Programmering fortsättningskurs

Philip Larsson

2013–03–09

Innehåll

1 Träd **1**
- 1.1 Binära träd . . . 1
- 1.2 Strikt binärt träd . . . 1
- 1.3 Binärt sökträd . . . 1
- 1.4 Balanserade binära sökträd . . . 1

2 Traversering **2**

3 Interfacet Set **2**

4 Interfacet Comparator **3**

5 Interfacet Map **4**

6 Hashtabell **4**

7 Rekursiva metoder **6**

8 Skyddsnivåer **6**

9 Överskuggning och överlagring **6**

10 Exceptions **7**

11 Sortering **8**
- 11.1 Mergesort . . . 8
- 11.2 Quicksort . . . 9
- 11.3 Heapsort . . . 9
- 11.4 Insättningssortering (insertion sort) . . . 9
- 11.5 Urvalssortering (selection sort) . . . 9
- 11.6 Sammanfattning sortering . . . 10

12 Lista **10**

13 Kö **11**
- 13.1 Prioritetskö . . . 12

14 Stack **12**

15 Heap **13**

16 Interfaces **14**
16.1 Java Collections Framework 14

17 Klassen Object **15**

18 Algoritmers tidsåtgång **16**
18.1 Tidskomplexitet för några algoritmer 16

1 Träd

Träd är en icke linjär struktur.
Ett träd består av en huvudnod (roten) som har noll eller flera subträd.
En nod som saknar barn kallas löv.
En gren är en serie noder förbundna med varandra.

1.1 Binära träd

Ett binärt träd är ett träd där varje nod har högst två barn.

1.2 Strikt binärt träd

Ett strikt binärt träd är ett träd där varje nod har noll eller två barn.

1.3 Binärt sökträd

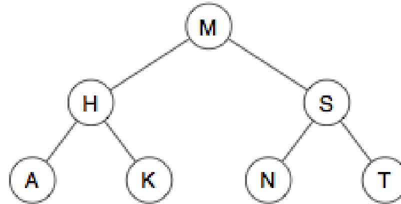
Ala värden som finns i vänster subträd är mindre än värdet som finns i föräldern. Alla värden som finns i höger subträd är större än värdet som finns i föräldern.
Dubbletter tillåts inte!

1.4 Balanserade binära sökträd

Ett binärt träd är balanserat om det för varje nod i trädet gäller att höjdskillnaden mellan dess båda subträd är högst ett.

2 Traversering

Att genomsöka alla noderna i ett träd kallas att traversera trädet. Detta kan göras genom post-, pre- eller inordertraversering.



- Preorder:
  Först roten, sedan vänster subträd i preorder, därefter höger subträd i preorder.
  M, H, A, K, S, N, T

- Inorder:
  först vänster subträd i inorder, sedan roten och därefter höger subträd i inorder (detta blir växande ordning i ett binärt sökträd).
  A, H, K, M, N, S, T

- Postorder:
  först vänster subträd i postorder, sedan höger subträd i postorder och därefter roten.
  A, K, H, N, T, S, M

3 Interfacet Set

En mängd (Set) får inte innehålla några dubbletter.

Interfacen `Collection` och `Set` har i Java samma operationer. Skillnaden är att i `Set` får inga dubbletter förekomma.
`Set` får innehålla null element, men bara ett null-element, pga dubblettförbudet.

Klasser som implementerar interfacet Set:

- TreeSet (balanserat träd)

- HashSet (använder hashtabell)

Några metoder som kan vara bra att känna till:

`boolean add(E obj)`	Lägger till obj om det inte redan finns
`boolean addAll(Collection<E> coll)`	Lägger till alla element om de inte redan finns, returnerar true om set:et är ändrat.
`boolean contains(Object obj)`	Returnerar true om set:et innehåller ett element som är identiskt(equal) till obj

4 Interfacet Comparator

```
public interface Comparator<T> {
   /**
   *Compares its two arguments for order.
   *Returns a negative integer, zero, or a positive
   *integer as the first argument is less than,
   *equal to, or greater than the second.
   */
   int compare(T e1, T e2);
}
```

Comparator är ett interface med en metod `compare` som ska användas för att jämföra två olika objekt av typen T. När man anropar sort skickar man med ett objekt av en klass som implementerar Comparator. Inuti sort används `compare` för jämförelser

En klass som implementerar interfacet `Comparable` implementerar en metod `compare` för jämförelse.

Comparator kan vara bra när man vill jämföra objekt av en klass på flera olika sätt.

