Föreläsning 2
Datastrukturer (DAT037)

Fredrik Lindblad[1]

2016-11-02

[1]Slides skapade av Nils Anders Danielsson har använts som utgångspunkt. Se
`http://www.cse.chalmers.se/edu/year/2015/course/DAT037`

Tidskomplexitet

Hur analyserar man tidskomplexitet?

- *Mäta*.
  Nackdelar: Kan vara tidskrävande,
  kanske inget bra stöd för design. Nödvändigt i vissa fall för att se vad som är bäst i praktiken (för t. ex. små indata)
- *Räkna instruktioner*.
  Nackdelar: Komplicerat.
- *Förenklad modell*.
  Nackdelar: Inte tillämpligt i alla lägen.

Uniform kostnadsmodell

- Godtyckligt stora tal.
- Oändligt minne.

Uniform kostnadsmodell

- Godtyckligt stora tal.
- Oändligt minne.

- Inte realistisk.
- Fungerar ganska bra om man är försiktig.
- Används ofta i kursen.

Logaritmisk kostnadsmodell

- Tid beräknas i termer av indatas storlek, räknat i antal bitar.
- Oändligt minne.
- Lite mer realistisk, lite krångligare.

Värsta-, bästafallskomplexitet

Vad händer med exekveringstiden om en dublett hittas?

```
class IntMultiSet {
 int[] a;
 ...
 boolean hasDuplicate() {
  for (int i = 0; i < a.length; i++) {
   for (int j = i + 1; j < a.length; j++) {
    if (a[i] == a[j]) return true;
   }
  }
  return false;
 }
}
```

Exekveringstiden beror inte alltid enbart på indatans storlek, utan kan bero av dess beskaffenhet i övrigt. Därför talar man om värsta falls-, bästa falls- (och medel-)komplexitet. För de två exemplen räknade vi med att det inte fanns dubletter så att slingorna skulle exekveras fullt ut. Det var alltså värsta fallskomplexiteten. I bästa fall hittas en dublett med en gång, d.v.s. bästa fallskomplexiteten är $O(1)$. Vanligast är att prata om värsta fallskomplexiteten eftersom man använder $O()$ och är intresserad av hur lång tid det tar som längst.

Exempel – vad är tidskomplexiteten (som funktion av längden på x)?

```
static void filter(double[] x) {
 for (int i = 0; i <= x.length - 5; i++) {
  double s = 0;
  for (int j = i; j < i + 5; j++) {
   s += x[j];
  }
  x[i] = s / 5;
 }
}
```

$n = \texttt{x.length}$

Iteration av inre slingan: $O(1)$
Inre slingan (upprepas 5 ggr): $O(5 * 1) = O(1)$
Resten av en iteration av yttreslingan: $O(1)$
En iteration av yttre slingan totalt:
$O(1 + 1) = O(1)$
Yttre slingan: $O((n - 4) \cdot 1) = O(n)$
$T(n) = O(n)$

Ett exempel till. Tidskomplexitet?

```
static void f(int[] x) {
 for (int i = x.length; i > 0; i /= 2) {
  x[i - 1] = 0;
 }
}
```

Antag att $n = 2^k$. Då gäller att antal iterationer är:

$$I(2^k) = 1 + k$$

Räknar man noga på det kan man visa att för godtyckligt n gäller

$$I(n) = 1 + \lfloor {}_2\log(n) \rfloor \qquad (n > 0)$$

Tidskomplexitet är allstå:

$$T(n) = O(\log(n))$$

Dynamisk array

Åter till vår första variant av `IntMultiSet`. En metod add lägger till ett nytt tal.

```
class IntMultiSet {
 int[] a;
 ...
 void add(int x) {
  int[] newa = new int[a.length + 1];
  for (int i = 0; i < a.length; i++) {
   newa[i] = a[i];
  }
  newa[a.length] = x;
  a = newa;
 }
 ...
 boolean hasDuplicate() { ... }
}
```

Varje insättningsoperation tar $O(n)$. Går att göra bättre.

```
class IntMultiSet {
 int[] a;
 int size;  // #used elements in a
 ...
 void add(int x) {
  if (size == a.length) {
   int[] newa = new int[size + 10];
   for (int i = 0; i < size; i++) {
    newa[i] = a[i];
   }
   a = newa;
  }
  a[size] = x;
  size++;
 }
 ...
}
```

