Föreläsning 20

Sortering och urval

TDDD86: DALP

Utskriftsversion av föreläsning i *Datastrukturer, algoritmer och programmeringsparadigm*
23 november 2015

Tommy Färnqvist, IDA, Linköpings universitet

20.1

Innehåll

Innehåll

20.2

1 Sortering

1.1 Introduktion

Sorteringsproblemet

Indata:

- En lista L innehållande data med *nycklar* från en linjärt ordnad mängd K

Utdata:

- En lista L' innehållande samma data sorterat i stigande ordning m.a.p. nycklarna

Exempel
$[8,2,9,4,6,10,1,4] \rightarrow [1,2,4,4,6,8,9,10]$

20.3

Aspekter på sortering

- in-place vs använda extra minne
- internt vs externt minne
- stabil vs icke stabil
- jämförelsebaserad vs digital

20.4

Strategier

Sortering genom insättning
Titta efter rätt plats att sätta in varje nytt element som ska läggas till i den sorterade sekvensen... *linjär insättning*, Shell-sort, ...

Sortering genom urval
Sök i varje iteration i den osorterade sekvensen efter det minsta kvarvarande datat och lägg det till slutet av den sorterade sekvensen... *rakt urval*, *Heap-sort*, ...

Sortering genom platsbyten
Sök fram och tillbaka i något mönster och byt plats på par i fel inbördes ordning så fort ett sådant upptäcks... *Quick-sort*, *Merge-sort*, ...

20.5

1.2 Insättningssortering

(Linjär) insättningssortering

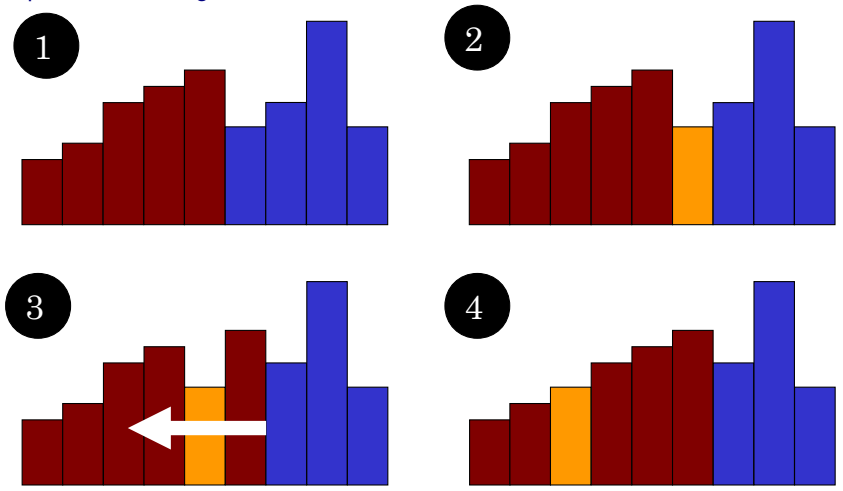
- Algoritmen är in-place!
- Dela arrayen som ska sorteras $A[0,\ldots,n-1]$ i två delar
  - $A[0,\ldots,i-1]$ som är sorterad
  - $A[i,\ldots,n-1]$ ej ännu sorterad

  Initialt är $i=1$, i vilket fall $A[0,\ldots,0]$ (trivialt) är sorterad

procedure INSERTIONSORT($A[0,\ldots,n-1]$)
  for $i=1$ **to** $n-1$ **do**
    sätt in $A[i]$ i rätt (=sorterad) position i $A[0,\ldots,i-1]$

20.6

Exempel: Visualisering av Insertion-sort



20.7

Värstafallsanalys av Insertion-sort

1: **procedure** INSERTIONSORT($A[0,\ldots,n-1]$)
2: **for** $i=1$ **to** $n-1$ **do**
3: $j \leftarrow i$; $x \leftarrow A[i]$
4: **while** $j \geq 1$ **and** $A[j-1] > x$ **do**
5: $A[j] \leftarrow A[j-1]$; $j \leftarrow j-1$
6: $A[j] \leftarrow x$

- t_2: $n-1$ pass
- t_3: $n-1$ pass
- t_4: Låt I vara antalet iterationer i värsta fallet av innerloopen:

$$I = 1+2+\ldots+(n-1) = n(n-1)/2 = (n^2-n)/2$$

- t_5: I pass
- t_6: $n-1$ pass
- Totalt: $t_2+t_3+t_4+t_5+t_6 = 3(n-1)+(n^2-n) = n^2+2n-3$ Alltså $O(n^2)$ i värsta fallet... *men bra om sekvensen nästan sorterad*

