade med telefonnummer.

Key	Value
Andreas	073123456
Lovisa	070123456
Carl-Johan	079123456

Map-interfacet i Java

```
package java.util;

import java.util.function.BiConsumer;
import java.util.function.BiFunction;
import java.util.function.Function;
import java.io.Serializable;
```

> An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

- Det finns många klasser som implementerar det här interfacet i Java
- På den här kursen intresserar vi oss för två: **HashMap** och **TreeMap**

Map: vanliga operationer

```
Map<Integer, String> map = new …;    //Skapar en map med en int som nyckel och string som värde
map.put(1, "Andreas");    //Lägger till en nyckel och ett värde
map.put(2, "Lovisa");
map.putIfAbsent(2, "Carl");    //Lägger bara till om nyckeln inte redan finns

boolean isTrue = map.containsKey(1);     //Verifierar att nyckeln 1 finns (sant)
boolean isFalse = map.containsValue("Carl");   //Kommer inte att hitta värdet “Carl”

Set<Integer> keys = map.keySet();   //Plockar ut ett set med alla nycklar
Collection<String> values = map.values();     //Plockar ut en lista med alla värden
```

En nyckel kopplad till flera värden

- Vi kan använda objekt inklusive samlingar som nycklar och värden.
  Men hur gör vi när en nyckel kan vara kopplad till flera värden?

- Föreställ er att vi äger Spotify och vill skapa en map för
  artister där namnet på artisten/bandet är nyckel och värdet är
  alla album de släppt. Vid varje input vet vi inte om det redan
  Finns album förknippade med artisten eller ej:

```
Map<String, List<String>> artistAlbums = new...;     //TreeMap eller HashMap; välj en

while (hasMoreData())
{
  String artist = getNextArtist();
  String album = getNextAlbum();

  //Skapar bara en tom lista om artisten inte redan finns:
  artistAlbums.putIfAbsent(artist, new ArrayList<>());

  //Lägg till albumet i listan som är förknippad med nyckeln (dvs artisten)
  artistAlbums.get(artist).add(album);
}
```

Att loopa genom en Map

- För att iterera genom en Map behöver vi använda en enhanced for-loop

- En map sparar ett Entry-set internt (tänk på Entry som ett objekt som lagrar nyckel och värde) och det är detta set vi itererar genom:

```
for (Map.Entry<Integer, String> e : map.entrySet())
{
   System.out.println(e.getKey() + " " + e.getValue());
}
```

- En vanlig for-loop kan inte göra detta

```
kaffepaus(15);
```

TreeMap: en trädliknande struktur

- En TreeMap är en av de mest kraftfulla datastrukturerna i Java eftersom den erbjuder oss **både en map med nyckel-värde-par och automatisk sortering vid insättning**
- Den implementerar `Map<K, V>`-interfacet i Java
- Den kallas för TreeMap eftersom den implementerar ett binärt sök-träd som underliggande datastruktur (vi ska prata om vad det här är för något nästa vecka och bygga några enkla exempel!)
- Mer exakt implementerar den något som kallas för ett Red-and-Black Tree, vilket inte har något med fysisk färg att göra, utan bara är ett sätt att säga att trädet är självbalanserande: det ser till internt att det inte blir ojämnt
- Tillåter inte att nycklar är null (men värden är ok)
- Har **O(logn) för insättning och uthämtning**

