Vad ska du kunna?

***Tidskomplexitet:***

* förstå begreppen: *algoritm*, *effektivitet*, *tidskomplexitet*.

algoritm - betyder metod för att stegvis lösa ett problem.

effektivitet - Att göra den så snabb och icke krävande som möjligt.

tidskomplexitet (tillväxthastighet) - hur lång tid det tar att lösa en algoritm.

* beräkna *tidskomplexitet T(n)* för enklare algoritmer där alla operationer tar konstant tid.
* ha en uppfattning om vad olika storleksordningar av tidskomplexiteten innebär (konstant, logaritmisk, linjär, kvadratisk etc.)

konstant - det tar lika lång tid hela tiden

logaritmisk - delas med N hela tiden.

linjär - kurvan ökar

kvadratisk - kurvan ökar med n^2 tex.

1 < log n < N < n log n < n^2 < n^3 < 2^n

***Sorterings algoritmer och binärsökning :***

* redogöra för och jämföra olika sorteringsalgoritmer:
* *Insättningssortering (InsertionSort)*

InsertionSort är en kvadratisk (n^2) sorteringsalgoritm. (kvadratisk)

Den pekar på ett värde och går framåt ett steg och jämför med den. Är den mindre så byter de plats. Ju mer data som är redan sorterade desto snabbare går det. Den "inre"-loopen kommer inte köras.

* *Heapsort* (behandlas i samband med prioritetsköer).

Är av order N log N. (subkvadratisk.)

Bygger heap genom att kopiera data till en heap. Hoppar igenom alla föräldrar och börjar med den sista för att få största värdet som föräldrar, man byter alltså med barn och föräldrar till man fått en heap-egenskap.

När man fått den största på toppen, så lägger man den till sista värdet i arrayen. Sedan roterar man upp den största vördet till toppen och lägger den nästsist osv..

* *Mergesort*, rekursiv variant

Är av order N log N (subkvadratisk).

Den delar arrayen i två halvor, sortera halvorna och sedan sätter ihop halvorna (mergar) i en sorterad array. Den delar hela arrayen tills den inte kan delas mer. När man kommit så långt att alla objekt är delade till egna arrayer så sätter man ihop dem med varandra sorterade.

**Nackdel**: Använder mycket minne för att skapa temporära arrayer för varje rekursion.

* *Quicksort*

Är av order N log N (subkvadratisk)

Delar arrayen i två delar genom att välja ett element i arrayen som pivot. Element som är mindre än pivoten flyttas till vänstra delen och de större till högra. Sedan sorteras de på det viset.

**Nackdel**: Arrayen delas alltid i en tom del och en del som består av alla element utom pivoten. Då blir det n^2 (händer väldigt sällan):

* förklara samt kunna implementera binär sökning algoritmen i arrayer

Binär sökning förutsätter att datan är sorterad.

**public** **static** **int** binS(**int**[] a, **int** x){

**int** not\_found = -1;

**int** low = 0;

**int** high = a.length-1;

**int** mid;

**while**(low <= high){

mid = (low + high)/2;

**if**(x > a[mid]){ low = mid + 1; }

**else** **if** (x < a[mid]){high = mid - 1;}

**else** **return** mid;

}

**return** not\_found;

}

Sätt ett värde utifall den inte hittas. Low ska sättas till 0, minsta värdet. High blir högsta alltså arrayens längd - 1.

Så länge low är mindre än high så ska vi köra.

Sätt mid till (low+high)/2.

Jämför indexet med storleken i arrayen på mitten platsen. Är den större så sätt low till mid+1, den är alltså större än mitt talet.

Tvärtom om x är mindre än mitten, då ändrar man high till mid - 1.

***Abstrakta datatyper och Listor***

förstå begreppen: *abstraktion*, *abstrakt datatyp*, *specifikation*, *implementation*.