20.8

1.3 Urvalssortering

(Rak) urvalssortering

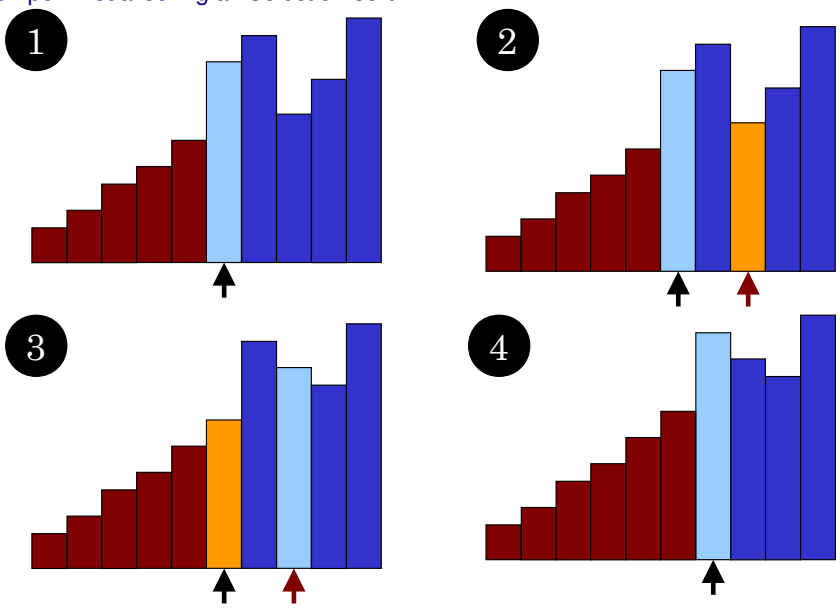
- Algoritmen är in-place!
- Dela arrayen som ska sorteras $A[0,\ldots,n-1]$ i två delar
  - $A[0,\ldots,i-1]$ som är sorterad (alla element mindre än eller lika med $A[i,\ldots,n-1]$)
  - $A[i,\ldots,n-1]$ ej ännu sorterad

  Initialt är $i=0$, d.v.s. den sorterade delen är tom (och trivialt sorterad)

procedure SELECTIONSORT($A[0,\ldots,n-1]$)
  for $i=0$ **to** $n-2$ **do**
    hitta ett minimalt element $A[j]$ i $A[i,\ldots,n-1]$
    byt plats på $A[i]$ och $A[j]$

20.9

Exempel: Visualisering av Selection-sort



20.10

Värstafallsanalys av Selection-sort

```
1: procedure SelectionSort(A[0,...,n−1])
2:     for i = 0 to n−2 do
3:         s ← i
4:         for j ≥ i+1 to n−1 do
5:             if A[j] < A[s] then s ← j
6:         Swap(A[i],A[s])
```

- t_2: $n-1$ pass
- t_3: $n-1$ pass
- t_4: Låt I vara antalet iterationer i värsta fallet av innerloopen:

$$I = (n-2)+(n-3)+\ldots+1 = (n-1)(n-2)/2 = (n^2-3n+2)/2$$

- t_5: I pass
- t_6: $n-1$ pass
- Totalt: $t_2+t_3+t_4+t_5+t_6 = 3(n-1)+(n^2-3n+2) = n^2-1 \in O(n^2)$

20.11

1.4 Divide-and-conquer

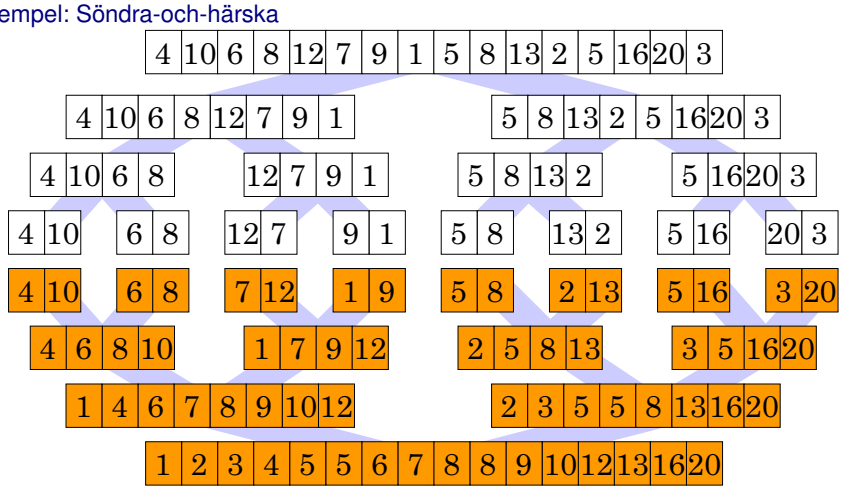
Principen söndra-och-härska för algoritmkonstruktion

- söndra: dela upp problemet i mindre, oberoende, delproblem
- härska: lös delproblemen rekursivt (eller direkt om trivialt)
- kombinera lösningarna till en lösning till ursprungsproblemet

