Föreläsning
Datastrukturer (DAT036)

Nils Anders Danielsson

2013-12-02

Tidigare

Sortering:

- Insättningssortering.
- Mergesort.
- Heapsort.

Idag

Sortering:

- Undre gräns.
- Radixsortering.
- Heapsort.
- Splaysort.
- Quicksort.

Idag

Antagande: Jämförelser tar konstant tid.

Undre gräns

Undre gräns

Sortering baserad enbart på
binära jämförelser kräver,
i värsta fallet,
$\Omega(n \log n)$ jämförelser.

Beslutsträd

- Träd som dokumenterar vilka jämförelser en sorteringsalgoritm utför då den sorterar listor av en viss längd n.
- Låt oss anta att alla element är olika.
- Interna noder: Jämförelser. ("Är $a[i] < a[j]$?")
- Ett barn per resultat ($a[i] < a[j]$, $a[i] > a[j]$).
- Löv: Sorterade listor ($a[3] < a[1] < a[5] < ...$).
- Jämförelsebaserade sorteringsalgoritmer: Löv får endast förekomma när jämförelserna ovanför i trädet ger tillräcklig information för att kunna sortera listan.

Ge en skarp undre gräns för antalet löv i ett beslutsträd.

- $n\lceil \log_2 n \rceil$.
- $n(n+1)/2$.
- $n!$.
- 2^n.

Undre gräns

- Djupet av ett visst löv $=$
  antalet jämförelser för att sortera viss lista.
- Trädets höjd $=$ värsta fallet.
- Ska visa att höjden är $\Omega(n \log n)$.

Undre gräns

- Ett binärt träd med höjd h har $\leq 2^h$ löv.
- Ett icketomt binärt träd med ℓ löv har höjd $\geq \log_2 \ell$.
- Ett beslutsträd för sortering av listor med distinkta element har $\geq n!$ löv (minst ett per möjlig permutation av listan).
- Beslutsträdet har alltså höjd $\geq \log_2(n!\,) = \Omega(n \log n)$.

Radixsortering

Hinksortering, bucket sort

- Sortering av lista med max N distinkta element.
- Allokera N hinkar.
- Stoppa in varje element i rätt hink.
- Gå igenom hinkarna och bygg ny lista.
- Stabil om implementerad på rätt sätt.

Hinksortering, bucket sort

Nedan är `bucket` en funktion som tar element till heltal $\in \{ 0, \ldots, \mathtt{N} - 1 \}$.

```
bucket-sort(N, bucket, xs):
  buckets = array of size N containing empty lists

  for x in xs do
    buckets[bucket(x)].add-last(x)

  ys = new empty list

  for b in 0, …, N - 1 do
    for y in buckets[b] do
      ys.add-last(y)

  return ys
```

Vad är tidskomplexiteten för hinksortering? Anta att `bucket` tar konstant tid.

- $\Theta(Nn)$.
- $\Theta(n \log n)$.
- $\Theta((N+n)\log(N+n))$.
- $\Theta(N+n)$.
- $\Theta(Nn \log Nn)$.

Radixsortering

- Hinksortering är inte praktiskt om n är "mycket" mindre än N.
- Alternativ: Radixsortering, lexikografisk sortering av nycklar med d delnycklar.
- Exempel: Tal bestående av d siffror.

Radixsortering

Least significant digit (LSD) radix sort
för tal med d siffor:

- Sortera talen stabilt med avseende på den minst signifikanta siffran.
- Sortera talen stabilt med avseende på den näst minst signifikanta siffran.
- Och så vidare...

Radixsortering

- Stabil.
- Om hinksortering med $O(1)$ `bucket` används:
  $\Theta(d(N+n))$,
  där N är talbasen (decimalt: 10, binärt: 2).
- Notera att man kan välja N själv.
  Exempel: 32-bitars tal kan delas upp i
  fyra 8-bitsdelar ($d = 4$, $N = 256$),
  eller två 16-bitsdelar ($d = 2$, $N = 65536$).

Heapsort

Prioritetskösortering

- Enkel sorteringsalgoritm: Sätt in varje element, kör `delete-min` tills kön är tom.
- $O(n \log n)$ om `insert` och `delete-min` har tidskomplexiteten $O(\log n)$.

Heapsort

- Variant av den enkla prioritetskösorteringsalgoritmen.
- *In-place*: Kräver $O(1)$ extra minne.

Heapsort

- Bygg binär maxheap in-place med `build-heap` (med roten på position 0).
- Kör `delete-max` upprepade gånger.
  Lägg största elementet sist i arrayen,
  nästa element näst sist, och så vidare.

Är heapsort stabil?

- Ja.
- Nej.