9 av 10 insättningar går på $O(1)$ men var tionde på $O(n)$. En sekvens av insättningar går 10 ggr snabbare i praktiken, men ändå $O(n)$ i genomsnitt. Använd istället en dynamisk array som växer multiplikativt.

```
class IntMultiSet {
 int[] a;
 int size;  // #used elements in a
 ...
 void add(int x) {
  if (size == a.length) {
   int[] newa = new int[2 * size];
   for (int i = 0; i < size; i++) {
    newa[i] = a[i];
   }
   a = newa;
  }
  a[size] = x;
  size++;
 }
 ...
}
```

Amortized complexity / bokföringsmetoden

För en sekvens av operationer är amorterad kompexitet den tid per operation som gått åt. För billiga exekveringar betalar man lite extra ”tids-mynt” som man kan använda vid senare dyra exekveringar. För den dynamiska arrayen som dubblar storlek kan man betala två extra mynt. Ett för att täcka kopieringen av elementet själv vid nästa dubblering och ett för att täcka kopieringen av ett av elementet i gamla halvan. Detta gör att varje element har ett mynt vid nästa dubblering så hela kopieringen kan betalas med sparade mynt. Den amorterade komplexiteten är alltså $O(1)$.

Detta fungerar inte med varianten som ökar storleken additivt. Där skulle vi behöva spara $n/10$ mynt per operation för att täcka nästa kopiering. Alltså amorterad (och värsta falls-)komplexitet $O(n)$.

Generics

Att återanvända kod är en princip som genomsyrar programmerandet och design av programspråk. Vi har sub-rutiner och klass-arv etc. som stöder detta.
En sak som också främjar detta är generics/polymorfism.
När data inte behöver vara en av specifik typ kan vi i java säga att den kan vara vad som helst.

Här är ett exempel på en generisk metod:

```
static <E> void reverse(E[] arr) {
 E tmp;
 for (int i = 0; i < arr.length / 2; i++) {
  tmp = arr[i];
  arr[i] = arr[arr.length-1-i];
  arr[arr.length-1-i] = tmp;
 }
}
```

Förutom generiska metoder kan man även definiera generiska interface och klasser.
Generics kommer vi använda mycket eftersom datastrukturer handlar om att organisera många data-värden. Man vill förstås implementera datastrukturer för alla möjliga typer av data på en gång.
Ofta kan inte de generiska typerna vara helt godtyckliga utan måste ha någon egenskap som används av metoden/klassen.

Här är ett exempel där jämförelse måste vara definierad för den okända typen:

```
static <E extends Comparable<? super E>>
       E findMin(E[] arr) {
 if (arr.length == 0) return null;
 E min = arr[0];
 for (int i = 1; i < arr.length; i++) {
  if (arr[i].compareTo(min) < 0) min = arr[i];
 }
 return min;
}
```

Listor, stackar, köer

Problem/algoritm

Problem: sortera en lista.

Algoritmer:

- Insertion sort.
- Mergesort.
- Quicksort.
- ...

Abstrakt datatyp/datastruktur

Abstrakt datatyp (matematisk abstraktion): lista.

Datastrukturer (implementation):

- Array (dynamisk?).
- Enkellänkad lista.
- ...

Listor, stackar och köer: ADT-er

ADT	Operationer (exkl konstruerare)
Stack	push, pop
Kö	enqueue, dequeue
Lista	add(x), add(x, i), remove(x), remove(i), get(i), set(i, x), contains(x), size, iterator
Iterator	hasNext, next, remove

(Inte exakta definitioner.)

Listor, stackar och köer: datastrukturer

ADT	Implementationer
Lista	dynamisk array, enkellänkad lista, dubbellänkad lista
Stack	lista
Kö	lista, cirkulär array

ADT-er

- Kan använda en datastruktur för en viss ADT för att implementera en annan ADT.

Collections i Java

- Java Collections Framework.
- ADT $\sim$ gränssnitt, datastruktur $\sim$ klass.

Stackar och köer: tillämpningar

Stackar

- Implementera rekursion.
- Evaluera postfix-uttryck.
- ...

Köer

- Skrivarköer.
- Bredden först-sökning (grafalgoritm).
- ...

Länkade listor

Länkade listor

Många varianter:

- Objekt med pekare till första noden, kanske storlek.
- Pekare till sista noden?
- Enkellänkad, dubbellänkad?
- Vaktposter (sentinels)? Först/sist/både och?

Sammanfattning

- Amorterad tidskomplexitet.
- Generics
- ADT-er/datastrukturer.
- Listor, stackar, köer.