Eng. *divide-and-conquer*

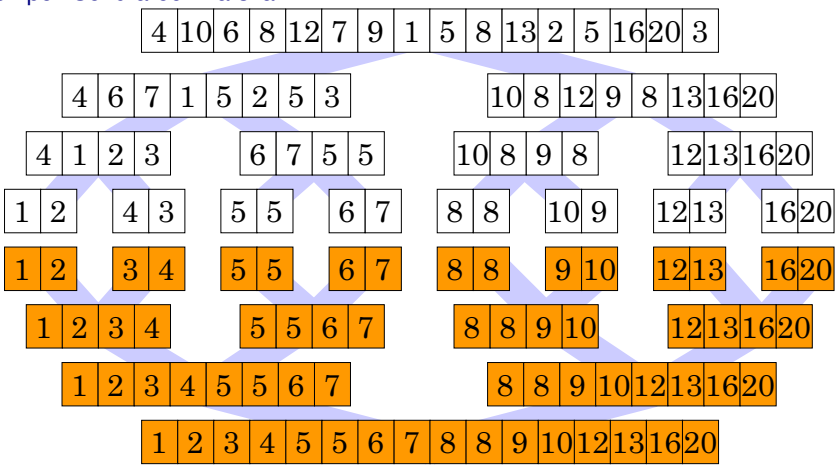
20.12

Exempel: Söndra-och-härska



20.13

Exempel: Söndra-och-härska



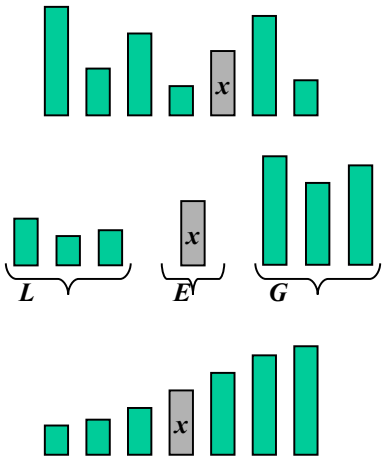
20.14

1.5 Quick-sort

Quick-sort

Quick-sort är en *randomiserad* sorteringsalgoritm baserad på paradigmet söndra-och-härska

- söndra: välj slumpvist ett element x (kallat pivot) och partitionera S till
  - L element mindre än x
  - E element lika med x
  - G element större än x
- härska: sortera L och G
- kombinera L, E och G



20.15

Partitionering

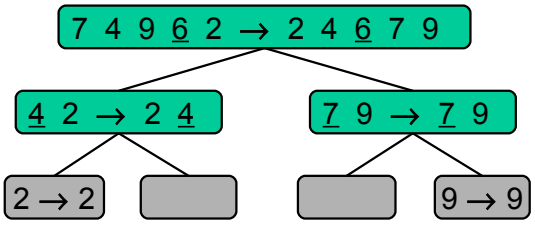
- Vi partitionerar indatasekvensen på följande vis:
  - Vi tar bort, i tur och ordning, varje element y från S och
  - Vi sätter in y i L, E eller G beroende på resultatet av jämförelsen med pivot-elementet x
- Varje insättning och borttagning är i början eller slutet av en sekvens och tar alltså $O(1)$ tid
- Alltså tar partitioneringssteget i quick-sort $O(n)$ tid

```
function PARTITION(S, p)
    L, E, G ← tomma sekvenser
    x ← S.REMOVE(p)
    while ¬S.ISEMPTY() do
        y ← S.REMOVE(S.FIRST())
        if y < x then
            L.INSERTLAST(y)
        else if y = x then
            E.INSERTLAST(y)
        else
            G.INSERTLAST(y)
    return L, E, G
```

20.16

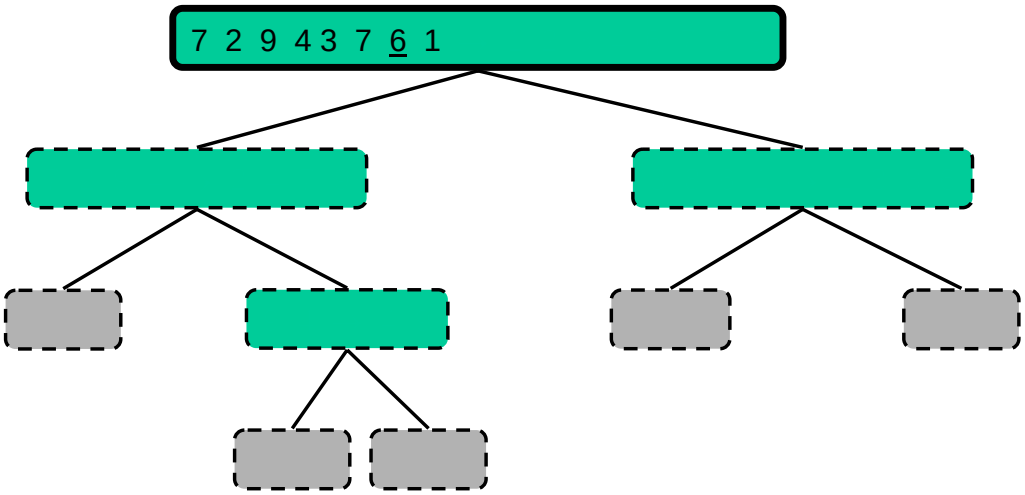
Quick-sortträdet

- Exekveringen av quick-sort kan visualiseras som ett binärt träd
  - Varje nod representerar ett rekursivt anrop till quick-sort och lagrar
    - Osorterad sekvens före exekveringen och dess pivot
    - Sorterad sekvens efter exekveringen
  - Roten är ursprungsanropet
  - Löven är anrop på delsekvenser av storlek 0 eller 1



