Länkade strukturer, parametriserade typer och undantag

Programmering för språkteknologer 2

Sara Stymne

2013-09-18

Idag

- Parametriserade typer
- Listor och länkade strukturer
- Komplexitet i länkade strukturer resp. arrayer
- Undantag

Parametriserade typer (Generics)

- Exempel:

  `ArrayList<Integer>`

  `ArrayList<String>`

  `ArrayList<...>`
- Praktiskt att kunna
  - använda samma lista för alla typer
  - återanvända kod

Parametriserade typer (Generics)

- Exempel:

  `ArrayList<Integer>`

  `ArrayList<String>`

  `ArrayList<...>`
- Praktiskt att kunna
  - använda samma lista för alla typer
  - återanvända kod
- Man kan även skapa egna generiska klasser och metoder!
- Det kallas att parametrisera en klass/metod

Parametriserade type (Generics)

- En ArrayList beter sig likadant oavsett vad du lagrar i den
- Implementationen är inte beroende av innehållet
- Innehållets typ introduceras i en "platshållare" <...>,
  dvs en parameter och kan bestå av godtycklig klass
- När vi skapar en konkret instans anger vi vad det ska vara för innehåll
- Exempel:

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

Parametrisering på olika nivå

- Generiskt interface (gränssnitt) – Ett gränssnitt som kan hantera olika typer
- Generisk klass – En klass som kan hantera olika typer
- Generisk metod – En metod som kan hantera olika typer

Parametriserad metod

```
public <T> append(ArrayList<T> from, ArrayList<T> to) {
   for (T value: from) {
      to.add(value);
   }
}

// anrop, antag att ArrayList<String> a och b finns
append(a, b);
```

Parametriserad klass

```
public class MyClass<T>{

  T someVariable;
  ArrayList<T> someListVariable;

  public append(ArrayList<T> from, ArrayList<T> to) {
     for (T value: from) {
        to.add(value);
     }
  }

  public void foo( {
     // anrop, antag att ArrayList<String> a och b finns
     MyClass<String> c = new MyClass<String>;
     c.append(a, b);
  }
}
```

Parametriserade typer – användning

- Vad kan man göra med parametriserade klasser?
- Inte så mycket. De kan ju tillhöra vilken klass som helst, så vi kan inte anta att några metoder finns, förutom de i klassen Object
- Några exempel på problem:
  - `new T(...)` – omöjligt vi kan inte skapa nya objekt av T
  - `compareTo(T)` – går ej att använda, eftersom den inte finns för alla typer, men det går att lösa!

Bundna typer

- Det går att binda en parametriserad typ till ett interface eller en klass
- Exempel, vi vill kunna använda `compareTo(T)`
  - Definierad i gränssnittet (interface) `Comparable`
  - Skriv public class `MyClass<T extends Comparable<T>{ ...`

Bundna typer

- Det går att binda en parametriserad typ till ett interface eller en klass
- Exempel, vi vill kunna använda `compareTo(T)`
  - Definierad i gränssnittet (interface) `Comparable`
  - Skriv `public class MyClass<T extends Comparable<T>{ ...`
  - Nu kan den parametriserade klassen bara användas för klasser som implementerar `Comparable`, och därmed kan metoden `compareTo(T)` användas!

Abstrakta datatyper

- Abstrakt data type (ADT)
- Ett teoretiskt begrepp
- Definieras som
  - En typ
  - En lista med operationer som ska kunna utföras
- Den abstrakta typen säger ingenting om hur typen ska implementeras, bara om vad man ska kunna göra med den

Abstrakta datatyper

- Abstrakt data type (ADT)
- Ett teoretiskt begrepp
- Definieras som
  - En typ
  - En lista med operationer som ska kunna utföras
- Den abstrakta typen säger ingenting om hur typen ska implementeras, bara om vad man ska kunna göra med den
- I Java motsvaras en ADT oftast av ett interface, som beskriver de metoder som ska finnas för ADT:n

Abstrakta datatyper – lista

- Lista är en abstrakt datatyp
- Motsvarar det matematiska begreppet sekvens
- En lista är en **finit, ordnad sekvens** av dataobjekt

Abstrakta datatyper – lista

- Lista är en abstrakt datatyp
- Motsvarar det matematiska begreppet sekvens
- En lista är en **finit, ordnad sekvens** av dataobjekt
- Ordnad på så sätt att alla objekt har en plats, inte nödvändigtvis sorterad