Vanliga operationer för en TreeMap

```
TreeMap<Integer, String> map = new TreeMap<>();
map.put(1, "Carl");
…
map.put(100, "Buster");

int firstKey = map.firstKey();   //O(1) i uthämtning
int lastKey = map.lastKey();     //O(1) i uthämtning

Set<Integer> reversedKeys = map.descendingKeySet();    //O(1) i uthämtning

Map<Integer, String> reversed = map.descendingMap();   //O(1) i uthämtning

Map<Integer, String> from1to50 = map.headMap(50, true); //Hämtar första 50 elementen
Map<Integer, String> from50to100 = map.tailMap(50, true); //Hämtar sista 50 elementen
Map<Integer, String> from20to50 = map.subMap(20, true, 50, true); //Hämtar ett godtyckligt span

Set<Map.Entry<Integer, String>> entrySet = reversed.entrySet(); //Hämtar ett Entry-set

for (Map.Entry<Integer, String> e : entrySet)    //Loopar genom Entry-setet på O(n) tid
{
   System.out.println(e.getKey() + " " + e.getValue());
}
```

SubMap: att hämta ut en vy

- Submap är en av de mest användbara funktionerna i en TreeMap, och det är också en funktionalitet som t.ex. HashMap saknar

- En **subMap är egentligen bara en vy** som gör tre saker:

  * Den ger oss en referens till trädet i vår TreeMap
  * Den ger oss en referens till noden där vi ska börja iterera
  * Den ger oss en referens till noden där vi ska sluta

- Det här innebär att **det går på konstant tid - O(1) - att hämta ut en submap** eftersom inget arbete görs av själva metoden!

- När vi väl mottagit en submap måste vi göra något med den, t.ex. iterera genom den för att hämta ut värdena, och **det här arbetet kommer få tidskomplexiteten O(k)**

Nu hittar du på tidskomplexiteter igen! O(k)?? Du har sagt att det heter n!

- När man pratar om datamängder kallar man **den totala datamängden för n** (det här är bara en konvention som kommer från matematiken): vi brukar tänka på det som “nummer” eller “number of elements”

- När vi pratar om en **delmängd av den totala datamängden brukar vi referera till den som k:** det här är återigen bara en konvention och har ingen särskild mening

- Man använde redan i och j till indexplatser i loopar och fortsatte på samma mönster bara

- När vi säger att något tar O(k) tid att iterera menar vi att vi **bara itererar över en viss delmängd k av den totala datan** (som är n)

[Kodexempel]

Koden finns i SubMapExample.java

HashMap: en hashad struktur

- **En HashMap är ytterligare 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

```
Map<k, V> hashMap = new HashMap<>();
```

- När vi försöker lägga in någonting hashar den först nyckeln: det här innebär att den **översätter nyckeln till en integer**
- 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

Hashning

Exempel för en HashMap med int som nyckel och sträng som värde:

```
Map<Integer, String> hashMap = new HashMap<>();
hashmap.put(26, “Alice”);
```

Internt översätter HashMapen nyckeln till ett binärt 32-bitarsnummer:

26_{10} = 00000000 00000000 00000000 00011010_{2}

HashMapen beräknar sedan en indexplats i den underliggande arrayen som **utför en bitwise AND mot storleken på själva arrayen**. En HashMap skapas med storleken 16 som default (den kan förstora sig själv precis som en lista).

Formeln: indexplats = hash & (capacity - 1)

```
hash           = 26 =    11010
capacity-1     = 15 =    01111
-------------------------------
index     =          01010
```

10 i bas-10!

Alltså ska “Alice” stoppas in på plats 10 i arrayen inuti HashMapen!

Vad händer om två insättningar har samma hashkod?