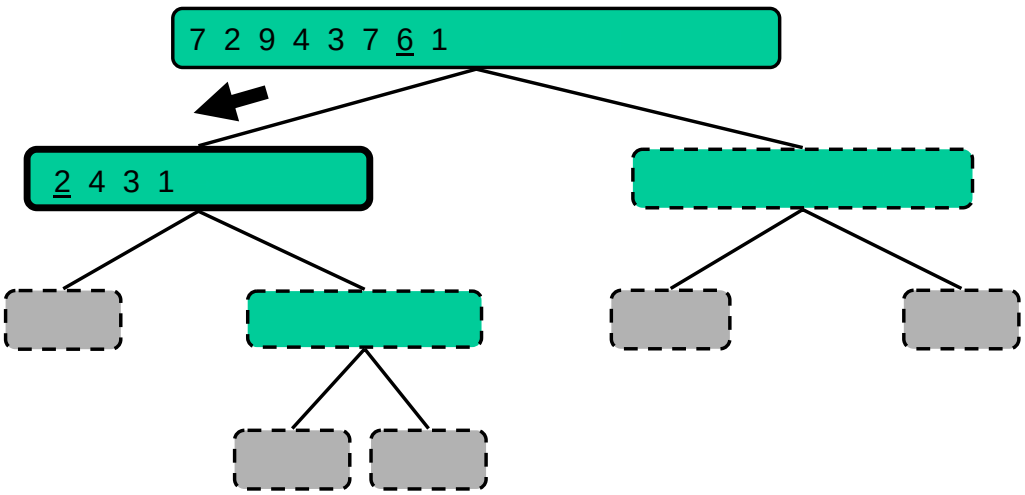
Exempel: Exekvering av quick-sort

- Val av pivot



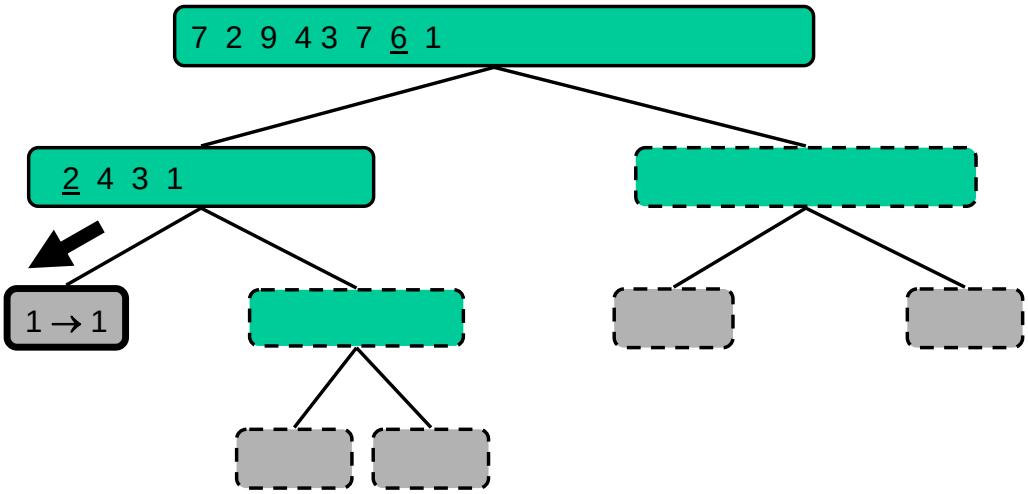
Exempel: Exekvering av quick-sort

- Partitionering, rekursivt anrop, val av pivot



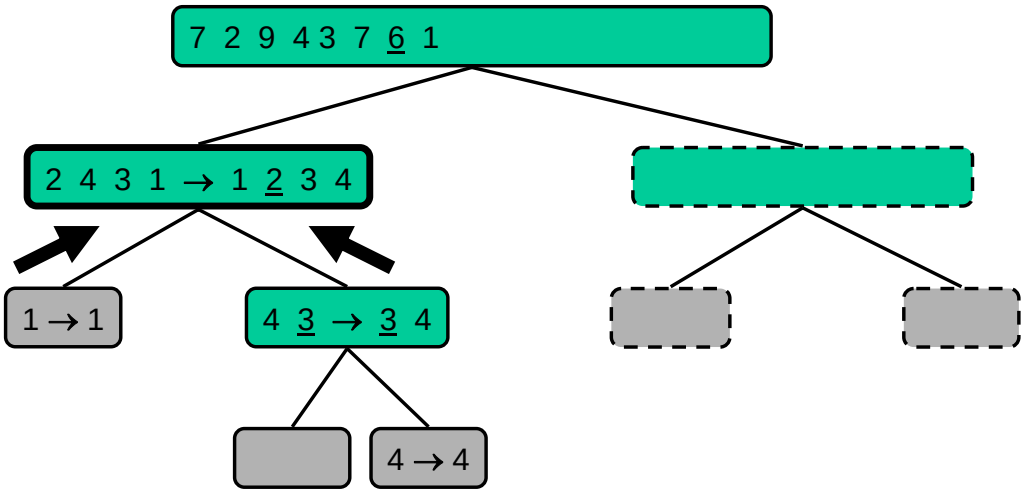
Exempel: Exekvering av quick-sort

- Partitionering, rekursivt anrop, basfall



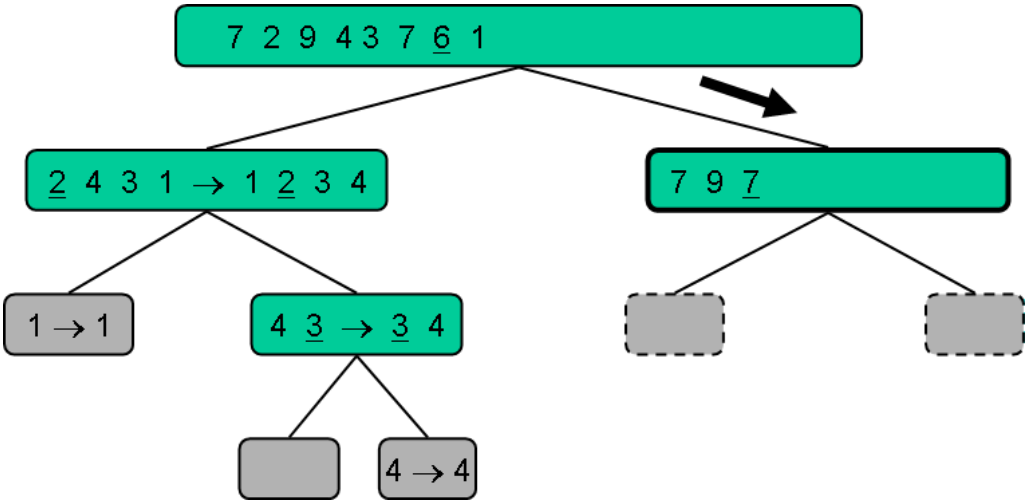
Exempel: Exekvering av quick-sort

- Rekursivt anrop, . . . , basfall, kombinera



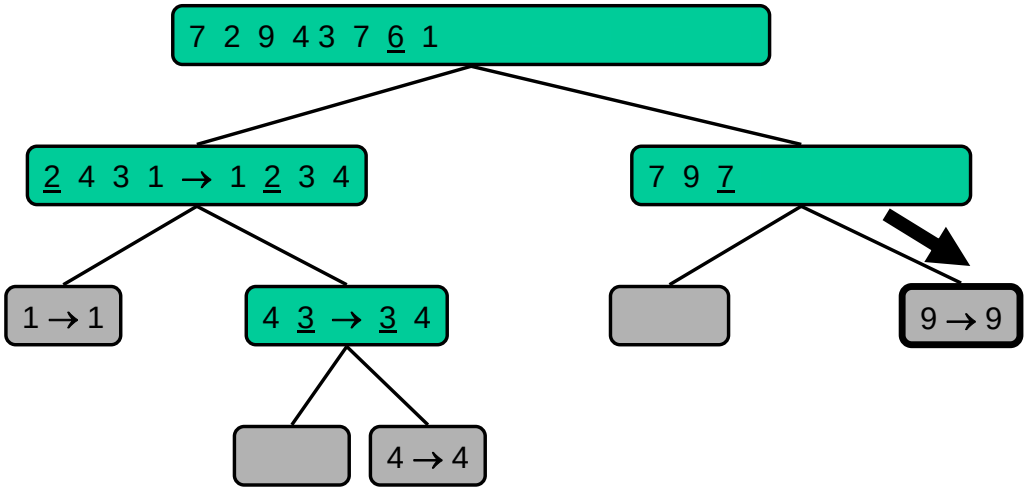
