Algoritmer och Datastrukturer

Föreläsning 2

Arrayer, ArrayList och länkad lista

Mål för dagen

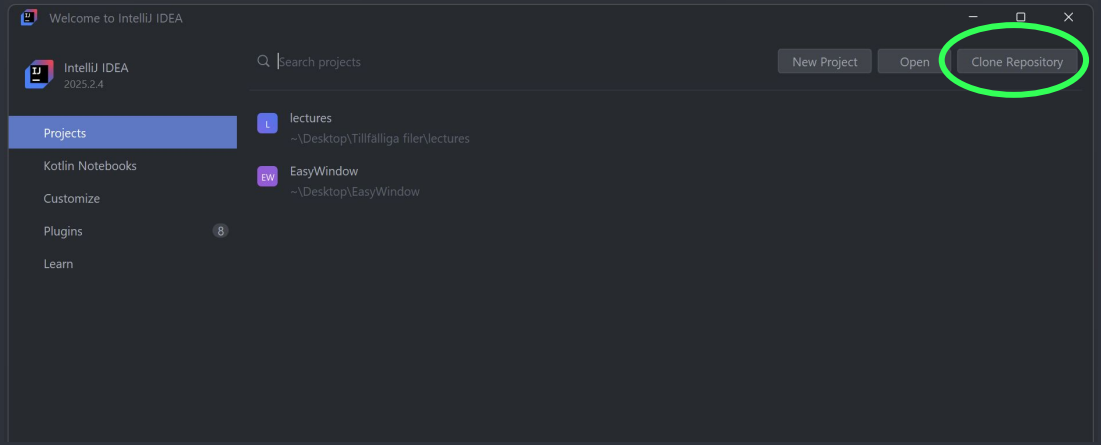
- Ha ett hum om hur logaritmisk tillväxt fungerar
- Undersöka arrayer och hur de lagras i minnet
- Förstå hur en ArrayList fungerar, och varför
- Veta vad en länkad lista är för någonting och hur den skiljer sig från en ArrayList
- Känna till några av de inbyggda algoritmerna i Collections-paketet i Java

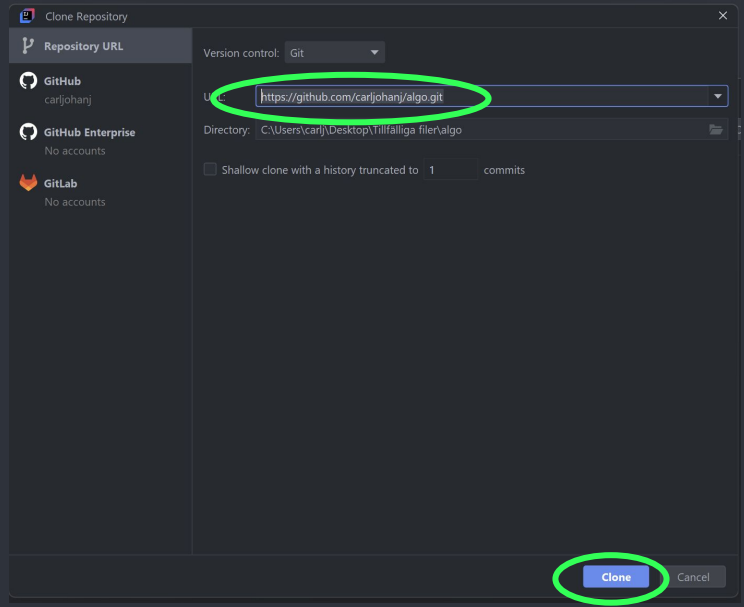
Innan vi börjar: Programråd

- Behöver systemvetare till programrådet
- Kontakta Thomas Ejnefjäll om ni läser programmet och är intresserade: thomas.ejnefjall@im.uu.se
- Bra merit: man får ett diplom
- De matar dig ibland på mötena!
- Två träffar per termin cirka: man har möjlighet att påverka programmet och bidra till att förbättra sin utbildning

Plocka hem kursrepositoryt!

https://github.com/carljohanj/algo.git





1. Starta IntelliJ (och stäng eventuella projekt som är öppna)
2. Klicka på “Clone repository”
3. Stoppa in länken och tryck på Clone
4. Presto!

Om repositoryt

- Jag kommer att publicera **allt föreläsningsmaterial** här så att ni kan komma åt det direkt i IntelliJ
- **Föreläsningar och kodexempel** kommer också att läggas upp på kurssidan men repot innehåller **även länkar, videoklipp och annat smått och gott** som kommer hjälpa er mycket när ni pluggar
- Bra sätt att organisera allt material
- Kommer att uppdateras före och efter varje föreläsning: bra tillfälle att bekanta sig med Git och IntelliJ-plugins
- **Lägg inte till egna filer i det här repot eftersom det är levande:** när ni trycker på update kommer det att skriva över era ändringar i så fall
- Ni får göra vad ni vill med det i övrigt: efter kursen är det bara att koppla bort remoten och behålla det för evigt

Snabb repetition: tidskomplexitet

- **Tidskomplexitet är hur vi bedömer algoritmisk effektivitet:** vi använder det här måttet eftersom hur snabbt en viss bit kod körs alltid kommer skilja sig kraftigt, inte bara mellan system utan även mellan olika tidpunkter på samma dator

- Det vi vill veta är: när vi stoppar in större och större datamängder, vad händer med tidsutvecklingen? **Hur ser tillväxthastigheten ut för koden?**