- Ex: En klass Person. Vi vill både kunna sortera personer i alfabetisk ordning och efter personnummer.

```
public class Person {
   private String name;
   private String pNbr;
   ...
```

```
}
public class NameComparator implements Comparator<Person> {
   public int compare(Person p1, Person p2) {
      return p1.getName().compareTo(p2.getName());
   }
}
```

```
/* This tree will be sorted by name */
TreeSet<Person> nameTree = new TreeSet<Person>(new
    NameComparator());
```

5 Interfacet Map

Element är tvådelade - en nyckel(key) med tillhörande värde(value). Nyckeln är unik, men inte värdena.

Klasser som implementerar interfacet Map:

- TreeMap (implementerar interfacet SortedMap, dvs elementen kommer att vara sorterade)
- HashMap (använder en hashtabell)

Några metoder som kan vara bra att känna till:

`V get(Object key)`	Returnerar värdet associerat med key. Returnerar null om det inte finns.
`V put(K key, V value)`	Associerar key med value i denna mapen. Dvs lägger till i mapen.
`V remove(Object key)`	Tar bort objektet med nyckeln key i mapen.

6 Hashtabell

En hashtabell är en datastruktur där data sparas tillsammans med en nyckel. Positionen i strukturen beräknas med en hashfunktion. Ofta behöver man en datastruktur som kan hantera både insättningar och sökningar effektivt.

- Sluten hashtabell:
  En vektor används för att lagra elementen. Hanterar kollisioner med t.ex. linjär teknik eller kvadratisk teknik.

- Öppen hashtabell:
  En vektor av listor. Kolliderande objekt placeras i samma lista.

Exempel på på sätt att hantera kollisioner:

- Linjär teknik: första lediga plats efter.
  Problem med linjär teknik:
  - Ger upphov till primär klustring i tabellen.
  - Objekt vars hashkoder är nära ett ”vanligt” hashvärde kommer att drabbas av kollisioner och bygga ut klustret.
  - Stora kluster gör sökningen långsam.

- Kvadratisk teknik: undviker primär klustring (att alla värden samlas intill varandra i vektorn). Funkar inte nästa plats 1^2 hoppar den till 2^2 steg bort, funkar inte det går den 3^2 osv.
  Alltså: Primärt hashvärde $=$ $pHash$ $=$ värdet där den ”egentligen” borde läggas in.
  $pHash, pHash + 1^2, pHash + 2^2, pHash + 3^3, \ldots, pHash + i^2$

Man måste skugga `hashCode` och `equals` i den klass vars objekt ska fungera som en nyckel i en hashtabell.

I Javas klassbibliotek används öppna hashtabeller i klasserna `HashSet` och `HashMap`.

Fyllnadsgraden i en hashtabell är $\frac{\text{antal element}}{\text{antal tabellplatser}}$.

Om man skall skriva en for-each sats över en hashtabell:

```
Set<String,Object> map = new Hashmap<String,Object>();

for (Map.Entry<String, Object> entry : map.entrySet()) {
    String key = entry.getKey();
    Object value = entry.getValue();
    // ...
}
```

Observera att element i en hashtabell inte är sorterade. Ett elements plats i tabellen bestäms av dess hashvärde.

7 Rekursiva metoder

En rekursiv metod måste ha:

- En eller flera parametrar
- Ett eller flera basfall
- Ett eller flera rekursiva anrop. De rekursiva anropen måste leda till att ett basfall så småningom nås.

8 Skyddsnivåer

- `public`: Attribut och metoder som vi skriver `public` framför kan anropas och användas varifrån som helst.
- `Ingen markering`: Attribut och metoder som vi inte skriver något framför kan bara anropas och användas av andra klasser i samma katalog.
- `private`: Attribut och metoder som vi skriver `private` framför kan bara anropas och användas inifrån den egna klassen.
- `protected`: Klassen själv, andra klasser i samma paket, och this i subklasser har tillgång till dem.

9 Överskuggning och överlagring

- Överskuggning innebär att vi ersätter en gammal (ärvd) operation med en ny som har samma signatur - vi får alltså ingen ny operation.
- Överlagring innebär att vi lägger till en ny operation som har samma namn som en ärvd operation, men annan signatur. (t.ex olika antal parametrar eller olika typer på sina parametrar)

Vi kan även använda den överskuggade operationen när vi överskuggar med hjälp av `super`.

```
public void forward(int n){
  super.forward(n);
  count += n;
}
```

Vi skriver `super.forward(n)` för att få kompilatorn att förstå att det är operationen `forward(int)` i superklassen som skall anropas. Om vi hade skrivit `forward(int)` hade vi anropat oss själva - sådana anrop kallas rekursiva och är väldigt användbara i många sammanhang, men här hade det bara inneburit at operationen hade anropat sig själv tills minnet tar slut (och vi får `java.lang.StackOverflowError`).