**abstraktion** - tänk bort vissa egenskaper hos ett föremål och lyft fram andra. Man kan säga att man döljer detaljer.

**abstrakt datatyp** - specifierar operationer för åtkomst och ändring.

Exempel på abstrakt datatyp:

saldo = kronor(); // Ett objekt av en abstrakt datatyp kronor

saldo.set(1500); // Sätter ett nytt värde

saldo.plus(1500); // Ändrar värdet.

saldo.get(); // Ger värdet.

**specifikation** - Vad programmet innehåller och gör

**implementation** - en praktiskt tillagd metod, algoritm eller dylikt för att uppnå ett specifkt resultat.

* definiera och implementera *gränssnitt* (*interface*).

Det handlar om någon form av "kontakt" eller "kommunikation". En eller flera tomma metoder som MÅSTE implementeras aav andra klasser för att garantera vissa egenskaper för ett objekt.

**public** **interface** Jump\_able{ //Skapar en egen interface

**public** **void** jump();

}

**class** Ball **implements** Jump\_able { //Implementerar till ball

<metoder, variabler osv>

}

* implementera interfacet Comparable, Comparator

Comparable använder sig av **compareTo**.

**public** **class** Question **implements** Comparable {

<metoder>

}

**public** **int** compareTo(Object x){

Question q = (Question) x; //Konverterar x till Question.

//Jämförelsen här..

}

--------------------------------

**Comparator** måste importeras genom import och använder metoden **compare**

**public** **class** Question **implements** Comparator {

<metoder>

}

**public** **int** compare(Object arg0, Object arg1) {

String name1 = arg0.getName();

String name2 = arg1.getName();

**return** name1.compareTo(name2);

}

* ta hand om *undantag* (*exceptions*).

Antingen gör man det genom **try / catch**-metoden:

**try** {

<Gör något>

}

**catch**(Exception ***e***) {

**return** ***e***;

}

--------------------------------

eller **throw**-metoden:

**public** **void** Jump\_able **throws** NullPointer {

<Gör något>;

**throw** **new** Exception();

}

* generera undantag.

Den här metoden "kollar" om ett namn finns med i en array. Finns det redan, så generar den ett undantag för namnet får inte finnas 2 gånger.

**public** **void** findAndInsert(Object name) {

**if**(name.equals(Array)){

**throw** **new** DuplicateItemException(name.toString()); //Dubbellett ej okej.

**else** {

add(name);

}

}

}

* veta vad en *lista*, *kö* och *stack* är för någonting och ungefär vilka operationer man kan utföra på dem.

**lista -** är en ordnad samling av objekt.

lista har ofta operationer som:

- add

- get

- remove

- iterator

**länkad lista -**  består av ett antal objekt och noder som sammanlänkar genom att varje nod referar till nästa nod. En länkad lista har oftast operationer som:

- add, get, remove, find, iterator, cotains, size

node har ofta operationer som:

- head

- next

**kö** - är en ordnad samling av objekt. Man kan bara komma åt den första i kön, och lägga till sist i kön. FIFO (first in, first out)

Operationer:

- enque (Lägg till)

- deque (Ta bort)

**stack** - är en följd av element där borttagning av ett element alltid avser det element som senast satts in, LIFO. (Last In, First Out.)

Operationer:

- push (Lägger till på stacken.)

- pop (Tar bort det som ligger högst upp)

- peek (Kollar vad som är högst upp.)

- isEmpty (om den är tom.)

- makeEmpty (Töm stacken.)

* implementera en lista, stack eller kö med hjälp av en array. Förstå hur array storlek kan ökas när det behövs.

Finns i eclipse.

* implementera en lista, stack eller kö med en *länkad struktur*.

Finns i eclipse.

* förstå begreppet *generisk lista* och kunna implementera en sådan.

En generisk lista skapas med <AnyType> och kan vara vilken datatyp som helst.

**public** **class** Node<AnyType> {

Node[] test = (Node[]) new Object[0];

}

* förstå vad en *iterator* är och kunna använda en sådan.

En iterator möjliggör en sekventiell genomgång eller bearbetning av element.

Iterator itr = lista.iterator();

itr.hasNext() osv.