- Tidskomplexitet definieras i “Big O”-notation, även kallad ordo

- Vanliga tidskomplexiteter:

  * O(1) - konstant, utför samma mängd arbete oavsett hur stor datan är

  * O(n) - linjär tidskomplexitet, när datamängden dubblas så dubblas ungefär tiden som koden tar att köra

  * O(n^2) - kvadratisk tidskomplexitet (även kallad polynom), när datamängden dubblas så fyrdubblas körtiden ungefär

Snabb repetition: vad blir tidskomplexiteten?

```
public int[] copy(int[] n)
{
   int[] copy = new int[n];          // O(n) +

   for (int i = 0; i < n; i++)       // O(n) *
   {
      copy[i] == n[i];               // O(1) +
   }

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

$O(n) + O(n) * \cancel{O(1)} + \cancel{O(1)} = O(n) + O(n) = \cancel{2}\ O(n)$

– Tidskomplexiteten blir O(n) eftersom vi alltid loopar genom alla n: när mängden n dubblas så dubblas även tiden

Vad blir tidskomplexiteten?

```java
public int accumulate(int[] n)
{
    int sum = 0;

    for (int i = 0; i < 100; i++)
    {
        sum += n[i];
    }

    return sum;
}
```

Beteende:

* Loopen körs alltid 100 ggr
* Antalet iterationer bestäms inte av storleken på input
* Metoden utför en konstant mängd jobb

– **Tidskomplexiteten blir O(1)** eftersom loopen aldrig går mer än 100 varv oavsett hur stor indatan är

Vad blir tidskomplexiteten?

```java
public int accumulate(int[] n)
{
   int sum = 0;

   for (int i = 0; i < 1_000_000; i++)
   {
      sum += n[i];
   }

   return sum;
}
```

Beteende:

* Loopen körs alltid en miljon ggr
* Antalet iterationer bestäms inte av storleken på input
* Metoden utför en konstant mängd jobb

- **Tidskomplexiteten blir O(1) även här** eftersom loopen aldrig går mer än en miljon varv oavsett hur stor indatan är
- En miljon varv kan verka väldigt mycket, men ur det större perspektivet, när datamängden (n) blir väldigt stor, kommer de ändå att se små och obetydliga ut

Vi kikade på det här O(1)-exemplet förra gången:

– O(1), eftersom tiden inte påverkas av storleken (n). Vi ser ingen ökning av tiden när vi ökar storleken på inputen.

n	tid
2	4
4	6
8	5
16	7
32	3
205	4

Det här är också O(1):

– Exekveringstiden är mycket längre, men vi ser fortfarande ingen ökning av tiden när vi ökar storleken på inputen, och grafen blir densamma:

n	tid
2	356 567
4	356 549
8	356 498
16	356 563
32	356 529
205	356 543

Insikten: konstanter kan vara enorma

- Att en operation är konstant (**O(1)**) innebär inte automatiskt att den är snabb
- Det innebär bara att den inte växer med storleken på input
- Konstanter kan ibland dominera prestanda och sätta en gräns för hur pass mycket någonting går att effektivisera
- En beräkning kan ta ett år att utföra, men om den tar ett år oavsett om storleken på problemet är 1 eller 100 eller 10^{15} är den konstant
- Det här är viktigt att inse:
  * Big-O-notation försöker inte besvara frågan **“Vilken algoritm är snabbast för en viss input?”**
  * Den besvarar frågan **“Hur utvecklas tiden när inputen ökar?”**
- Konstanter spelar ingen roll i slutändan, men tillväxt gör. En stor konstant kan dominera för tillfället, men **tillväxthastigheten kommer så småningom att dominera konstanten vid tillräckligt stora n-värden**

Lärdom: videorenderare

- Avancerad kurs i programmering: Uppgiften var att skapa en applikation som använde GPU:n för att generera en vinst när saker ritas upp på skärmen
- Trots avancerad kod gick det inte att optimera mer än ca 40% i bästa fall gentemot CPU-rendering eftersom konstanterna helt enkelt blev för stora: det kostade för mycket att skapa temporära buffertar, ladda upp all data till grafikkortet, osv
- Höll på i månader för att göra den så effektiv som möjligt, men den dyrköpta lärdomen var att konstanterna helt enkelt var för stora
- Såna här problem är ganska vanliga inom programvara: det finns en gräns för hur effektiv kod kan bli

Vänster: indata. Höger: utdata.

```
2 references
private void RenderFramesOnGPU(string inputDirectory, string outputDirectory)
{
    object lockObject = new object();

    ReadOnlyCollection<ComputePlatform> platforms = ComputePlatform.Platforms;
    ComputeDevice gpuDevice = platforms.SelectMany(p => p.Devices).FirstOrDefault(device => device.Type == ComputeDeviceTypes.Gpu);
    Console.WriteLine($"Selected GPU Device: {gpuDevice.Name}");

    ComputeContextPropertyList properties = new ComputeContextPropertyList(platforms[0]);
    ComputeContext context = new ComputeContext(new ComputeDevice[] { gpuDevice }, properties, null, IntPtr.Zero);
    string kernelCode = Kernel.ApplyShader;
    ComputeProgram program = new ComputeProgram(context, kernelCode);
    program.Build(null, null, null, IntPtr.Zero);

    string[] framePaths = Directory.GetFiles(inputDirectory, "*.png").OrderBy(f => f).ToArray();
    int chunks = Environment.ProcessorCount;
    var framePathChunks = PartitionIntoChunks(framePaths, chunks);

