Föreläsning 21

Heap-sort, merge-sort. Undre gränser för sortering. Sortering i linjär tid?

TDDD86: DALP

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

Tommy Färnqvist, IDA, Linköpings universitet

Innehåll

Innehåll

1 Sortering

1.1 Heap-sort

Sortering med prioritetskö

- Använd en prioritetskö för att sortera en samling jämförbara element
  - Sätt in elementen med en serie insättningsoperationer
  - Ta bort elementen i sorterad ordning med en serie removeMin-operationer
- Körtiden beror på implementationen av prioritetskön:
  - Osorterad sekvens ger urvalssortering och $O(n^2)$ tid
  - Sorterad sekvens ger insättningssortering och $O(n^2)$ tid
- Kan vi åstadkomma något bättre?

```
procedure PQSort(S)
    P ← tom priokö
    while ¬S.isEmpty() do
        e ← S.remove(S.first())
        P.insert(e)
    while ¬P.isEmpty() do
        e ← P.removeMin()
        S.insertLast(e)
```

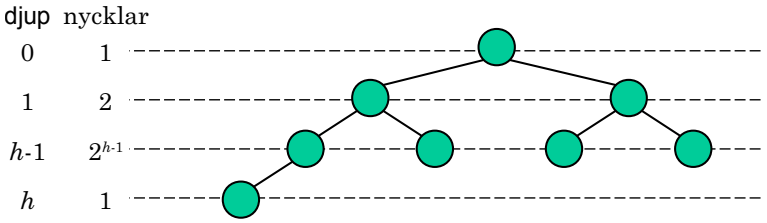
Höjden av en heap

Proposition 1. En heap som lagrar n nycklar har höjd $O(\log n)$

Bevis. Vi använder att en heap är ett fullständigt binärt träd.

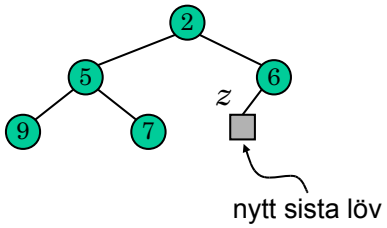
- Låt h vara höjden av en heap som lagrar n nycklar
- Eftersom det finns 2^i nycklar på djup $i = 0, \ldots h-1$ och minst en nyckel på djup h får vi $n \geq 1+2+4+\ldots+2^{h-1}+1$
- Alltså är $n \geq 2^h$, d.v.s. $h \leq log_2 n$

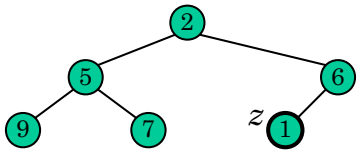
□



Insättning i en heap

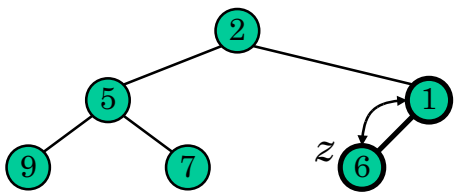
- Metoden insert i ADT priokö svarar mot insättning av nyckel k i heapen
- Insättningsalgoritmen består av tre steg
  - Hitta platsen för insättning z (det nya sista lövet)
  - Lagra k vid z
  - Återställ heapegenskapen

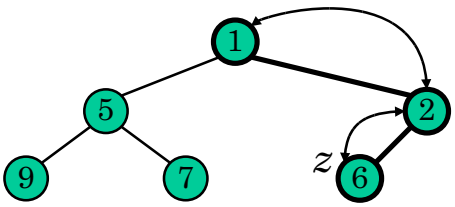




Upheap

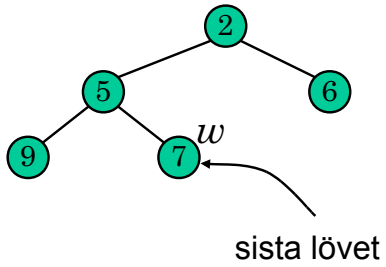
- Efter insättning av en ny nyckel k är det inte säkert att heapegenskapen fortfarande är uppfylld
- Metoden upheap återställer heapegenskapen genom att byta k längs uppåtgående stig från insättningsnoden
- upheap terminerar när nyckel k når roten eller en nod vars förälder har en nyckel som inte är större än k
- Eftersom en heap har höjd $O(\log n)$ går upheap i tid $O(\log n)$