* implementera en iterator.

import java.util.Iterator;

**public** **class** LinkedListX **implements** Iterator {

<Metod>;

}

**public** **boolean** hasNext() {

**return** header.next != **null**;

}

**public** Object next() {

Node temp = header;

**return** temp.next;

}

* använda Javas standardbibliotek med klasser och interface som beskriver samlingar av element (*the Collection API*), t ex Iterator, ListIterator, LinkedList och ArrayList.
* förstå hur valet av datastruktur påverkar effektiviteten.
* beräkna tidskomplexitet för algoritmer som använder abstrakta datatyper.

***Rekursion:***

förstå begreppen: *rekursion*, *basfall*,

**rekursion** - är en metod som anropar sig själv.

**basfall**  - Är när metoden ska sluta.

* förstå hur rekursion fungerar.

Rekursion fungerar på så sätt att den anropar sig själv tills de är klart.

**public** **static** **int** sumOf(**int** n){

**if**(n==1) **return** 1;

**else** {

**return** somOf(n-1)+n;

}

}

* formulera rekursiva algoritmer och skriva rekursiva metoder.

se ovan.

***Träd, binära träd:***

* förstå begreppen: *träd*, *nod*, *båge*, *förälder*, *barn*, *löv*.

träd -

nod -

båge -

förälder -

barn -

löv -

* förstå hur träd kan implementeras med länkad struktur.
* förstå begreppen: *binära träd*, *höjden* av ett träd, *djupet* för en nod, *minimal höjd*.
* förstå hur binära träd kan implementeras med länkad struktur.
* förstå implementationen för rekursiva operationer på träd.
* känna till de olika sätten att *traversera* träd; *preorder*, *inorder*, *postorder*.
* förklara resultatet av olika traverseringar av binära träd
* i ord beskriva *Huffmans algoritm* (används i samband med filkomprimering)

***Binära sökträd:***

* förstå hur data lagras i ett binärt sökträd och hur sökning, insättning och borttagning går till.
* känna till hur värstafallen beror på trädets höjd och att man måste göra någonting åt trädets form för att kunna garantera bättre värstafallskomplexitet än O(n).

Naiv implementation av binära träd.

***Balanserade binära sökträd:***

* förstå begreppen *AVL-träd* och *balans* samt varför man vill balansera binära sökträd.
* känna till att värstafallstider för operationerna på *balanserade binära sökträd* är O(log n).
* utföra *rotationer* på enkla exempel i stil med dem som finns på föreläsningsbilderna.
* känna till och kunna använda träd som Symbol tabell samt klasserna TreeSet och TreeMap som ingår i the Collections API.

***Hash Tabell:***

* förstå begreppen *hashtabell*, *hashfunktion*, *symboltabbel*
* förstå begreppen *sluten hashtabell*, *fyllnadsgrad*, *primär klustring*.
* veta hur problemet med *kollisioner* i slutna tabeller löses med *linjär* resp
* kunna implementera enkla hasfunktioner och hashtabeler.

***Prioriteskö och heapen :***

* känna till datatypen *prioritetskö* och dess vanligaste operationer.
* känna till och kunna jämföra olika sätt att implementera en prioritetskö.
* förstå begreppen *heap*, *partiellt ordnat träd*, *komplett träd*.
* förstå hur en heap kan implementeras med hjälp av en array
* förstå hur operationerna insert, findMin, deleteMin går till i en heap.
* känna till dessa metoders tidskomplexitet.
* förstå hur man bygger en heap på linjär tid från en osorterad lista.

***Graf:***

* förstå begreppen *graf*, *nod*, *båge*, *oriktad graf*, *riktad graf* (digraph)
* förstå hur en graf implementeras med en matris.
* förstå hur en graf implementeras med *närhetslistor*.
* redogöra för olika grafalgoritmer t ex
* *Bredden-först-traversering*
* *Dijkstras algoritm* för kortaste väg

***Test:***

* kunna implementera enkla testmetoder enligt Junit-test

***Implementation:***

* kunna implementera alla algoritmer och lösningar av den komplexiteteten som har förekommit på laborationerna