    ComputeKernel kernel = program.CreateKernel("applyShader");
    ComputeCommandQueue queue = new ComputeCommandQueue(context, context.Devices[0], ComputeCommandQueueFlags.None);

    foreach (var framePathChunk in framePathChunks)
    {
        Parallel.ForEach(framePathChunk, framePath =>
        {
            using (Bitmap image = new Bitmap(framePath))
            {
                string fileNameWithoutExtension = Path.GetFileNameWithoutExtension(framePath);
                int width = image.Width;
                int height = image.Height;

                BitmapData bmpData = image.LockBits(new System.Drawing.Rectangle(0, 0, width, height), ImageLockMode.ReadOnly, PixelFormat.Format32bppArgb);
                ComputeBuffer<byte> imageBuffer = new ComputeBuffer<byte>(context, ComputeMemoryFlags.ReadWrite | ComputeMemoryFlags.CopyHostPointer, width * height * 4, bmpData.Scan0);

                lock (lockObject)
                {
                    kernel.SetMemoryArgument(0, imageBuffer);
                    kernel.SetValueArgument(1, width);
                    kernel.SetValueArgument(2, height);
                    queue.ExecuteTask(kernel, null);
                }

                byte[] resultData = new byte[width * height * 4];
                GCHandle resultHandle = GCHandle.Alloc(resultData, GCHandleType.Pinned);
                IntPtr resultDataPtr = resultHandle.AddrOfPinnedObject();
                ComputeBuffer<byte> resultBuffer = new ComputeBuffer<byte>(context, ComputeMemoryFlags.ReadWrite | ComputeMemoryFlags.UseHostPointer, resultData.Length, resultDataPtr);
                queue.ReadFromBuffer(imageBuffer, ref resultData, true, null);

                using (Bitmap processedImage = new Bitmap(width, height, width * 4, PixelFormat.Format32bppArgb, resultDataPtr))
                {
                    string outputPath = Path.Combine(outputDirectory, $"{fileNameWithoutExtension}.png");
                    processedImage.Save(outputPath, ImageFormat.Png);
                }

                resultBuffer.Dispose();
                resultHandle.Free();
                imageBuffer.Dispose();
                image.UnlockBits(bmpData);
            }
        });
    }
    kernel.Dispose();
    queue.Dispose();
    program.Dispose();
    context.Dispose();
}
```

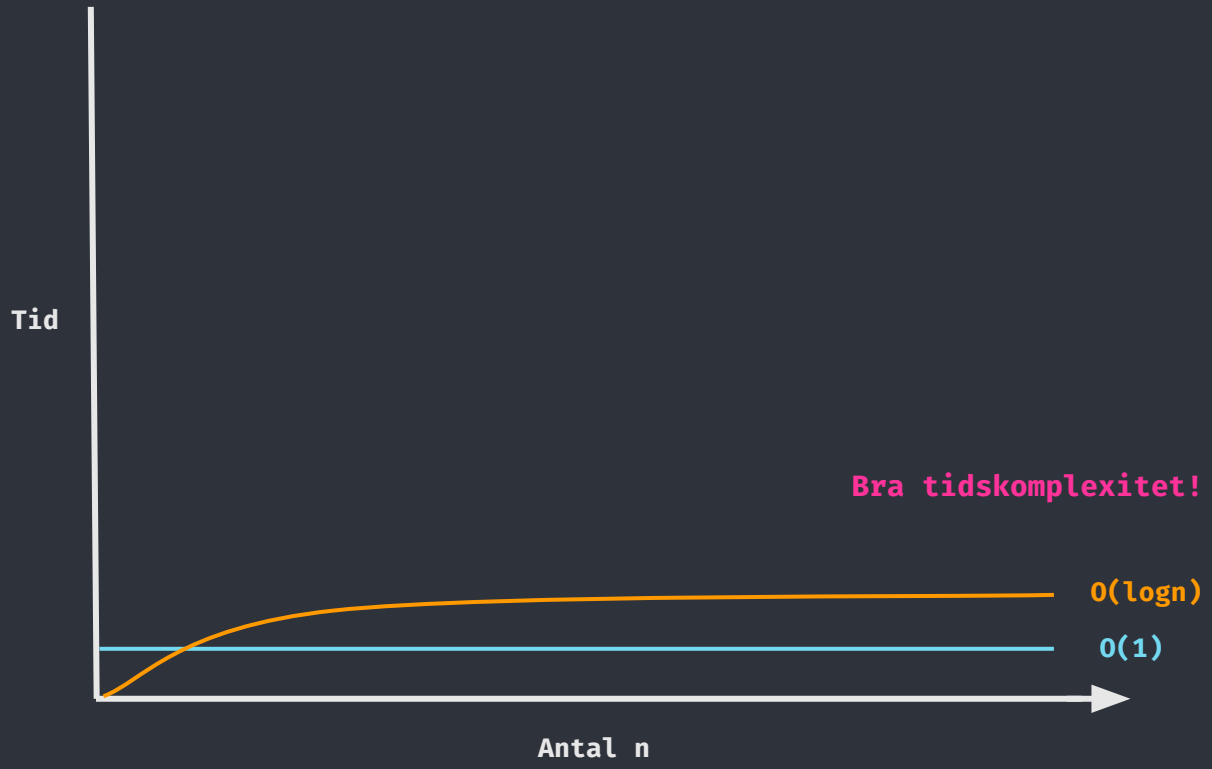
Hela det här kodblocket är en konstant med O(1) tid!

“Your algorithm is O(1) because it never finishes.”

“There exists an n such that your algorithm is fast. Unfortunately, n > RAM.”

Hjälpsamma ord från Stack Overflow 😢

Logaritmisk tillväxt



Logaritmisk tillväxt

- Förra lektionen tittade vi på några exempel på kod som hade **polynomisk tidskomplexitet,** $O(n^2)$: dvs när datamängden dubblas så fyrdubblas körtiden

- Ju längre en sådan algoritm körs desto mindre effektiv blir den, och vid stora n-värden kommer den vara helt värdelös

- Logaritmisk tillväxt, däremot, fungerar tvärtom: **ju större datamängd desto mer effektiv blir den!**

- En logaritm är en inverterad exponent. Ett exempel:

  $10^3 = 1000$
  $\log_{(10)}1000 = 3$

- Datorer representetar tal i binärt (bas-2) till skillnad från oss, så när vi pratar om logfaktor i algoritmer menar vi $\log_2$ men skriver oftast inte ut tvåan

  $2^4 = 16$
  $\log_2 16 = 4$

Logaritmisk tillväxt: O(logn)

– Saker som har tidskomplexiteten O(logn) eller O(log*n) - båda sätten är giltiga att skriva på - växer väldigt långsamt. En logfaktor halverar datamängden varje gång:

```
public int divide(int n)
{
   int steps = 0;

   while (n > 0)
   {
      n = n/2;
      steps++;
   }
}
```

Antal n	steg
1	1
2	2
4	3
8	4
16	5
32	6
64	7
128	8
...	...
1000	10
1.000.000	20
1.000.000.000	30

– O(logn) är så nära O(1) att det är omöjligt att se skillnad på dem rent tidsmässigt: körtiden kommer se konstant ut i en tabell med empiriska värden

Varför bryr vi oss då?

– Vi bryr oss eftersom den ändå spelar roll i kombination med andra faktorer

–

```
public int divide(int n)
{
   int steps = 0;

   while (n > 0)
   {
      n = n/2;
      steps++;
   }
}
```

Antal n	logn	nlogn
1	1	
2	2	
4	3	
8	4	
16	5	
32	6	
64	7	
128	8	
...	...	
1000	10	
1.000.000	20	
1.000.000.000	30	

Arrayer
och Listor

Arrayer: effektiva datastrukturer

- Vi har pratat om tidskomplexitet för algoritmer hittills, men **även datastrukturer har tidskomplexitet:** det tar olika lång tid för olika datastrukturer att hämta ut eller stoppa in värden, söka efter data, osv

- En array är den mest effektiva datastrukturen både tidsmässigt och rent minnesmässigt

- Vi kan läsa från och skriva till valfri plats i den på O(1) tid:

```
array[4] = 16;                  //O(1)

int number = array[1_000_000];  också //O(1)
```

- Det här är eftersom indexet för en array alltid går att beräkna på samma vis. När en array skapas lagras all data i en följd i minnet:

Basadress: **B**
Elementstorlek: **S byte**

- För att få access till `array[i]` beräknar CPU:n bara B + (i * S)

- Alltid en multiplikation, en addition och en minnesskrivning/läsning

Array: Minneslagring

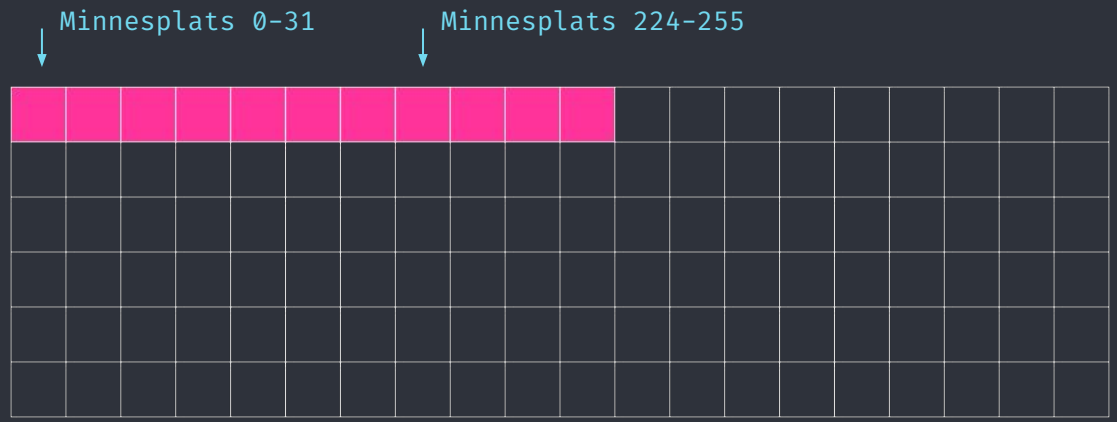
- Man kan utgå från att initieringen av en array **alltid är O(n),** eftersom samtliga element initieras när den skapas: Om vi har en int-array kommer den t.ex. att fylla alla platser med en 0:a