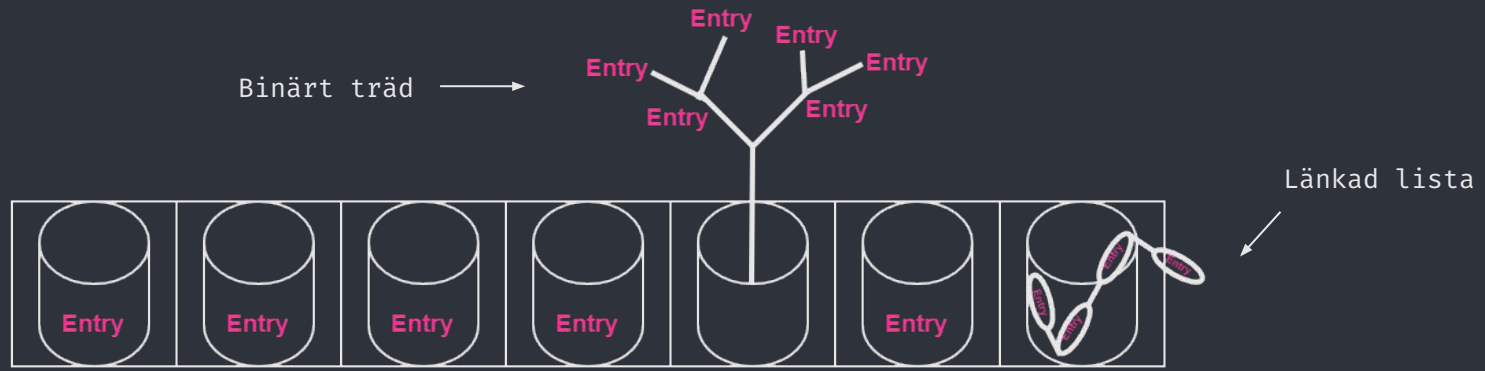
- **Hashkoder är inte unika:** två olika koder kan ha (och producerar ofta) samma index dels eftersom de komprimeras, dels eftersom arrayen bara har ett visst antal indexplatser
- En indexplats i en HashMap kallas för en **"bucket"**: namnet är menat att indikera att den kan innehålla **mer än ett** nyckel-värde-par
- Om en bucket 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(logn)** (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
O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)

HashMap



Index 0	Index 1	Index 2	Index 3	Index 4	Index 5	Index 6
O(1)	O(1)	O(1)	O(1)	O(logn)	O(1)	O(n)

Load Factor

- **Load Factor** kallas det mått som bestämmer om Hashmapen behöver förstora den underliggande arrayen: om det här sker kommer den dels att skapa en dubbelt så stor array, dels att hasha om alla element och stoppa in i den nya arrayen: båda operationerna har O(n) i tid

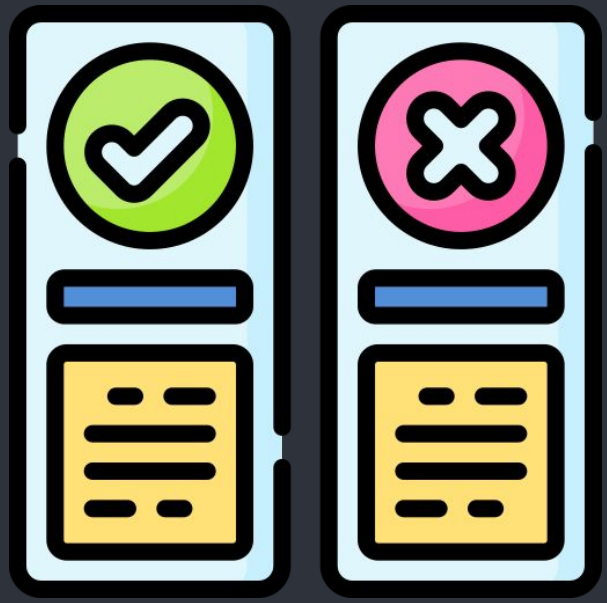
$$\text{Load Factor} = \frac{\text{Antal lagrade element}}{\text{Storlek på den interna arrayen}}$$

- Om Load Factor är **större än 0.75** kommer HashMapen att förstora sig

- Att förstora och omhasha är dyra operationer, och de har dessutom dålig platskomplexitet (de är minnesineffektiva)