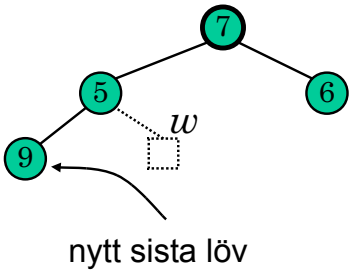




Borttagning från en heap

- Metoden removeMin i ADT priokö svarar mot borttagning av rotnyckeln från heapen
- Borttagningsalgoritmen består av tre steg
  - Ersätt rotnyckeln med nyckeln i det sista lövet w
  - Ta bort w
  - Återställ heapegenskapen

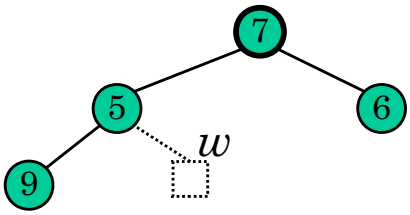


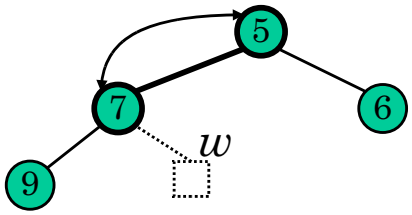


21.7

Downheap

- Efter ersättning av rotnyckeln med nyckel k från sista lövet är det inte säkert att heapegenskapen fortfarande är uppfylld
- Metoden downheap återställer heapegenskapen genom att byta k längs nedåtgående stig från insättningsnoden
- downheap terminerar när nyckel k når ett löv eller en nod vars barn har nycklar som inte är mindre än k
- Eftersom en heap har höjd $O(\log n)$ går downheap i tid $O(\log n)$





21.8

Heap-sort

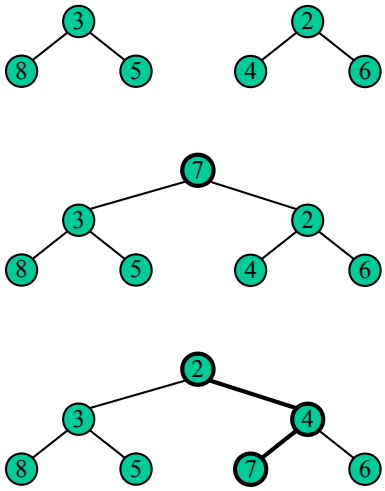
- Betrakta en priokö med n element implementerad m.h.a. en heap
  - minnesåtgången är $O(n)$
  - insert och removeMin tar $O(\log n)$ tid
  - size, isEmpty och min tar $O(1)$ tid

- Genom att använda en heapbaserad priokö kan vi sortera en sekvens av n element i $O(n \log n)$ tid
- Den resulterande algoritmen kallas heap-sort
- Heap-sort är mycket snabbare än kvadratiska sorteringsalgoritmer