```
int[] array = new int[n];   //O(n)
```

- I en array ligger alla element i en följd, eller ett enda minnesblock: de är inte spridda i minnet utan ligger i en lång sekvens. Den här typen av lagring **kallas för sammanhängande minne** (“contiguous memory” på engelska)

- Om arrayen startar på minnesplatsen 0 och innehåller integers vet vi att element 7 startar på minnesplatsen 224: 0 + (7 * 32)

- Varför multiplicera med 32? Eftersom en int alltid är 4 byte i Java, dvs 32 bit (8 * 4)



Array: Minneslagring

- Eftersom arrayer kräver kontinuerligt eller sammanhängande minne kan vi dock få problem om vi behöver skapa stora arrayer, även i program där det teoretiskt Finns tillräckligt med ledigt minne:

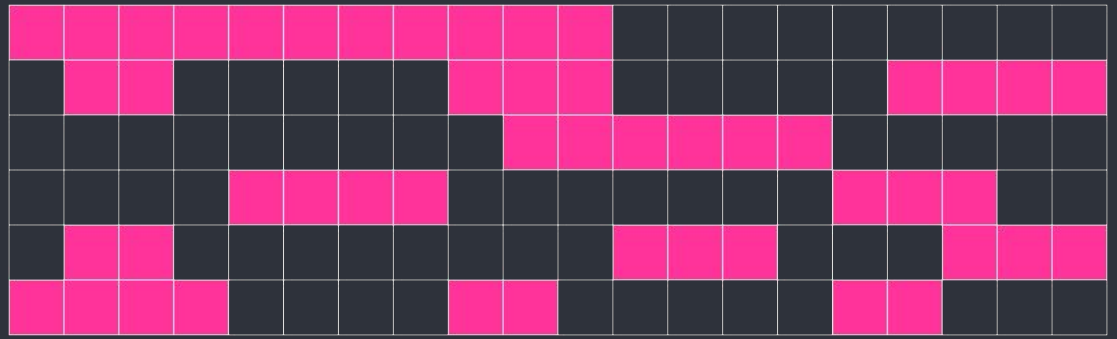
  ```
  int[] array = new int[11];
  ```

- En array med storleken 11 behöver totalt 352 bit i en sekvens (11 * 32)

- I exemplet på den här sliden finns det totalt 2772 lediga bitar (71 * 32) i minnet, men eftersom det inte finns 352 bitar kontinuerligt minne kan vi inte skapa vår array

- Java kastar i så fall ett fel: **"java.lang.OutOfMemoryError: Java heap space"**

  (Noll risk att ni stöter på det här felet i era program!)



Del av RAM-minnet där varje cell representerar 32 bitar

Minneskomplexitet

- Vi har pratat om tidskomplexitet hittills, men **finns även något som heter minneskomplexitet eller platskomplexitet** (“space complexity” på engelska)
- Handlar om hur minneseffektiv en datastruktur är; dvs hur mycket utrymme den tar upp i RAM
- Minne är en dyr resurs i programmering, och det händer ofta att man kompromissar och använder algoritmer med lite sämre tidskomplexitet om de har fördelaktig platskomplexitet
- Vi kommer prata mer om det här senare under kursen och visa mer konkreta exempel
- **Arrayer har bäst platskomplexitet av alla datastrukturer eftersom de är så kompakta:** allt ligger lagrat i sekvens
- Mest minneseffektiva datastrukturen, där vi får denna effektivitet på bekostnad av flexibilitet (går ej att förstora) och dynamiskt beteende (vi kommer out of bounds om vi lägger till för mycket data)

Att loopa genom en array

- Eftersom vi alltid kan skriva och läsa till valfri plats i en array på O(1) får vi O(n) totalt när vi loopar genom en array, oavsett vilken sorts loop vi använder:

```java
public void printArray(int[] data)
{
   for (int i = 0; i < data.length; i++)
   {
      System.out.println(data[i]);
   }
}
```

```java
public void printArray(int[] data)
{
   for (int i : data)
   {
      System.out.println(i);
   }
}
```

- Vi kan aldrig få bättre tidskomplexitet än O(n) när vi loopar genom samlingar, läser från filer, osv om koden kräver att vi går genom alla värden

Listor

- Listor, till skillnad från arrayer, är dynamiska: de kan växa. Vi kan fortsätta lägga till saker i dem vilket gör att de är användbara i situationer där vi inte vet på förhand hur många element vi behöver spara

- Det finns två listor i Javas standardbibliotek som vi kommer att kika på i den här kursen: **LinkedList** och **ArrayList**

- Det finns även en äldre typ som heter **Vector** (namngiven efter C++ version av listor) men den betraktas som deprecated numer och används endast i legacykod som måste underhållas

- Listor är implementationer av ett interface kallat för **List**, och när Vi arbetar med listor bör vi alltid sträva efter att arbeta mot en så generell datatyp som möjligt. Om vi jobbar mot List kan vi enkelt byta ut den konkreta klassen:

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

ArrayList

- En ArrayList heter som den gör eftersom den använder en underliggande array för att spara sin data i

- Den är **egentligen bara en array inbakad i en klass** som sen gör två saker om arrayen blir full:

  * Den skapar en ny array som är dubbelt så stor
  * Den kopierar över all data i den

- **Att skapa och kopiera är dyra operationer,** och vi vill **undvika dem så ofta det är möjligt.** Om vi t.ex. skapar en tom ArrayList och sen har en loop som lägger till tusen värden i den:

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

for (int num : list)
{
    list.add(num);
}
```

... så kommer den att behöva förstora sig själv flera gånger för att kunna lagra fler och fler nummer

ArrayList med förutbestämd storlek

- Ett bättre sätt, om vi vet ungefär hur många element listan ska innehålla, är att definiera storleken vi vill ha direkt när vi initialiserar en ArrayList

- **ArrayList har en överlagrad konstruktor** som kan ta en storlek som argument (överlagrade metoder är sådana som heter samma sak och är av samma typ men har andra parametrar) och listan initieras då till den storleken direkt

Vanlig (tom) konstruktor:

```
ArrayList<Integer> aList = new ArrayList<>();
```

Överlagrad konstruktor där vi anger en storlek på förhand:

```
ArrayList<Integer> aList = new ArrayList<>(1000);
```

- Om vi överskrider den här gränsen kommer listan fortsätta fördubbla sig själv precis som vanligt

Låt oss bygga vår egen ArrayList!