Exempel: Exekvering av quick-sort

- Rekursivt anrop, val av pivot



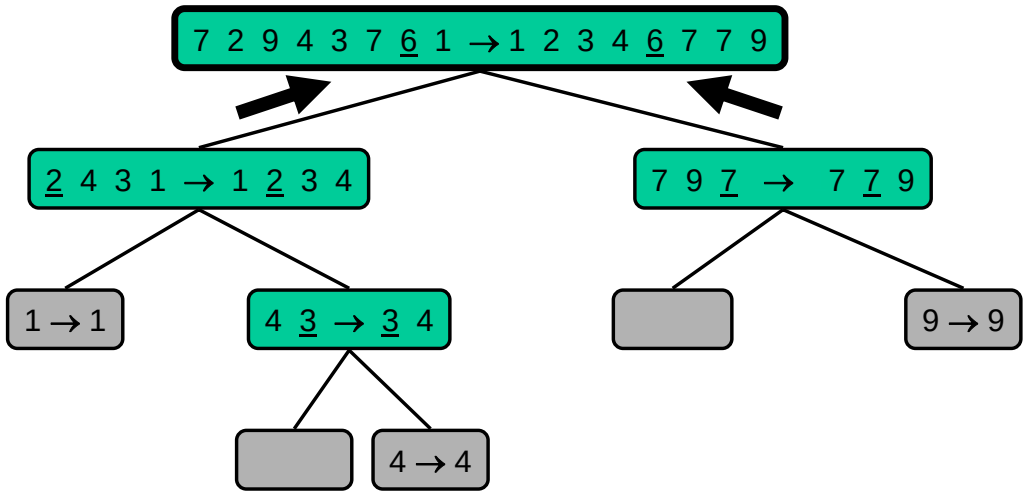
Exempel: Exekvering av quick-sort

- Partitionering, . . . , rekursivt anrop, basfall



Exempel: Exekvering av quick-sort

- Kombinera, kombinera



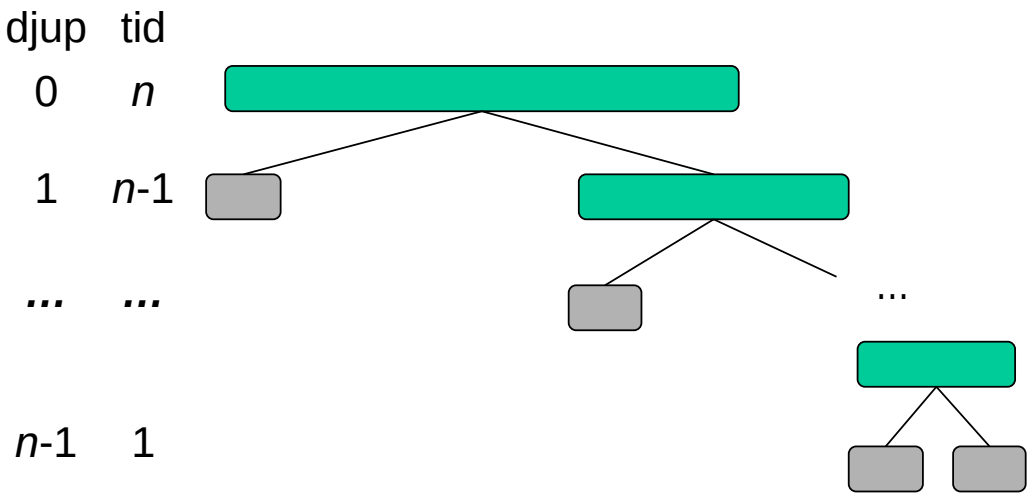
Exekveringstid i värsta fallet

- Värsta fallet för quick-sort uppkommer när pivotelementet är ett unikt minimalt eller maximalt element
- en av L eller G har storlek $n-1$ och den andra har storlek 0
