# 1 Samlingar

## 1.1 Frekvenstabell

### Kommentarer

Se javakoden.

- En **Integer** är icke-muterbar (precis som **String**, **Float**, **Boolean** et.c.). Icke-muterbarhet har många fördelar, men en nackdel är att ett helt nytt objekt måste skapas när ett nytt värde ska lagras, något som kan dra ned programmets prestanda. Ett frekvensvärde i uppgift 2 lagras som en Integer och när ett sådant värde ökas, är det alltså istället ett nytt objekt med det högre värdet som skapas.
- Vid testkörning av programmet `w5/ex1/frequencytable/sol/HistogramTest` är skillnaden inte särskilt stor[1] (givetvis lite beroende på mängden unika nycklar). Provkör gärna och se vad du får för skillnad. Typiskt hos mig är att **Histogram1** kör på 1.6 sekunder och **Histogram2** kör på 2.4 sekunder per test.

## 1.2 Iteratorrättning

### Kommentarer

Se javakoden.

## 1.3 En exklusiv union

### Kommentarer

Se javakoden.

## 1.4 Under huven

### Kommentarer

- Övningen vill visa på vikten av valet av datastruktur, trots att de ofta erbjuder samma funktionalitet genom sina gränssnitt.
- En länkad lista (**x**) är som synes bäst för tillägg av innehåll (add) medan den är betydligt sämre än de andra två andra på att söka upp (**contains**) ett lagrat element. Detta beror på att allting i en länkad lista läggs "på rad", med bara direkt tillgång till första (och ev. sista) elementet i denna rad. Att lägga till ett element är därför lätt, men att hitta det kräver att programmet börjar i ena änden av raden och går igenom ett element i taget, i värsta fall kan det därför behöva stega sig igenom alla.

[1] ... på min maskin, Pentium i5, GNU/Linux 3.0.0-13-generic x86_64, OpenJDK 64-bit Server VM 1.6.0_23

Figur 1: Antal steg för olika datastrukturer som en funktion av antal element de innehåller ($n$), något förenklat.

| Struktur | Mean add | Worst add | Mean contains | Worst contains |
|---|---|---|---|---|
| Länkad lista | 1 | 1 | $n/2$ | $n$ |
| Balanserat träd | $\sim \log_2(n)$ | $\log_2(n)$ | $log_2(n) - 1$ | $log_2(n)$ |
| Hashtabell | $\sim 1$ | $n - 1$ | $\sim 1$ | $n$ |

- Ett träd (**y**) visar sig ungefär lika bra vid tillägg och sökningar. Trädet lagrar sina element i form av, ja just det, ett träd :) I förhållande till den länkade listan behöver den (förmodligen) inte gå lika lång väg, den börjar vid trädets rot och letar sig steg för steg ut till rätt gren. Detta behöver den göra vid både tillägg och sökning, och därför tar dessa ungefär lika lång tid. Vid tillägg behöver den ev. också ändra på trädets struktur för att det ska fortsätta vara just ett träd (i värsta fall skulle ett träd annars kunna bli en länkad lista, d.v.s. alla grenar ligger på en rad efter varandra). Denna omstrukturering kallas för *balansering*[2]. För att kunna avgöra var elementen ska lagras måste de kunna jämföras med varandra (d.v.s. implementera **Comparable<T>** i Java).

- En hashtabell (**z**) ter sig i det här testet som det bästa alternativet, även om länkade listan slår den för tillägg. Dess snabbhet beror på att den internt använder sig av en array, men den kan ändå inte garantera att alla element lagras på olika positioner. Flera element som lagras på samma position läggs därför i en länkad lista. Om (teoretiskt) alla element lagrades på en position skulle alltså en hashtabell degraderas till en enda länkad lista och heller inte prestera bättre. Utmaningen ligger därför i att försöka placera elementen så jämt över arrayens positioner som möjligt. En position i en array nås som vanligt genom dess index, alltså ett positivt heltal. För att bestämma indexet för ett element utförs ofta en beräkning som involverar elementets inneboende tillstånd (känns det bekant? i Java används metoden **hashCode()** som ju alla objekt innehåller).

- Varför används då träd överhuvudtaget om hashtabellen ändå presterar bättre? Ett träd har ytterligare en egenskap, det håller elementen sorterade.