Koden finns i MyArrayList i repot

ArrayList: tidskomplexitet

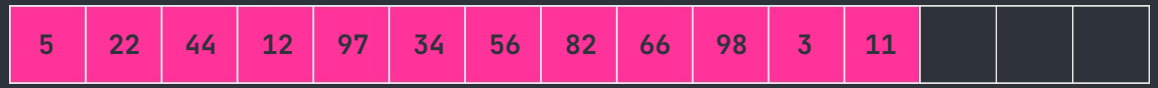
- Eftersom den i princip är en array med lite extra overhead har den ungefär samma styrkor och nackdelar: den går fort att skriva till och läsa från men behöver allt minne allokerat i en enda obruten sekvens
- En ArrayList går dock fortare att lägga till saker i slutet på eftersom den alltid har koll på vilken indexplats det sista elementet ligger lagrat på

Tidskomplexitet för några vanliga operationer:

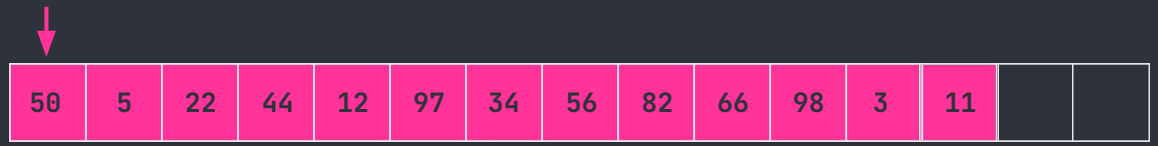
Datatyp	Lägga till först	Lägga till sist	Lägga till på valfri plats	Indexera (hämta ut) valfri plats	Iterera genom
Array	O(n)	O(n)	O(n)	O(1)	O(n)
ArrayList	O(n)	O(1)	O(n)	O(1)	O(n)

Varför O(n) för lägga till först?

- Eftersom vi då måste flytta alla andra värden ett steg åt höger i arrayen (och för att göra det måste vi loopa genom alla värden en gång)
- Både arrayer och ArrayList har samma tidskomplexitet för det här
- Samma med att lägga till någon annanstans i listan: vi behöver i värsta fall flytta på alla eller nästan alla värden, och därför blir tidskomplexiteten O(n) i sämsta fallet




```
list.add(0, 50);          //Lägger till värdet 50 på plats 0
```



Förskjut alla värden ett steg genom att skriva över dem med föregående

Collections i Java

- Collections är Javas standardbibliotek med algoritmer för att hantera listor:

```
import java.util.Collections;
```

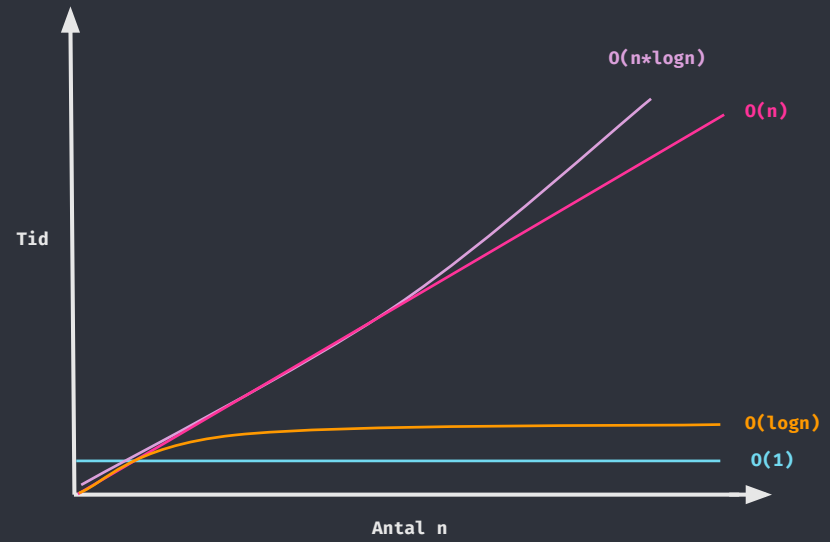
- Precis som System (som vi använder ofta för t.ex. System.out.println()) är den statisk, och vi gör alltså inget objekt av den utan anropar metoder direkt på själva klassen
- Innehåller metod för att sortera listor automatiskt:

```
List<Integer> list = new ArrayList<>(List.of(7, 6, 1, 3, 2));
Collections.sort(list);     //Sorterar i stigande ordning
System.out.println(list);   //Skriver ut [1, 2, 3, 6, 7]
```

- Den här inbyggda sorteringen sorterar på O(n*logn) tid i både värsta och genomsnittliga fallet

n*logn?? Du sa att det hette logn innan!