- Exekveringstiden blir proportionell mot summan

$$n+(n-1)+\ldots+2+1$$

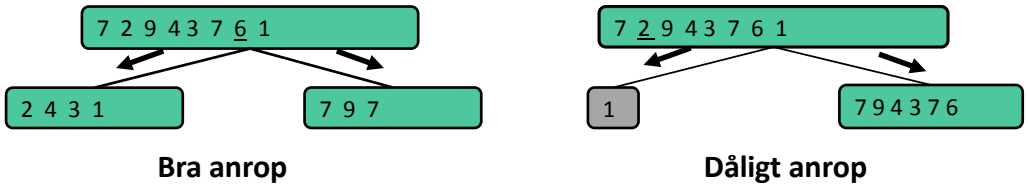
- Alltså, värstafallstiden för quick-sort är $O(n^2)$

Exekveringstid i värsta fallet

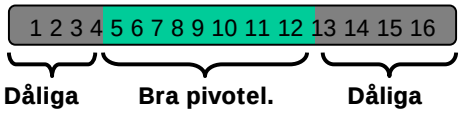


Förväntad exekveringstid

- Betrakta ett rekursivt anrop till quick-sort på en sekvens av storlek s
  - Bra anrop: storlekarna av L och G är båda $< 3s/4$
  - Dåligt anrop: en av L och G har storlek $\geq 3s/4$