21.9

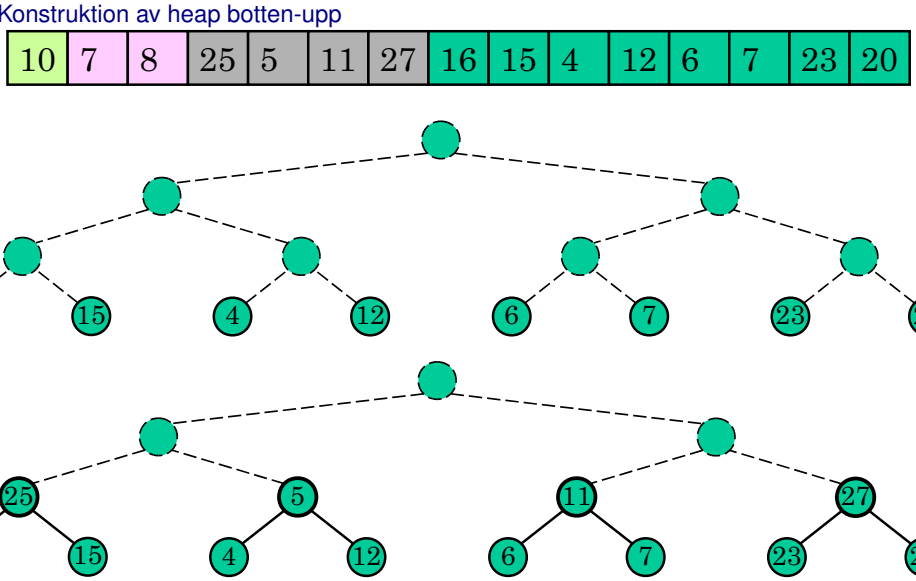
Slå ihop två heapar

- Givet två heapar och en nyckel k
- Skapa en ny heap där rotnoden lagrar nyckel k och med de två givna heaparna som delträd
- Kör downheap för att återställa heapegenskapen



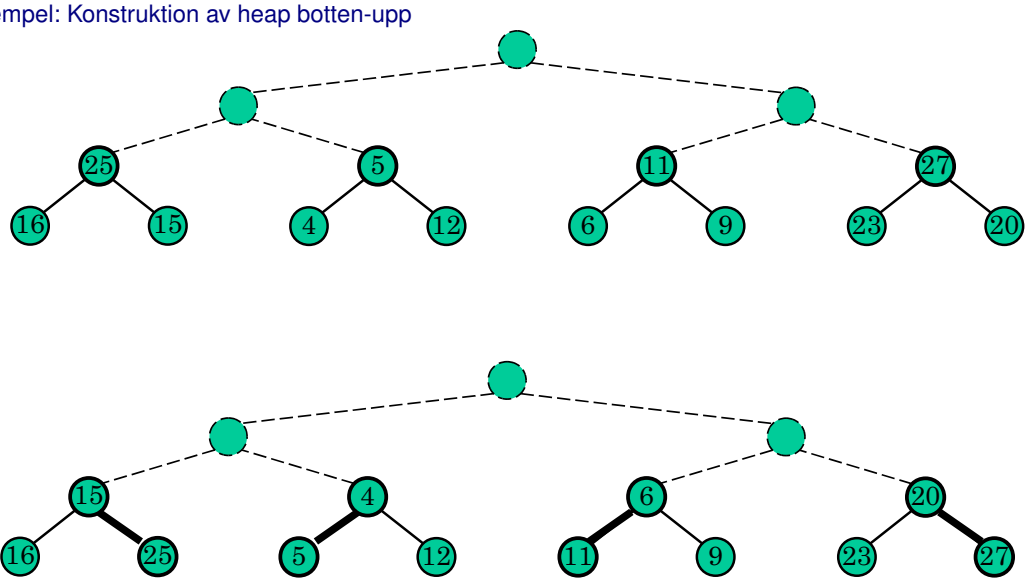
21.10

Exempel: Konstruktion av heap botten-upp



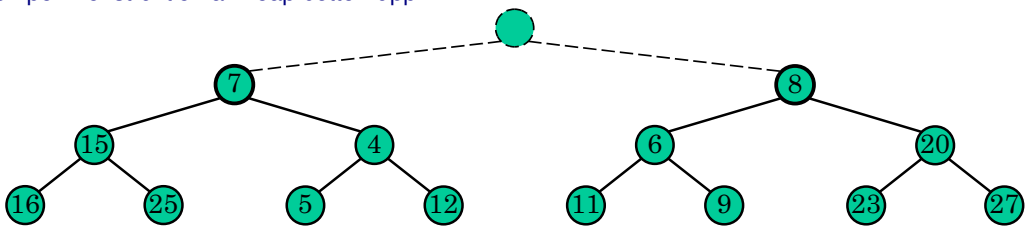
21.11

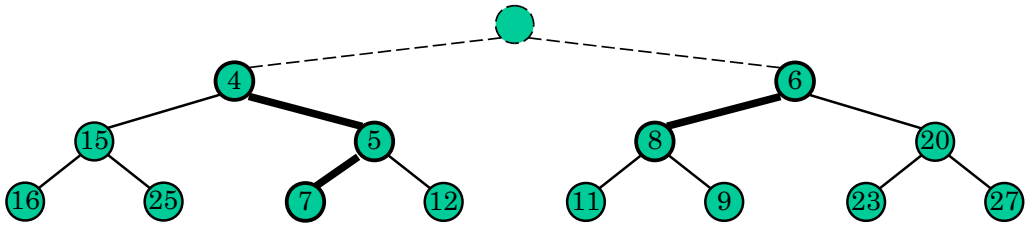
Exempel: Konstruktion av heap botten-upp