Abstrakta datatyper – lista

- Lista är en abstrakt datatyp
- Motsvarar det matematiska begreppet sekvens
- En lista är en **finit, ordnad sekvens** av dataobjekt
- Ordnad på så sätt att alla objekt har en plats, inte nödvändigtvis sorterad
- Man kan definiera vilka metoder som ska finnas för en lista

List-gränssnittet

- I Java motsvaras ADT:n lista av gränssnittet (Interface) List
- Innehåller bland annat följande metoder:
  - boolean isEmpty()
  - boolean add(E element)
  - boolean add(int index, E element)
  - E get (int index)
  - E remove (int index)
  - ...

List – implementation

- Hur en List implementeras är ej specificerat
- Två vanliga implementationer:
  - Med hjälp av en array
  - Som en länkad struktur

List – implementation

- Hur en List implementeras är ej specificerat
- Två vanliga implementationer:
  - Med hjälp av en array – ArrayList
  - Som en länkad struktur – LinkedList

RITA BILDER!!

Lista som array

- Implementerad i Java som ArrayList<T>
- Använder internt en array T[]
- Default-storlek: 10
- När arrayen blir full, så skapas en större array, och alla värden kopieras över dit (tar tid)
- I en array så har vi ”random access” det vill säga kan komma åt element var som helst i arrayen på konstant tid

Länkade strukturer

- Länkad lista
  - En struktur där en lista byggs upp med hjälp av en länkad struktur
- Varje element känner till sitt värde och vad nästa element är
- Den som hanterar listan behöver enbart känna till det första elementet

Länkad lista – implementation

```
//Nodklass:
public class Node<T> {
   T value;
   Node next;

   Node (Node next, T value) {
      this.next = next;
      this.value = value;
   }
}

//Listklass
public class MyLinkedList<T> {
   private Node<T> head = null;

   ...
}
```

Nodklassen

- Används för att representera varje nod
  - `next` pekar på nästa nod, eller är `null` om listan är slut
- Kan ligga som en **intern klass** i listan, eftersom den inte behöver användas utanför listklassen

Länkad lista – intern nodklass

```
//Listklass
public class MyLinkedList<T> {
   //Nodklass:
   private class Node {
      T value;
      Node next;

      Node (Node next, T value) {
         this.next = next;
         this.value = value;
      }
   }

   private Node<T> head = null;

   ...
}
```

Länkad lista – operationer

- Hämta värde på en plats
- Insättning av nytt värde
- Ta bort värde

Länkad lista – operationer

- Hämta värde på en plats
- Insättning av nytt värde
- Ta bort värde

Gå igenom på tavlan!!!

Exempel – sätt in nytt värde först i lista

```
public class MyLinkedList<T> {
    private Node<T> head;

    public void insertFirst(T value) {
        Node n = new Node(head, value);
        head = n;
    }
}
```

Exempel – skapa en sträng av alla värden

```
public class MyLinkedList<T> {
   private Node<T> head;

   public String toString() {
      Node current = head;
      StringBuilder res  = new StringBuilder();
      while (current != null) {
         res.append(current.value + " ");
         current = current.next;
      }
      return res.toString();
   }
}
```

Olika typer av länkade listor

- Enkellänkad lista
  - Varje element känner till nästa element
  - Klassen känner till det första elementet
- Dubbellänkad lista
  - Varje element känner till både nästa och föregående element
  - Klassen känner till det första och sista elementet
  - Javas LinkedList är dubbellänkad

Olika typer av länkade listor

- Enkellänkad lista
  - Varje element känner till nästa element
  - Klassen känner till det första elementet
  - Klassen kan möjligen känna till det sista elementet också
- Dubbellänkad lista
  - Varje element känner till både nästa och föregående element
  - Klassen känner till det första och sista elementet
  - Javas LinkedList är dubbellänkad

Enkellänkad lista – tidskomplexitet

- Hämta värde
  - Först i listan: $O(1)$
  - På plats x: $O(n)$

Enkellänkad lista – tidskomplexitet

- Hämta värde
  - Först i listan: $O(1)$
  - På plats x: $O(n)$
- Insättning/borttagning av värde
  - Först i listan: $O(1)$