För- och nackdelar med HashMap: summering

- Hashning 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**
- Samma antal operationer utförs alltid för att beräkna numret och därför blir operationen konstant, O(1)
- 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 tidskomplexitets-mässigt (**O(n) för att kopiera över all data**) och platskomplexitetsmässigt (vi får en dubbelt så stor array)
- Den **sorterar inte data automatiskt och bibe-håller heller inte insättningsordningen:** hash-funktionen 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(logn)**
- **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
- HashMap kan inte heller bevara insättningsordning (men en LinkedHashMap kan!)
- Ofta är det **sorteringsbehovet** man väljer efter: behöver man värden i en viss ordning vill man helst att datatypen **fixar det snabbt vid insättning**

Tidskomplexitetsjämförelse

– Tidskomplexitet för bästa / sämsta fallet för ett antal vanliga operationer:

Operation	HashMap	TreeMap
Lägga till	O(1) / O(logn)	O(logn)
Hämta ut	O(1) / O(logn)	O(logn)
Hämta intervall av element	O(k) / O(k * logn)	O(1)
Hämta samtliga nycklar	O(1)	O(1)
Hämta samtliga värden	O(1)	O(1)
Iterera genom	O(n)	O(n)

```
Lunch.sleep(60);
```

Vi samlas igen i D24 kl 13.15

Sortering

- Sorteringar har **två egenskaper:** de kan vara stabila och in-place

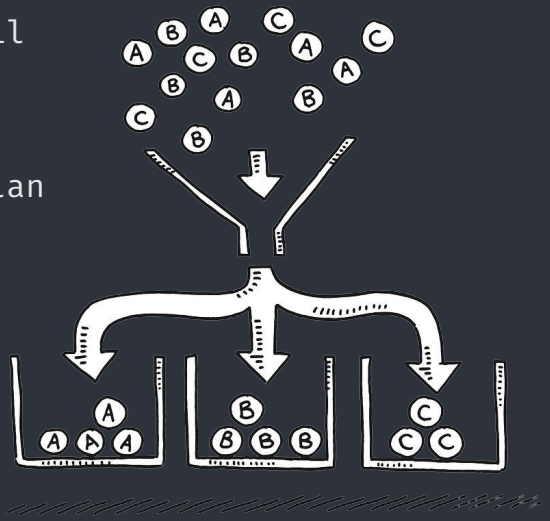
- Med en **stabil sorteringsalgoritm** menar vi något som bibehåller den relativa ordningen mellan objekten

  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: ofta önskvärt beteende när datan är stor



Enklare jämförelsebaserad sortering

	Best Case	Worst Case	Average Case
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

— Vi brukar dela upp sortering i två kategorier:

* Jämförelsebaserad sortering
  * Enklare sortering $O(n^2)$
  * Divide-and-conquer $O(n \log n)$
* Sortering utan jämförelser $O(n)$

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

Enklare jämförelsebaserad sortering

	Stable	In-place
Insertion Sort	Ja	Ja
Bubble Sort	Ja	Ja
Selection Sort	Nej	Ja

Stable: Element med samma värde behåller sin inbördes ordning.

In-place: Kan sorteringen göras utan extra datastrukturer?

Fördelar:

* Enkla att implementera
* Fungerar bra för små datamängder

Bubble Sort

```
public void bubbleSort(int[] a)
{
   for (int i = 0; i < a.length; i ++)
   {
     for(int j = 0; j < a.length - 1 - i; j ++)
     {
        if(a[j] > a[j + 1])
        {
          int temp = a[j];
          a[j] = a[j + 1];
          a[j + 1] = temp;
        }
     }
   }
}
```

Gör:

- Jämför två element och byter plats på dem om de är i fel in-Bördes ordning

- Det största elementet “bubblar upp till ytan” för varje varv

Används när:

- Vi har väldigt små värden, och i undervisningssyfte

Best case: O(n) (redan sorterad)
Worst case: $O(n^2)$
Avg. case: $O(n^2)$

Selection Sort

```
public void selectionSort(int[] a)
{
   for (int i = 0; i < a.length; i ++)
   {
     int min = i;

     for (int j = i + 1; j < a.length; j ++)
     {
        if (a[j] < a[min])
        {
          min = j;
        }
     }

     int temp = a[i];
     a[i] = a[min];
     a[min] = temp;
   }
}
```

Gör:

- Algoritmen hittar det minsta kvarvarande elementet om och om igen och stoppar det på sin rätta plats
- Utför antalet swaps på O(n) tid: dess stora fördel
- Är ej stabil eftersom swap kan medföra att element med lika värde kan byta inbördes ordning.