- **O(n*logn)** innebär att vi går genom alla n, och för varje n gör utför vi en operation som har O(logn) i tid
- Det här är den **bästa möjliga tidskomplexiteten för sorteringar** (om vi förutsätter att samlingen inte redan är sorterad - men är den det behöver den ju inte sorteras!)
- Folk **ritar ofta den här kurvan fel eftersom de inte förstår logaritmer:** många exempel och läroböcker ritar den som någonstans mellan O(n) och $O(n^2)$, men den är i själva verket väldigt nära O(n), på samma vis som O(logn) är väldigt nära O(1)



Jämförelser

– Här är en tabell över hur antalet operationer för en algoritm med den givna tidskomplexiteten ökar när mängden n ökar:

n	O(logn)	O(n)	O(n*logn)	O(n²)
10	~3.3	10	~33	100
100	~6.6	100	~660	10,000
1,000	~10	1000	~10,000	1,000,000
10,000	~13	10 000	~130,000	100,000,000
1,000,000	~20	1,000,000	~20,000,000	1,000,000,000,000
10,000,000	~23	10,000,000	~230,000,000	10^{15}

– För att sätta det här i lite perspektiv: om vi antar att en modern dator kan hantera ca en miljard beräkningar i sekunden skulle **O(nlogn) på sista raden köra färdigt på under en sekund medan O(n^2) skulle ta 27,8 timmar**

Fler bra Collections-grejer

- Bra att använda när ni skriver algoritmer eftersom de både är säkra och optimerade:

```
Collections.min(list);           //Letar upp minsta värdet i en lista

Collections.max(list);           //Letar upp största värdet i en lista

Collections.reverse(list);       //Vänder på elementen i en lista

Collections.shuffle(list);       //Randomiserar elementen i en lista

Collections.frequency(list, “apple”);   //Räknar förekomster av något i en lista

Collections.replaceAll(list, “a”, “b”); //Byter ut ett element mot ett annat
```

- Även bra att använda för att undvika att returnera null-värden i bland:

```
if (results.isEmpty()) { return Collections.emptyList(); }

return results;
```

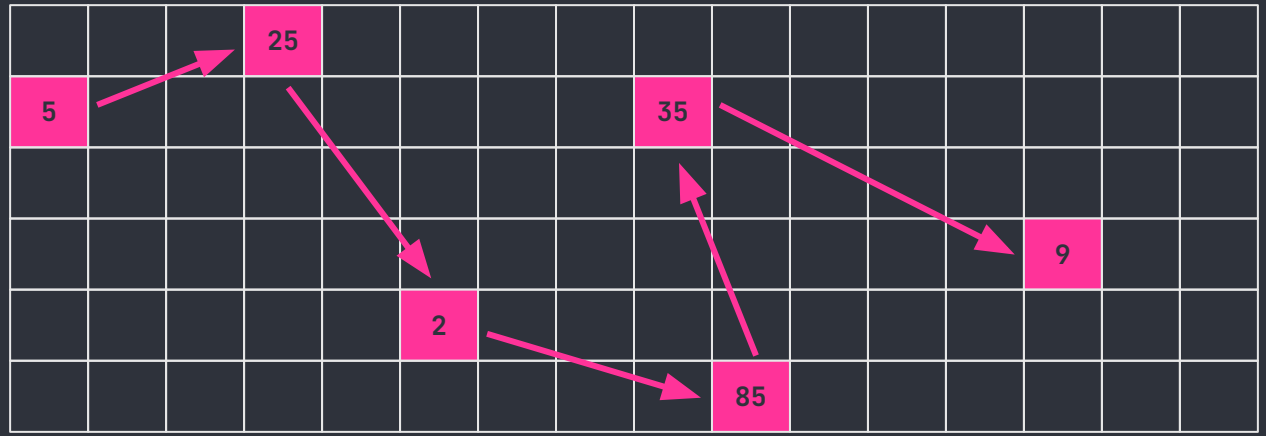
LinkedList

- Det finns ytterligare en lista utöver ArrayList i Javas standardbibliotek
- Man kan tänka på en LinkedList ungefär som en kedja med länkar
- Om man skulle vilja lägga till en länk någonstans skulle man lätt kunna bryta upp två länkar och lägga in en ny mellan dem
- På samma vis fungerar listan: den består av noder där varje nod innehåller en referens till nästa nod, och vi kan lätt skjuta in en länk mellan dem

- Länkade listor fungerar på det här viset eftersom de inte använder någon array under huven för att lagra data: datan kan lagras på vitt spritta platser i minnet

LinkedList: nackdelar

- Vi kommer att gå in mer i detalj på LinkedList och hur den fungerar de kommande veckorna, men ett problem är att vi aldrig vet var nästa värde är lagrat eftersom de är spridda i minnet
- Om vi letar efter ett visst värde måste vi börja från början och loopa genom Listan och besöka noderna en efter en tills vi hittar det: i värsta fall tar det här O(n) tid (värdet ligger sist i listan)



Ett block RAM-minne där olika noder lagrats på vitt skiljda platser

LinkedList: fördelar

- En länkad lista är väldigt snabb på insättning i båda ändarna eftersom den alltid håller koll på dem, och till skillnad från en array eller ArrayList behöver vi inte flytta om alla andra värden eftersom de inte ligger lagrade intill varandra
- Det här innebär att den har O(1) på insättning i ändarna
- Däremot är den oerhört långsam jämfört med de andra två om vi vill hämta ut ett specifikt värde eller ett specifikt index:

Datatyp	Lägga till först	Lägga till sist	Lägga till på valfri plats	Indexera (hämta ut) valfri plats	Iterera genom
Array	O(n)	O(n)	O(n)	O(1)	O(n)
ArrayList	O(n)	O(1)	O(n)	O(1)	O(n)
LinkedList	O(1)	O(1)	O(n)	O(n)	O(n)