Heapsort

- Notera att algoritmen inte är in-place om den implementeras med rekursion (i avsaknad av ”tail-call optimisation” e d): rekursion använder minne (stack).

Splaysort

Splaysort

Variant av den enkla prioritetskösorteringsalgoritmen:

- Sätt in alla element i ett splayträd ($O(n \log n)$).
- Gå igenom trädet i inordning ($\Theta(n)$).

Splaysort

- Dubbletter måste tillåtas.
- Stabil om dubbletter hanteras korrekt.

Vad är tidskomplexiteten för att splaysortera en lista (med distinkta heltal) som är...

...sorterad?

- $\Theta(n)$.
- $\Theta(n \log n)$.

...omvänt sorterad?

- $\Theta(n)$.
- $\Theta(n \log n)$.

Adaptiva sorteringsalgoritmer

- Adaptiva sorteringsalgoritmer:
  $\Theta(n)$ för större eller mindre klasser av
  mer eller mindre sorterade listor.
- Insättningssortering:
  $\Theta(n)$ för listor med $O(n)$ *inversioner*
  (en inversion per par $i < j$ med $\texttt{xs}[i] > \texttt{xs}[j]$).
  $\Theta(n^2)$ för omvänt sorterade listor.
- Splaysort: Verkar vara snabb för
  många sorters "ganska sorterade" listor.

Quicksort

Quicksort

Idé (ej implementation):

```
quicksort :: Multiset a -> List a
quicksort ∅   = []
quicksort {x} = [x]
quicksort S   = quicksort S₁ ++ [p] ++ quicksort S₂
  where
  p ∈ S
  S₁ ∪ S₂ = S \ p
  ∀ x ∈ S₁. x ≤ p
  ∀ x ∈ S₂. x ≥ p
```

Quicksort

- Sorterar array.
- Effektiv om implementerad på bra sätt (se boken för några detaljer).
- Tight inre loop, partitionerar array in-place.

Vad är värstafallstidskomplexiteten för quicksort om `p` alltid är arraysegmentets första element? Anta att partitionering tar linjär tid.

- $\Theta(n)$.
- $\Theta(n \log n)$.
- $\Theta(n^2)$.
- $\Theta(n^2 \log n)$.
- $\Theta(n^3)$.

Partitioneringsstrategi

- Viktigt välja pivotelementet $\mathtt{p}$ på ett bra sätt, $\mathtt{S}_1$ och $\mathtt{S}_2$ ska helst vara ungefär lika stora.
- Rekommenderade val:
  - Slumpmässigt element.
  - Medianen av tre element:
    första, sista och ett av de mittersta.
- Bör även ha bra strategi för element lika med $\mathtt{p}$.

Quicksort

Tidskomplexitet:

- I värsta fallet (bara dåliga val av `p`): $O(n^2)$.
- Medelkomplexitet: $O(n \log n)$.

Många implementationer ej stabila.

Animationer

http://www.sorting-algorithms.com/

Sortering i Haskell

Data.List.sort:

- En variant av mergesort som först hittar växande/strikt avtagande segment:
  $[1, 2, 3, 4, 3, 2, 1, 1, 1, 2, 3] \rightarrow$
  $[[1, 2, 3, 4], [1, 2, 3], [1, 1, 2, 3], []]$
- Stabil, $O(n \log n)$, adaptiv.

Sortering i Java

JDK 7, arrayer med primitiva typer
(med reservation för fel):

- För långa arrayer med `byte` (≥ 31 element), `short`, `char` (≥ 3202):
  Räknesortering (counting sort), en variant av hinksortering med räknare istället för hinkar.
- För korta `byte`-arrayer: Insättningssortering.
- För långa arrayer (≥ 287) med få (≤ 66) segment ($\leq/\geq/$korta $=$): Variant av mergesort.
- Annars: Quicksort med två pivotelement.
- Quicksort använder variant av insättnings-sortering för korta arrayer (≤ 46).

Sortering i Java

JDK 7, arrayer med primitiva typer
(med reservation för fel):

- Adaptiv.
- `byte`, `short`, `char`: $O(n)$.
- Andra primitiva typer: $O(n^2)$ i värsta fallet, medelkomplexitet gissningsvis $O(n \log n)$.
- Stabilitet ointressant för primitiva typer.

Sortering i Java

JDK 7, arrayer med objekt:

- Timsort: Stabil, $O(n \log n)$, adaptiv.
- Drar nytta av ordnade segment.
- Baserad på mergesort och (binär) insättningssortering.

Sammanfattning

Idag (om vi hann med allt):

- Undre gräns.
- Radixsortering.
- Heapsort.
- Splaysort.
- Quicksort.