Används när:

- När antalet minnesskrivningar behöver begränsas (swaps är dyra)

Best case: $O(n^2)$
Worst case: $O(n^2)$
Avg. case: $O(n^2)$

Insertion Sort

```
public void insertionSort(int[] a)
{
   for (int i = 1; i < a.length; i ++)
   {
     int j = i - 1;
     int current = a[i];

     while (j >= 0 && a[j] > current)
     {
        a[j + 1] = a[j];
        j = j - 1;
     }

     a[j + 1] = current;
   }
}
```

Gör:

- Kollar ena halvan av en array åt Gången och sorterar den genom att stoppa varje element på rätt plats

Används när:

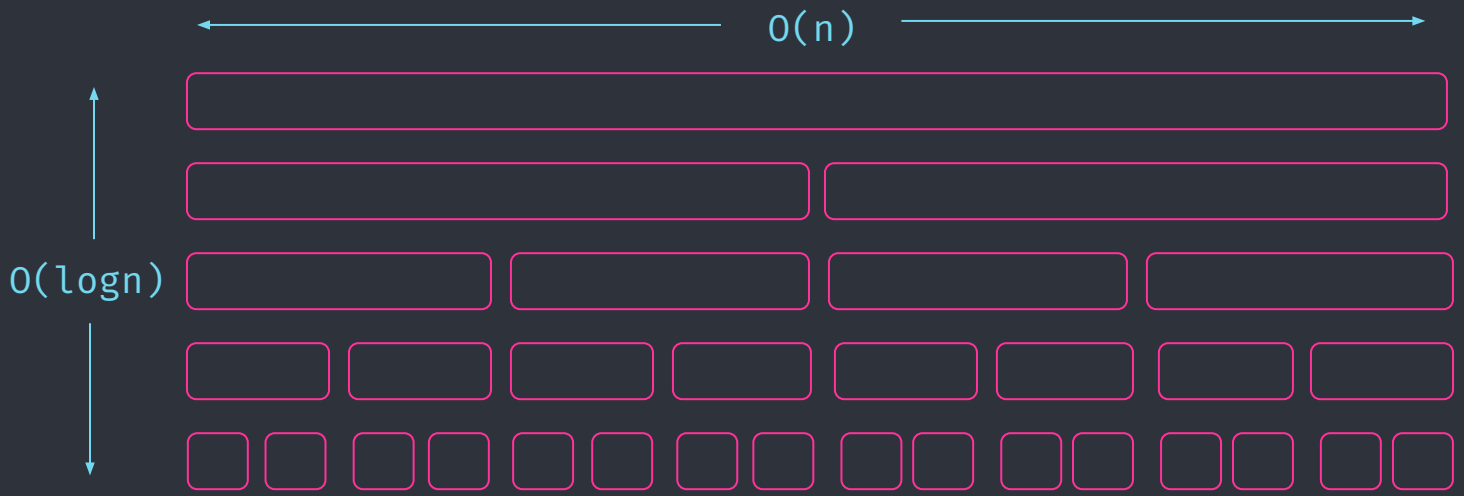
- Ofta när man har samlingar som redan är delvis sorterade: den är väldigt effektiv om vi vet att inte all data behöver flyttas om