10 Exceptions

Det finns två slag av Exceptions:

- Unchecked Exceptions Ð används då felet beror på programmeraren
  Ex: NullPointerException eller ArrayIndexOutOfBoundsException
- Checked Exceptions Ð används då felet inte beror på programmeraren
  Ex: FileNotFoundException om man försöker öppna en fil som inte finns

När man anropar en metod som genererar en checked exception måste man ta hand om det. Normalt fångar man det i en try-catch-sats:

```
Scanner scan = null;
try {
   // try to open file fileName
   scan = new Scanner(new File(fileName));
} catch (FileNotFoundException e) {
   System.err.println("Couldn't open file " + fileName);
   System.exit(1);
}
 ... use scan ...
```

Om exception inträffar, avbryts exekveringen av satserna i try-blocket och satserna i catch-blocket exekveras.

I satsen `catch(Exception e)` kan t.ex. följande metoder användas för att få mer information:

- `e.printStackTrace();` som skriver ut information om raden där felet inträffat och den/de metodanrop som lett till denna rad.
- `e.getMessage();` som returnerar en sträng med meddelande om felets art.

Generera egna exceptions:

```
throw new IllegalArgumentException("amount to deposit < 0");
```

Mönstret är:

```
throw new ExceptionClass();
throw new ExceptionClass(message);
```

Detta gör att exception-objekt skapas, exekvering av metoden avbryts. Javasystemet letar efter fångade catch-block.

Exempel:

```
/**
 * Deposits the specified amount.
 * pre: The specified amount is >= 0
 * post: The specified amount is added to balance
 * @param n The amount to deposit
 * @throws IllegalArgumentException if the specified amount is < 0
 */
public void deposit(int n) {
   if (n < 0) {
      throw new IllegalArgumentException("amount to deposit < 0");
   }
   balance = balance + n;
}
```

11 Sortering

Varför sortera?

- Göra sökning enklare.
- Förenkla vissa algoritmer.

11.1 Mergesort

En söndra och härda-algoritm.
Först sorteras vänstra halvan, sedan den högra. Sedan samsorteras de båda sorterade halvorna. Mergesort har tidskomplexiteten $O(n \log n)$ oavsett indata.

11.2 Quicksort

Välj ut ett element (förslagsvis i mitten), detta element kallas för pivotelementet.
Flytta element som är mindre än pivotelementet till vänster om det och flytta element som är större till höger. Quicksort har tidskomplexiteten $O(n \log n)$, men $O(n^2)$ i värsta fall.

11.3 Heapsort

Heapsort har tidskomplexiteten $O(n \log n)$, är vektorn sorterad tar det n för att sätta in och $\log n$ för att ta ut, alltså ändå $O(n \log n)$.

11.4 Insättningssortering (insertion sort)

Element på plats k i vektorn sätts in på rätt plats bland de redan sorterade elementen på plats $0 \to (k-1)$.
Tidskomplexiteten är $O(n^2)$
Insättningssortering är bra om vektorn nästan är sorterad från början.

11.5 Urvalssortering (selection sort)

Den kanske enklaste metoden att sortera är att börja med att leta upp det minsta värdet i vektorn och sätta in det först. Därefter letar vi upp det näst minsta elementet (som är det minsta av de återstående elementen) och sätter in det näst först, och så fortsätter vi att leta upp och sätta in det tredje minsta, fjärde minsta, etc, tills alla element är i rätt ordning.

```
public static void selectionSort(int[] a, int n) {
  for (int i = 0; i < n; i++) {
    int smallest = i;
    for (int k = i+1; k < n; k++) {
      if (a[k] < a[smallest]) {
        smallest = k;
      }
    }
    int tmp = a[i];
    a[i] = a[smallest];
    a[smallest] = tmp;
  }
}
```

För att få ett mått på hur effektiv denna metod är kan vi räkna antalet jämförelser som behövs för att sortera en vektor med n tal. I första varvet i den yttre for-loopen gör vi $n-1$ jämförelser (de görs inuti den inre for-loopen), i det andra varvet i den yttre for-loopen gör vi $n-2$ jämförelser, etc. Totalt får vi:

$$T(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2} = O(n^2)$$

jämförelser. dvs om vi dubblar antalet element i vektorn så krävs det ungefär fyra gånger fler jämförelser för att sortera elementen.