Enkellänkad lista – tidskomplexitet

- Hämta värde
  - Först i listan: $O(1)$
  - På plats x: $O(n)$
- Insättning/borttagning av värde
  - Först i listan: $O(1)$
  - På plats x:
    - Hitta rätt plats: $O(n)$
    - Själva insättningen/borttagnigen: $O(1)$

Enkellänkad lista – tidskomplexitet

- Hämta värde
  - Först i listan: $O(1)$
  - På plats x: $O(n)$
- Insättning/borttagning av värde
  - Först i listan: $O(1)$
  - På plats x:
    - Hitta rätt plats: $O(n)$
    - Själva insättningen/borttagnigen: $O(1)$
  - Sist i listan: $O(1)$ om vi har pekare till sista elementet, $O(n)$ annars

Andra länkade strukturer

- Länkade strukturer kan användas till annat än länkade listor
- Exempelvis binära träd
- Istället för att peka på nästa nod, pekar varje nod på höger resp. vänster barn

Andra länkade strukturer

- Länkade strukturer kan användas till annat än länkade listor
- Exempelvis binära träd
- Istället för att peka på nästa nod, pekar varje nod på höger resp. vänster barn
- Binära träd är inte listor, man kan inte komma åt värden på index på något vettigt sätt

Andra länkade strukturer

- Länkade strukturer kan användas till annat än länkade listor
- Exempelvis binära träd
- Istället för att peka på nästa nod, pekar varje nod på höger resp. vänster barn
- Binära träd är inte listor, man kan inte komma åt värden på index på något vettigt sätt
- Rita bild!

Länkat binärt träd – implementation

```
//Nodklass:
public class Node<T> {
   T value;
   Node right;
   Node left;

   Node (Node r, Node l, T value) {
      right = r;
      left = l;
      this.value = value;
   }
}

//Listklass
public class MyLinkedBinaryTree<T> {
   private Node<T> root;

   ...
}
```

Binära sökträd

- Binära sökträd är ett specialfall av binära träd, där nodernas värden ligger i en sorterad ordning
- Har ibland värde+nyckel som hashtabell, men kan även bara ha värden
- Om trädet är balanserat är det effektivt att söka efter värden: $O(\log\, n)$

Binära sökträd

- Binära sökträd är ett specialfall av binära träd, där nodernas värden ligger i en sorterad ordning
- Har ibland värde+nyckel som hashtabell, men kan även bara ha värden
- Om trädet är balanserat är det effektivt att söka efter värden: $O(\log n)$
- Det finns metoder för att balansera träd
  - AVL-träd
  - Röd-svarta träd
  - ...

Binära sökträd

- Binära sökträd är ett specialfall av binära träd, där nodernas värden ligger i en sorterad ordning
- Har ibland värde+nyckel som hashtabell, men kan även bara ha värden
- Om trädet är balanserat är det effektivt att söka efter värden: $O(\log n)$
- Det finns metoder för att balansera träd
  - AVL-träd
  - Röd-svarta träd
  - ...
  - Detaljer är överkurs!

Komplexitet för olika listimplementationer

	Array	Länkad lista
Iteration	$O(n)$	$O(n)$
Hämta värde på index x	$O(1)$	$O(n)$
Sätta in värde först	$O(n)$	$O(1)$
Sätta in värde på index x	$O(n)$	$O(n)$
Sätta in värde sist	$O(1)$	$O(1)$ el. $O(n)$
Ta bort värde först	$O(n)$	$O(1)$
Ta bort värde på index x	$O(n)$	$O(n)$

Komplexitet för olika listimplementationer

	Array	Länkad lista
Iteration	$O(n)$	$O(n)$
Hämta värde på index x	$O(1)$	$O(n)$
Sätta in värde först	$O(n)$	$O(1)$
Sätta in värde på index x	$O(n)$	$O(n)$
Sätta in värde sist	$O(1)$	$O(1)$ el. $O(n)$
Ta bort värde först	$O(n)$	$O(1)$
Ta bort värde på index x	$O(n)$	$O(n)$