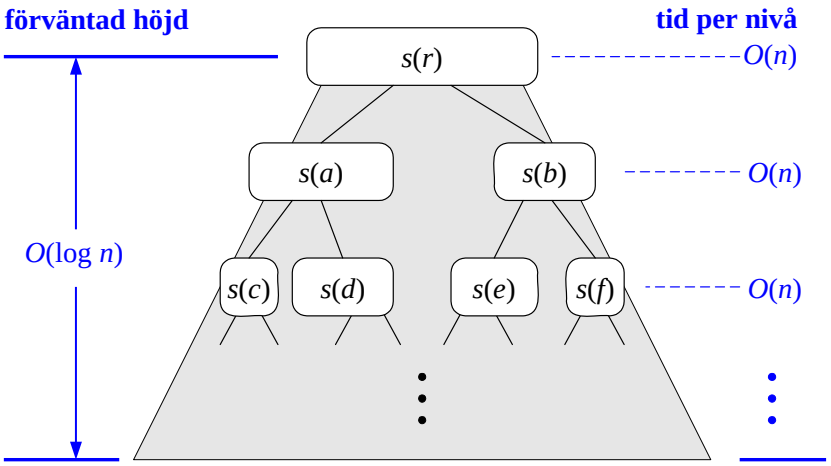
- Ett anrop är bra med sannolikhet 1/2
  - Hälften av alla möjliga pivotelement orsakar bra anrop:



Förväntad exekveringstid

- Probabilistiskt faktum: Det förväntade antalet slant-singlingar som behövs för att få klave k gånger är $2k$
- För en nod på djup i förväntar vi oss att
  - $i/2$ förfädrar är bra anrop
  - storleken på indatasekvensen för det aktuella anropet är som mest $(3/4)^{i/2}n$
- Alltså har vi att
  - För en nod på djup $2\log_{4/3} n$ är den förväntade storleken på indata 1
  - Den förväntade höjden på quick-sortträdet är $O(\log n)$
- Mängden arbete som utförs i noderna på samma djup är $O(n)$
- Alltså är den förväntade exekveringstiden för quick-sort $O(n \log n)$

Förväntad exekveringstid

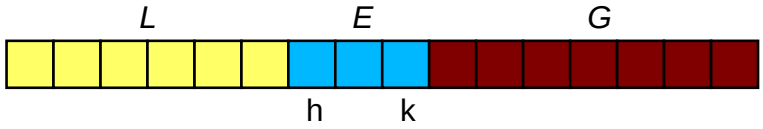


förväntad total tid: $O(n \log n)$

20.29

Quick-sort med konstant extra minne

- Quick-sort kan implementeras för att köra *in-place*
- I partitioneringssteget använder vi ersättningsoperationer för att arrangera om elementen i indatasekvensen så att
  - elementen mindre än pivotelementet har rank mindre än h
  - elementen lika med pivotelementet har rank mindre mellan h och k
  - elementen större än pivotelementet har rank större än k



20.30

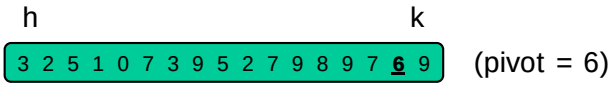
Algoritm för quick-sort med konstant extra minne

procedure INPLACEQUICKSORT(S, l, r)
  if $l \geq r$ **then return**
  $i \leftarrow$ slumpvist heltal mellan l och r
  $x \leftarrow S$.ELEMATRANK(i)
  $(h,k) \leftarrow$INPLACEPARTITION(x)
  INPLACEQUICKSORT($S, l, h-1$)
  INPLACEQUICKSORT($S, k+1, r$)

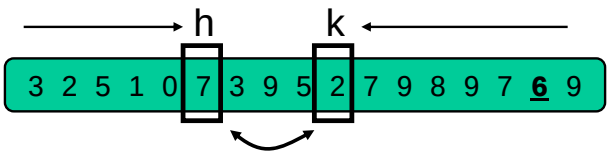
20.31

Partitionering med konstant extra minne

- Utför partitioneringen m.h.a. två index för att dela S i L och $E \cup G$ (en liknande metod kan användas för att dela $E \cup G$ i E och G)



- Upprepa till h och k möts/korsar varandra:
  - Svep h åt höger till ett element $\geq$ pivotelementet hittas
  - Svep k åt vänster till ett element $<$ pivotelementet hittas
  - Byt plats på elementen vid platserna h och k



20.32

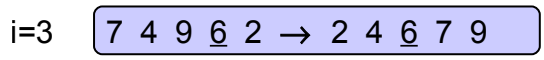
2 Urval

2.1 Introduktion

Urvalsproblemet

- Givet ett heltal i och n element $x_1, x_2, \ldots, x_n$ tagna från en totalordning, hitta det i:te minsta elementet i sekvensen.

- Vi kan sortera sekvensen i $O(n \log n)$ tid och sedan indexera det i:te elementet i konstant tid.



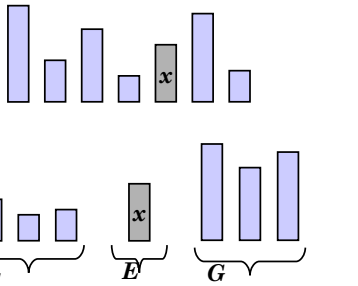
- Kan vi lösa urvalsproblemet snabbare?