11.6 Sammanfattning sortering

Urvalssortering $O(n^2)$	Långsam för stora n. Efter k pass är de k minsta sorterade.
Insättningssortering $O(n^2)$	Bra för nästan sorterad indata (linjär då).
Heapsort $O(n \log n)$	Kan utformas så att inget extra minnesutrymme krävs. I praktiken långsammare än Quicksort. Efter k pass är de k största elementen sorterade.
Mergesort $O(n \log n)$	Kräver extra minnesutrymme. I praktiken långsammare än Quicksort. Kan utformas iterativa och användas för att sortera element som finns p
Quicksort $O(n \log n)$	Men $O(n^2)$ i värsta fall. Inget extra minnesutrymme för temporär vektor krävs. Bäst av de nämnda i praktiken om man väljer pivot och utför partionering förnuftigt.

12 Lista

Implementering av listor:

- Vektor, föregångare och efterföljare finns ”naturligt”.
- Länkad datastruktur. Består av noder som har information om efterföljande element(eventuellt också föregående = dubbellänkad lista)

En lista är en ordnad följd av element.

- Det finns en före-efter-relation mellan elementen

- Begrepp som ”första element i listan”, ”efterföljaren till visst element i listan” är meningsfulla. Det finns alltså ett positionsbegrepp.

Vi kan skapa en Lista med interfacet list:

```
List<String> names;
```

och på så vis vänta med att välja vilken typ av lista vi vill ha. Två exempel på olika klasser som implementerar List.

- `ArrayList`, som är särskilt bra på att snabbt hämta värden med get-operationen. Insättningar bör man bara göra sist i listan, då insättningar i andra positioner orsaker flyttningar av element.

- `LinkedList`, som är särskilt bra på att sätta in och ta bort element i början av listan.

Vi skapar en lista såhär:

```
List<String> names = new ArrayList<String>();
```

Observera:

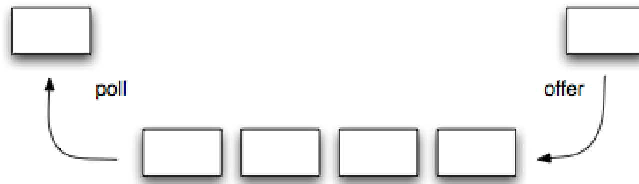
```
// works but considered ugly:
ArrayList<String> names = new ArrayList<String>();

// better! :
List<String> names = new ArrayList<String>();
```

13 Kö

En kö kallas även FIFO-lista. First In First Out.
Insättningar göras alltid sist i följden, och bortagning avser alltid första elementet.
Analogi: En (vanlig) kö när man handlar mat på ICA...
Några metoder som kan vara bra att känna till:

`boolean offer(E item)`	Sätter in item sist i kön.
`E poll()`	Tar bort och returnerar första elementet i kön
`peek`	Returnerar första elementet i kön (utan att ta bort det).



13.1 Prioritetskö

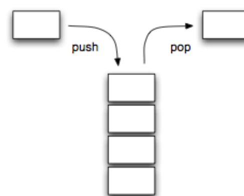
En prioritetskö är en samling element där varje element har en prioritet (som används för att jämföra elementen med). Elementen plockas ut i prioritetsordning till skillnad mot en vanlig kö där elementen plockas ut i den ordning de satts in i kön. De operationer man ska kunna göra på en prioritetskö är

- sätta in element.
- ta reda på det högst prioriterade elementet (minsta elementet).
- ta bort det högst prioriterade elementet (minsta elementet).

Det finns inget speciellt interface för prioritetsköer. Man använder interfacet `Queue<E>`.

14 Stack

En stack kallas även LIFO-lista. Last In First Out.
Operationer sker på toppen av stacken, dvs. nya element läggs till längst upp, och element tas bort längst upp.
Analogi: En tallriks dispenser, en hög med papper.

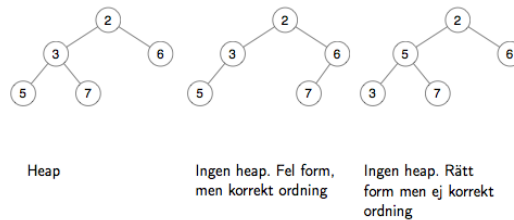


Några metoder som kan vara bra att känna till:

`void push(E obj)`	Lägger obj överst på stacken.
`E pop()`	Tar bort och returnerar översta elementet från stacken.
`E peek()`	Returnerar första elementet i kön (utan att ta bort det).
`boolean isEmpty()`	Undersöker om stacken är tom.