Komplexitet för olika listimplementationer

	Array	Länkad lista
Iteration	$O(n)$	$O(n)$
Hämta värde på index x	$O(1)$	$O(n)$
Sätta in värde först	$O(n)$	$O(1)$
Sätta in värde på index x	$O(n)$	$O(n)$
Sätta in värde sist	$O(1)$	$O(1)$ el. $O(n)$
Ta bort värde först	$O(n)$	$O(1)$
Ta bort värde på index x	$O(n)$	$O(n)$

- För borttagning/insättning krävs färre flyttar i en länkad lista, dock fler jämförelser
- En ArrayList som blir full behöver förstoras, vilket tar $O(n)$

Komplexitet för loop

```
for (int i=0; i<lista.size(); i++) {
   //Do something with lista[i]
}
```

Komplexitet för loop

```
for (int i=0; i<lista.size(); i++) {
   //Do something with lista[i]
}
```

	Array	Länkad lista
loop med indexering	$O(n)$	$O(n^2)$

Komplexitet för loop

```
for (E value: lista) {
   //Do something with value
}
```

	Array	Länkad lista
Foreach-loop	$O(n)$	$O(n)$

Komplexitet för loop

```
for (E value: lista) {
   //Do something with value
}
```

	Array	Länkad lista
Foreach-loop	$O(n)$	$O(n)$

- For-each loopen använder en iterator som stegar igenom listan.
- Kommer att gås igenom nästa föreläsning

Sortering i listor

- De sorteringsalgoritmer vi gick igenom förra föreläsningen fungerar för arrayer, eftersom de bygger på ”random access”
- Väldigt ineffektiva för länkade strukturer
- Vissa av dem går dock att modifiera för länkade strukturer
  - Insertion sort $O(n^2)$
  - Merge sort $O(n \log n)$

Sortering i listor

- De sorteringsalgoritmer vi gick igenom förra föreläsningen fungerar för arrayer, eftersom de bygger på ”random access”
- Väldigt ineffektiva för länkade strukturer
- Vissa av dem går dock att modifiera för länkade strukturer
  - Insertion sort $O(n^2)$
  - Merge sort $O(n \log n)$
- Alternativ: stoppa in värdena sorterat i listan (smidigare att göra i länkad lista än i array) – men ännu bättre att använda binärt sökträd, kanske

Komplexitet, lista + sökträd

	Array	Länkad lista	Binärt sökträd
Itereration	$O(n)$	$O(n)$	$O(n)$
Sök efter värde	$O(n)$	$O(n)$	$O(\log n)$
Hämta värde på index x	$O(1)$	$O(n)$	–
Sätta in värde sorterat	$O(n)$	$O(n)$	$O(\log n)$
Sätta in värde på index x	$O(n)$	$O(n)$	–
Sätta in värde sist	$O(1)$	$O(1)$	–
Ta bort givet värde	$O(n)$	$O(n)$	$O(\log n)$

Vilken datastruktur ska man välja?

- Beror på uppgiften!
- Tänk igenom vilken typ av operationer du behöver använda och hur din data ser ut och kommer användas

Vilken datastruktur ska man välja?

- Beror på uppgiften!
- Tänk igenom vilken typ av operationer du behöver använda och hur din data ser ut och kommer användas
- Om man ska söka väldigt många gånger efter olika värden
  - Binärt sökträd eller hashtabell

Vilken datastruktur ska man välja?

- Beror på uppgiften!
- Tänk igenom vilken typ av operationer du behöver använda och hur din data ser ut och kommer användas
- Om man behöver loopa igenom alla värden ofta, men inte söka efter specifika värden
  - lista går bra

Vilken datastruktur ska man välja?

- Beror på uppgiften!
- Tänk igenom vilken typ av operationer du behöver använda och hur din data ser ut och kommer användas
- Om man bestämt sig för en lista
  - Om man vet ungefär hur många värden man ska jobba med, och antalet värden är relativt konstant
    - Array
  - Om antalet värden är okänt, och varierar mycket under programmets livstid
    - Länkad lista kan vara bra
  - Om man ofta behöver sätta in värden mitt i listan
    - Länkad lista (men man bör även överväga hashtabell tex)

Undantag / Exceptionella händelser

- Exception på engelska
- Ett sätt att hantera fel utan att programmet måste krascha
- När något märkligt händer kan man ”kasta ett undantag” som någon annan får ta hand om