LinkedList vs ArrayList

- En bra tumregel är att bara använda en länkad lista om man har en särskild anledning att göra det, såsom att man ofta behöver lägga till data i början, eller om man vill lägga till en länk på en plats som redan är känd
- I alla andra fall är den undermålig, och att använda den utan att veta hur den fungerar innebär ofta att
- **ArrayList är rätt val i 90% av användningsfallen i Java.** För att förstå varför, betrakta följande kod som kan ta antingen en ArrayList eller en LinkedList eftersom båda är typer av List-interfacet i Java:

```
public int summarize(List<Integer> list)
{
   int sum = 0;

   for (int i = 0; i < list.size(); i++)
   {
      sum += list.get(i);
   }
}
```

För ArrayList

* Loopen är O(n)
* get() är O(1)
* O(n) * O(1) = **O(n)**

SNABB ALGORITM

För LinkedList

* Loopen är O(n)
* get() är O(n)
* O(n) * O(n) = **O(n^2)**

KASS ALGORITM

LinkedList vs ArrayList

- En bra tumregel är att bara använda en länkad lista om man har en särskild anledning att göra det, såsom att man ofta behöver lägga till data i början, eller om man vill lägga till en länk på en plats som redan är känd
- I alla andra fall är den undermålig, och att använda den utan att veta hur den fungerar innebär ofta att
- **ArrayList är rätt val i 90% av användningsfallen i Java.** För att förstå varför, betrakta följande kod som kan ta antingen en ArrayList eller en LinkedList eftersom båda är typer av List-interfacet i Java:

```
public int summarize(List<Integer> list)
{
   int sum = 0;

   for (int i = 0; i < list.size(); i++)
   {
      sum += list.get(i);
   }
}
```

För ArrayList

* Loopen är O(n)
* get() är O(1)
* O(n) * O(1) = **O(n)**

SNABB ALGORITM

För LinkedList

* Loopen är O(n)
* get() är O(n)
* O(n) * O(n) = **O(n^2)**

KASS ALGORITM

Enhanced for-loopar

- Eftersom Java är ett så exakt språk kallas den här loopen ibland för **förenklad eller simpel for-loop**, men också för **avancerad for-loop**
- Kallas ibland även för foreach()-loop, vilket är särskilt missvisande, även om den fungerar precis som en foreach() i C#: Java har en egen forEach()-loop som används för att iterera genom strömmar och den fungerar inte som foreach() i C#!
- Jag kallar dem för **enhanced for-loopar,** för det är vad de egentligen heter

```
for (String string : list)
{
    System.out.println(string);
}
```

- **Säkra loopar:** kan aldrig hamna out of bounds
- **Eleganta:** uttrycker avsikten med koden tydligt
- **Optimerade:** Använder iteratorer under huven (ni behöver inte veta vad detta är, men kan läsa på som fördjupning om ni vill)

Enhanced for-loopar

Använd när:

* Ni itererar över samlingar som inte behöver modifieras
* Ni itererar över arrayer
* Enda sättet att iterera över många av datatyperna vi kommer prata om som t.ex. Maps
* Ni vill skriva kod som är renare och tydligare att läsa och förstå

Undvik när:

* Ni behöver direkt indexåtkomst för listan eller arrayen ni itererar över
* När ni behöver plocka bort vissa av elementen ni itererar över
* När ni behöver iterera över något bakifrån
* När ni behöver skriva över värden i samlingen

```
for (String string : list)
{
   string = “Hello!”;
}
```

!

string är en lokal variabel som skapas på nytt i varje loopvarv

Vi skriver över den lokala variabeln men inte strängen i den faktiska listan

Resten av veckan

- Jobba med övningsuppgifterna under modulen för vecka 4 i studium, och gör Laboration 1
- Ni förväntas göra alla övningsuppgifter under kursens gång: ni har ett stort eget ansvar för era studier
- Laboration 2 är nu upplåst och ni bör kunna använda det vi gått genom i dag för att börja lösa den
- Nu när vi introducerat ämnet litegrann bör ni läsa kapitel 3, 4 och 7 i kursboken: **se till att faktiskt göra det.** Att återkoppla regelbundet till vad vi gått genom är det bästa sättet att lära sig på. Kolla genom presentationerna och repo-materialet också
- Nästa vecka kommer vi prata om rekursion, stackar och köer. Kapitel 5 och 6 behandlar dessa ämnen men det är OK om ni vill lyssna på föreläsningarna först innan ni läser sidorna i boken (rekommenderas att ta det i denna ordning!)

Namn på olika tidskomplexiteter (hårdplugga det här så att ni kan det utantill)

- **Konstant, O(1),** innebär att antalet operationer aldrig ökar när storleken på indatan ökar

- **Logaritmisk tid, som betecknas O(logn) eller O(log*n),** växer oerhört långsamt

  En logaritm är en inverterad exponent: **$\log_2 8 = 3$, eftersom $2^3 = 8$**

  Vi menar $\log_2 n$ när vi pratar om logn i algoritmdesign men skriver inte ut tvåan: Siffran är basen, dvs talsystemet, och datorer räknar ju binärt som är bas-2.

  När man pratar om logaritmer i matematik menar man däremot ofta $\log_{10}$ eftersom vårt normala talsystem i riktiga världen är bas-10.

- **Linjär tid, O(n),** innebär att tiden ~dubblas när indatan dubblas

- Vi kallar alla algoritmer som växer med en faktor **n^k för polynomisk tid.**
  Exempel: O(1), O(n), O(n^2), O(n^3), O(n^4), osv. Logaritmer är inte polynomiska.