2.2 Quick-select

Quick-select

Quick-Select är en randomiserad urvalsalgoritm baserat på paradigmet *prune-and-search*:

- *Prune*: välj x slumpmässigt och partitionera S till
  - L element mindre än x
  - E element lika med x
  - G element större än x
- *Search*: beroende på i finns antingen svaret i E eller så behöver vi fortsätta rekursivt i någon av L eller G



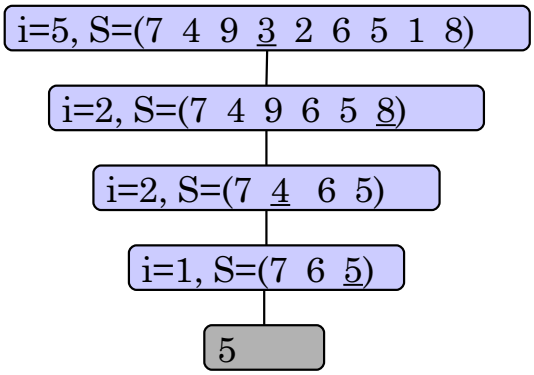
Partitionering

- Vi partitionerar indatasekvensen som i algoritmen för quick-sort:
  - Vi tar bort, i tur och ordning, varje element y från S och
  - Vi sätter in y i L, E eller G beroende på resultatet av jämförelsen med pivot-elementet x
- Varje insättning och borttagning är i början eller slutet av en sekvens och tar alltså $O(1)$ tid
- Alltså tar partitioneringssteget i quick-select $O(n)$ tid

```
function PARTITION(S, p)
    L, E, G ← tomma sekvenser
    x ← S.REMOVE(p)
    while ¬S.ISEMPTY() do
        y ← S.REMOVE(S.FIRST())
        if y < x then
            L.INSERTLAST(y)
        else if y = x then
            E.INSERTLAST(y)
        else
            G.INSERTLAST(y)
    return L, E, G
```

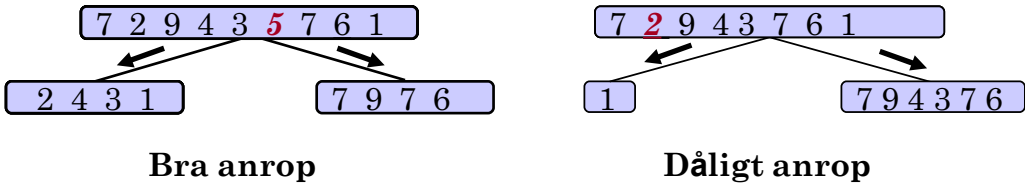
Visualisering av Quick-select

- Exekveringen av quick-select kan visualiseras m.h.a. en rekursionsstig
  - Varje nod representerar ett rekursivt anrop till quick-select och lagrar i och den kvarvarande sekvensen S



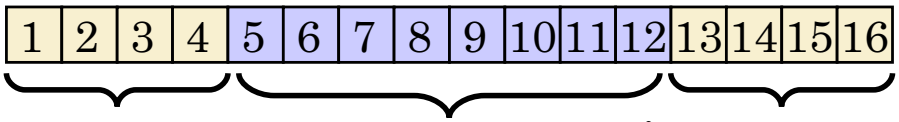
Förväntad exekveringstid

- Betrakta ett rekursivt anrop till quick-select på en sekvens av storlek s
  - Bra anrop: storlekarna av L och G är båda $< 3s/4$
  - Dåligt anrop: en av L och G har storlek $\geq 3s/4$



Bra anrop **Dåligt anrop**

- Ett anrop är bra med sannolikhet 0.5
  - Hälften av alla möjliga pivotelement orsakar bra anrop:



Dåliga pivotelement Bra pivotelement Dåliga pivotelement

Förväntad exekveringstid

- Probabilistiskt faktum: Det förväntade antalet slantsinglingar som behövs för att få klave en gång är två
- Probabilistiskt faktum: Väntevärdet är en linjär funktion:
  - $E(X+Y) = E(X)+E(Y)$
  - $E(cX) = cE(X)$ för varje konstant c
- Låt $T(n)$ vara den förväntade exekveringstiden för quick-select
- Av det andra faktumet får vi $T(n) \leq b \cdot n \cdot g(n) + T(3n/4)$ där
  - b är någon konstant
  - $g(n)$ är det förväntade antalet anrop innan ett bra anrop inträffar

Förväntad exekveringstid

- Alltså
  - $T(n) \leq b \cdot n \cdot g(n) + T(3n/4)$
- Genom det första faktumet får vi
  - $T(n) \leq 2 \cdot b \cdot n + T(3n/4)$
- D.v.s. $T(n)$ är en geometrisk serie:
  - $T(n) \leq 2 \cdot b \cdot n + 2 \cdot b \cdot n \cdot (3/4) + 2 \cdot b \cdot n \cdot (3/4)^2 + 2 \cdot b \cdot n \cdot (3/4)^3 + \ldots$
- Alltså gäller $T(n) \in O(n)$
- Vi kan lösa urvalsproblemet i förväntad tid $O(n)$ (värstafallstiden är $O(n^2)$)