Exempel – undantagshantering

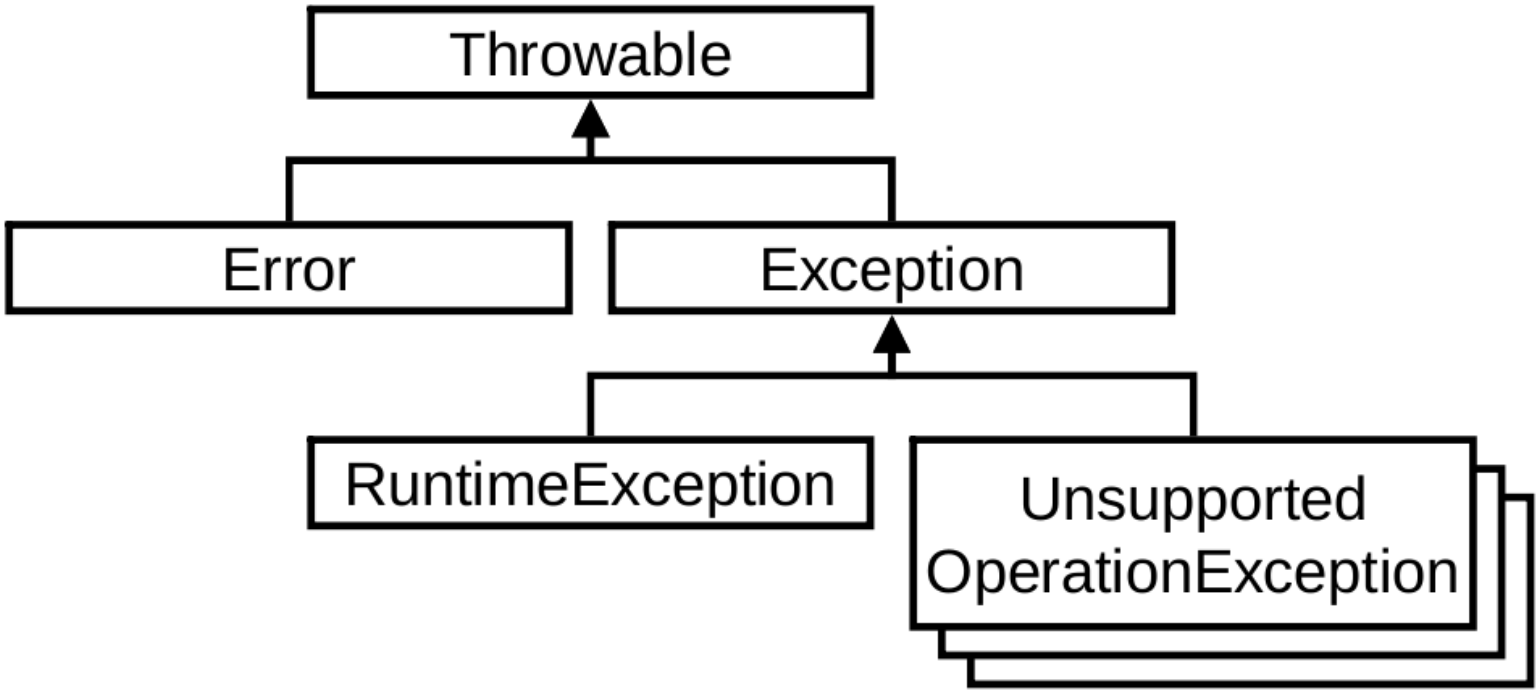
```
ArrayList<String> sentences = new ArrayList<String>();

try {
   Scanner in = new Scanner(new File("minfil.txt"));
   while (in.hasNextLine()) {
      sentences.add(in.nextLine());
   }
}
catch (FileNotFoundException e) {
   System.out.println("Kunde inte hitta filen " + args[0]);
   return; // metoden avslutas -- räcker det?
}
```

Att kasta undantag

- Undantag kastas med hjälp av det reserverade ordet `throw`, följt av ett undantagsobjekt
- En metod som innehåller ett throw-uttryck måste deklarera att den kan kasta det aktuella undantaget
- Detta görs med det reserverade ordet `throws` följt av en lista med undantagsklasser

```
public void foo() throws UnsupportedOperationException {
   throw new UnsupportedOperationException();
}
```

Undantagsklasser



Feltyper

- Error
  - Helt utanför programmerarens kontroll
  - Till exempel slut på minne
- RuntimeException
  - Uppstår på grund av dålig programmering
    - NullPointerException
    - ArrayIndexOutOfBoundsException
- Övriga Exception
  - Beror på andra orsaker, till exempel användarinmatningar, filläsning
    - FileNotFoundException

Feltyper

- Error
  - I princip ohanterliga
- RuntimeException
  - Korrigera programmet så att de inte uppstår
- Övriga Exception
  - Hantera felet!