15 Heap

Heap - ett komplett binärt träd där varje nod innehåller ett element som är $\leq$ barnens element.
Alltså: Trädets grenar är så lika långa som möjligt. I fall det inte är möjligt så fylls den nedersta nivån på från vänster (vänsterbalanserat). För varje nod



gäller det att roten innehåller det minsta elementet.

En heap kan med fördel lagras i en vektor.

- Roten finns på plats 0.
- Barnen till noden på plats i finns på platserna $2i + 1$ och $2i + 2$ i vektorn. Noden på plats i har alltså sin förälder på plats $(i - 1)/2$.

Offer (insättning i heap) görs genom att:

- Nya element placeras på första lediga plats i vektorn.
  Detta ger rätt form på trädet.
- Sedan byten uppåt tills rätt ordning.

16 Interfaces

Ett interface(svenska: gränssnitt) är som ett skal med metoder som den som implementerar måste använda. En klass kan implementera flera interface. Alla metoder är även implicit publika.

```
public interface InterfaceName{
   /** Returns size */
   int getSize();
}
```

```
public class ClassName implements InterfaceName{
   // attributs, methods, constructor as usual, but all methods in
       InterfaceName must be included
}
```

Om det skiljer mycket mellan metoderna i klasserna så använd interface istället för en superklass.

16.1 Java Collections Framework

- Är en hierarki av interface, abstrakta klasser och konkreta klasser för samlingar av element.
- Finns i paketet `java.util`.

JFC - interface hieraki

Collection	en samling av element, där dubbletter tillåts
Queue	en samling av element som utgör en kö
List	en samling element där dubbletter tillåts och där positionerna är möjlig (första, sista, element på plats i, ...)
Set	en samling element där dubbletter är förbjudna
SortedSet	som Set men med kravet att elementen går att jämföra
Map	en samling element där varje element har en nyckel och ett värde
SortedMap	som Map men med krav att nycklarna går att jämföra

Interface	Implementering
Queue	LinkedList, PriorityQueue
List	ArrayList, LinkedList
Set	HashSet
SortedSet	TreeSet
Map	HashMap
SortedMap	TreeMap

17 Klassen Object

Det finns en klass `Object`, som är "alla klassers moder", om vi inte uttryckligen ärver någon annan klass så ärver vi automatiskt klassen `Object`. Klassen `Object` innehåller en del metoder som kan vara bra att känna till:

- toString

  ```
  public String toString()
  ```

  ger en sträng som beskriver objektet. Denna operation anropas automatiskt om vi skulle försöka skriva ut ett objekt, och implementationen i klassen `Object` skriver ut klassnamnet och ett heltal (objektets hash-kod)

- Metoden `equals` används för att jämföra om två objekt är lika.

  ```
  public boolean equals(Object obj);
  ```

  Metoden returnerar true om och endast om de jämföra objekten är identiska. Om man istället vill att innehållet inuti objekten ska jämföras måste man skugga `equals`.

  ```
  public boolean equals(Object obj){
     if (obj instanceof Person){
        return idNbr == ((Person) obj).idNbr;
     }else{
        return false;
     }
  }
  ```

  – Observera att parametern till `equals` måste vara av typen `Object`, annars blir det inte skuggning. Därför måste också typomvandlig

till Person ske när man ska använda obj:s idNbr.

  – Uttrycket `obj instanceof Person` returnerar true om obj:s typ är av Person eller någon subklass till Person.
  – Uttrycket `obj instanceof Person` returnerar false om obj har värdet null.

- Förutom operationerna ovan innehåller även `Object` en operation

  `public int hashCode()`

  som ger en så kallad hash-kod, dvs ett slags numeriskt fingeravtryck för ett objekt. Detta innebär att alla objekt har en hashkod.
  Exempel:

```
public int hashCode() {
   return isbn.hashCode();
}
```

18 Algoritmers tidsåtgång

Storleksordning för funktioner:

$f(n)$	namn
1	konstant
$\log n$	logaritmisk
n	linjär
$n \log n$	log-linjär
n^2	kvadratisk
n^3	kubisk
2^n	exponentiell

18.1 Tidskomplexitet för några algoritmer

- Sortera n tal med urvalssortering, bubbelsortering eller insättningssortering: $O(n^2)$
- Sortera med Mergesort och Quicksort: $O(n \log n)$
- Söka bland n osorterade element: $O(n)$

- Söka med binärsökning bland n sorterade element: $O(\log n)$
- Sätta in element först i en lista med n element: $O(1)$.