- Används internt av TimSort

Best case: O(n) (redan sorterad)
Worst case: $O(n^2)$
Avg. case: $O(n^2)$

Divide-and-Conquer

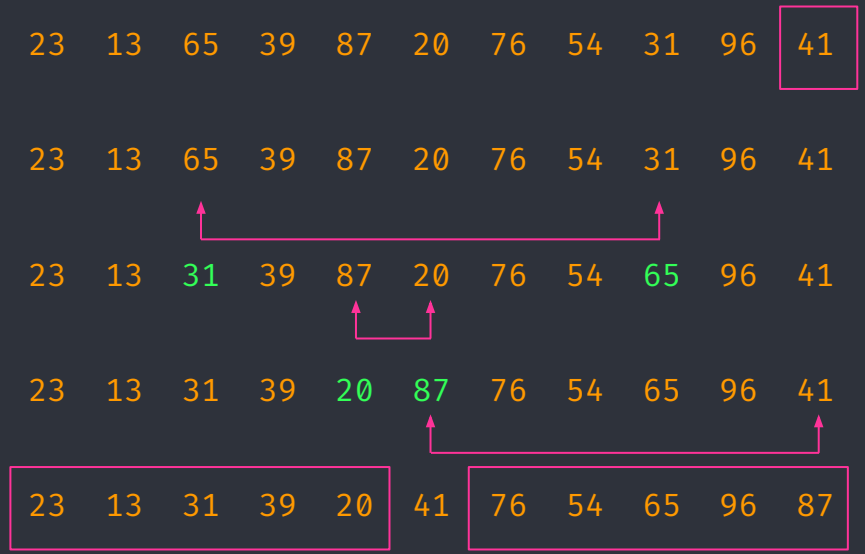
	Best Case	Worst Case	Average Case
Merge Sort	O(n logn)	O(n logn)	O(n logn)
Quick Sort	O(n logn)	$O(n^2)$	O(n logn)



O(n), O(logn) antal gånger = O(n logn)

```
kaffepaus(15);
```

QuickSort: pivotsortering



- Välj pivot (längst till höger). Pivot betyder ungefär “referenspunkt som allting annat jämförs med”
- Element från vänster som är större än pivot swappas med element från höger som är mindre än pivot
- När pilarna mötes, swappa pivot
- Tidigare pivot (41) är nu på rätt plats. Stegen ovan upprepas för de nya partitionerna

QuickSort

Best Case	Worst Case	Average Case
O(n logn)	$O(n^2)$	O(n logn)

- **Värsta fallet** för Quick Sort (ej optimerad) **inträffar när elementen redan är sorterade:** pivot-värdet kommer då hela tiden vara högsta värdet
- Detta innebär att **samtliga element hamnar i samma partition i varje steg.** Vi behöver då utföra n antal operationer n antal gånger i stället för log n antalet gånger och får då tidskomplexiteten $O(n^2)$

MergeSort

```
public void mergeSort(int[] arr)
{
    if (arr.length <= 1) { return; } //redan sorterad

    int mid = arr.length / 2;
    int[] left = new int[mid];
    int[] right = new int[arr.length - mid];

    System.arraycopy(arr, 0, left, 0, mid);
    System.arraycopy(arr, mid, right, 0, arr.length - mid);

    mergeSort(left);
    mergeSort(right);
    merge(arr, left, right);
}
```

Gör:

- Rekursiv algoritm som delar upp en array i halvor, sorterar varje halva och sedan slår samman dem (“merge”) igen

Används när:

- Dataseten är väldigt stora och in-place inte är ett krav

Best case: O(n logn)
Worst case: O(n logn)
Avg. case: O(n logn)

Jämförelse, olika sorteringsalgoritmer

Algoritm	Tidskomplexitet (Bäst/Sämst)	In-Place	Stabil	Används
QuickSort	O(n*logn) / O(n^2)	Ja	Nej	Stora dataset som behöver in-place-sortering
MergeSort	O(n*logn)	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*logn) i genomsnitt, och även O(n*logn) 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, eller
  2. Konvertera den **till en TreeMap,** som då sorterar automatiskt efter nycklar