Hantera felet

- Aktiv hantering
  - Upptäcka att det inträffade
  - Vidtag adekvat åtgärd
- Passiv hantering
  - Kasta vidare felet
  - Det bör dock hanteras någon annastans!

Aktiv hantering – upptäcka felet

- Försök utföra de satser där felet kan uppstå
- Var beredd på att fånga upp eventuella fel som kastas

```
try {
// ...
} catch (Exception e) {
// ...
}
```

Adekvat åtgärd

- Utför något i catch-satsen
  - Skriv ut ett felmeddelande
  - Avsluta programmet
  - Initialisera en variabel annorlunda
  - Kasta ett nytt undantag
  - ...
- Se till att felet inte påverkar resten av programkörningen (om du ej avslutar)

Passiv hantering

- Deklarera att metoden kastar ett undantag

```
ArrayList<String> readFileContents(String fileName)
                        throws FileNotFoundException  {

   // kod

   throw new FileNotFoundException();
}
```

Vilka fel måste hanteras

- Alla metoder kastar implicit `Error` och `RuntimeException`
- Dessa behöver inte fångas
- Men de går att fånga om man vill
  - Kan vara ett sätt att sköta sin felhantering
- Alla andra undantag **måste** hanteras!

Nya undantagstyper

- Mitt program genererar fel som inte passar in på någon befintlig undantagsklass
- Skapa en egen undantagsklass!

Nya undantagstyper

- Mitt program genererar fel som inte passar in på någon befintlig undantagsklass
- Skapa en egen undantagsklass!
- Ärv från någon befintlig Exception-klass, de är konstruerade som bra basklasser!

Nya undantagstyper

- Mitt program genererar fel som inte passar in på någon befintlig undantagsklass
- Skapa en egen undantagsklass!
- Ärv från någon befintlig Exception-klass, de är konstruerade som bra basklasser!
- Du behöver bara skriva de konstruktorer du behöver använda

Exempel – egen undantagsklass

```
public class MyException extends Exception {
  public MyException() {
     super();
  }
  public MyException(String msg) {
     super(msg);
  }
}
```

Kasta och fånga undantag

- Kasta med eller utan meddelande
  - `throw new MyException();`
  - `throw new MyException("Ett bra informativt meddelande");`
- När man fångar kan man få information om undantaget
  - getMessage() – ger meddelandet, eller `null` om det inte finns
  - toString() – gör om exception till en sträng, innehåller exceptionnamnet och ev. meddelande

Kasta och fånga undantag

- Kasta med eller utan meddelande
  - `throw new MyException();`
  - `throw new MyException("Ett bra informativt meddelande");`
- När man fångar kan man få information om undantaget
  - getMessage() – ger meddelandet, eller `null` om det inte finns
  - toString() – gör om exception till en sträng, innehåller exceptionnamnet och ev. meddelande

```
catch (MyException e) {
System.out.println("Det här gick fel " + e);
// implicit användining av e.toString()
}
```

Lab 2, del 2

- Sökning
- Ni får en given klass med kod för
  - linjär sökning
  - binär sökning, rekursiv
  - binär sökning, iterativ
- Koden är skriven för arrayer med heltal, int[]
- Uppgift
  - Gör om klassen så att den blir generisk
  - Gör om koden så att den fungerar för generisk ArrayList<T>

Kommande veckor

- Sökning och sortering
  - Deadline lab 2: 27/9 (fredag)
- Tema datastrukturer, paket, iteratorer mm (v. 39-41)
  - Föreläsning om iteratorer, stackar och köer mm
  - Två labtillfällen

Jobba själv

- Labbar
  - Lab 2
- Gör progammeringsövningar
  - Från boken
- Läs om veckans område
  - Sökning
  - Sortering
  - Komplexitet
- Läs inför nästa vecka
  - Parametriserade typer
  - Länkade strukturer
  - Stackar och köer
  - Undantag