  *Dessutom är värsta falls-beteendet för hashtabeller* ***linjär****, d.v.s. det kan bli lika dyrt att söka som i en länkad lista.* Skillnaden mellan förväntad genomsnittstid kontra värsta falls-tid kan ses i figur 1.4 där $n$ är antalet befintliga element i strukturen. Vi antar också att de träd vi använder garanterat är balanserade, vilket är fallet med t.ex. **TreeSet** och **TreeMap**. Genomsnittsfallet för hashtabeller är något förenklat men är inte en funktion av antalet element utan av *fyllnadsgraden* hos tabellen.

[2] För mer information, se kurser i datastrukturer och algoritmer.

# 2 Generics

## 2.1 Typsäkerhet

### Lösning

Se programkod.

### Kommentarer

Programmet krashar då en **Bicycle** inte har **Car** som superklass. Då programmet kommer till det element som är en **Bicycle** i `vehicles` blir det `ClassCastException`; en cykel släpper inte ut koldioxid (annat än möjligen genom cyklisten själv).

Då vi använder ”råa typer”, d.v.s. vi utelämnar den möjliga typparametern finns det ingen möjlighet för kompilatorn att lista ut att vi vid ett senare tillfälle förväntar oss att det endast finns objekt av en given typ i samlingen.

Lösningen blir alltså att inskränka samlingen genom att istället deklarera variabeln `Collection<Car> vehicles = new HashSet<Car>;`. Nu kompilerar programmet inte om man försöker lägga till t.ex. en cykel.

# 3 Singleton-mönstret

### Lösning

Se programkod.

### Kommentarer

- Klassen är inte en singleton eftersom flera instanser kan skapas av klassen.
- För att göra klassen till en singleton skulle följande behöva förändras:
  - Genom att istället göra en `enum` av klassen blir det omöjligt att skapa instanser av den. Notera semikolonet som indikerar att listan av enum-element är slut. Detta är alltså en enum utan enum-element!
  - För ett mer utvecklat resonemang, se gärna Item 3 i Joshua Bloch's "Effective Java".
- Det finns dock inget att tjäna på att göra klassen till en singleton eftersom det inte finns något tillstånd som behöver initieras vid ett bestämt tillfälle.[3] Men eftersom klassen bara innehåller statiska metoder bör den inte gå att instansiera överhuvudtaget. Att göra den till en **enum** hindrar oss också från att kunna skapa instanser genom deserialisering[4]. Att göra en vanlig klass med privat konstruktor skyddar oss inte från detta.
- Detta har ingenting med singleton egenskapen att göra men...

  Då denna veckas övningar delvis handlar om generics, studera generaliseringen som gjorts i **GameUtils**. Det är onödigt oflexibelt att bara kunna skapa och fylla fält av fält av **GameTile**. Genom några små förändringar har vi nu åstadkommit metoder som kan fylla och skapa fält av fält av vilken referenstyp som helst.

  Om vi bytt ut *typparametern* **T** mot *klassnamnet* **Object** hade vi kunna blanda element helt på måfå. Som metoderna ser ut nu kan vi bara fylla dem med element av just typen **T**.

[3] Det finns mycket kritik mot designmönstret Singleton, till stor del p.g.a. att en singleton innebär införande av ett globalt tillstånd. För att läsa mer, gör en internetsökning, det har diskuterats ordentligt...

[4] Se `http://en.wikipedia.org/wiki/Serialization#Java`

# 4 En udda dekoratör och iterator

### Kommentarer

- Klassen hade kunnat användas som iterator till en ”svagt icke-muterbar” samling, d.v.s. åtkomst till samlingens ingående objekt tillåts (vilka kan vara muterbara), men samlingen själv går inte att förändra.
- Java har t.ex. till skillnad från C++[5] en hårt specificerad design av sin iterator. Det hade varit naturligare om denna delats upp på flera klasser, där den ”minsta klassen” bara erbjudit metoderna ”nästa” och ”harNästa”. Större varianter skulle då inkludera fler operationer. Som övningen visar, tvingas vi i Java att erbjuda möjligheten att ta bort element med iteratorn (såvida inte **remove()** ”fulhackas” bort genom exception-kastning). Dessutom går det bara att stega framåt, inte bakåt.
- Eftervillkoren för de två metoderna skiljer sig åt såtillvida att `processIterator(ForwardInputIterator<T>)` omöjligen kan mutera den underliggande strukturen; `remove` är ju borta. Det tråkiga med denna ”lösning” är att vi får ett körningsfel om vi försöker anropa `remove` i metoden `processIterator(Iterator<T>)` med en **ForwardInputIterator**.

[5] http://www.cplusplus.com/reference/std/iterator/