21.12

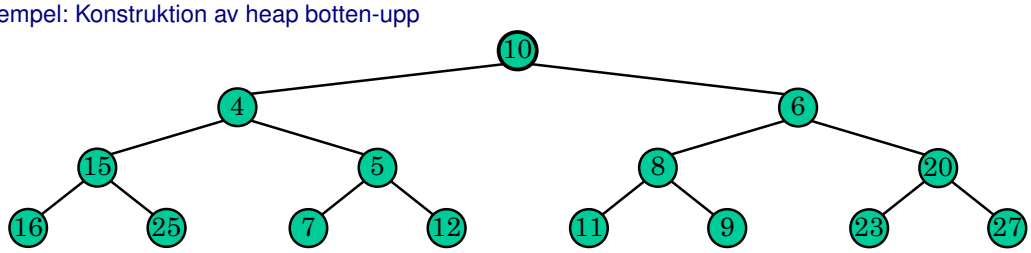
Exempel: Konstruktion av heap botten-upp

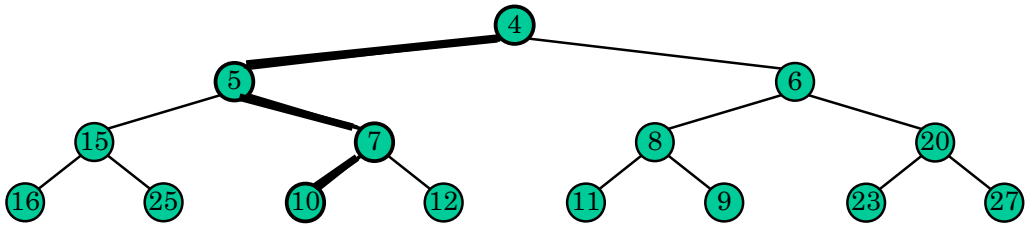




21.13

Exempel: Konstruktion av heap botten-upp

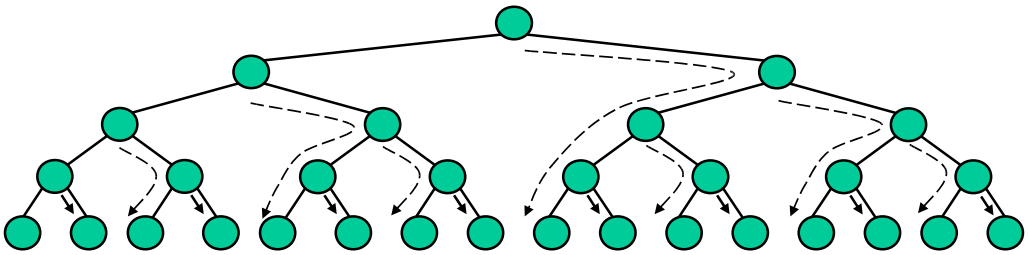




21.14

Analys

- Vi visualiserar värstafallstiden för ett anrop till downheap med en stig som först går till höger och sedan upprepade gånger går till vänster till botten av heapen
- Eftersom varje nod traverseras av som mest två sådana stigar är det totala antalet stigar $O(n)$
- Alltså är tiden för att konstruera en heap botten-upp $O(n)$
- Den här konstruktionsmetoden är snabbare än n upprepade insättningar och gör att första fasen av heap-sort blir effektivare



21.15

1.2 Merge-sort

Söndra-och-härska

- Merge-sort är en sorteringsalgoritm baserad på söndra-och-härska
- Likt heap-sort

  - har den exekveringstid $O(n \log n)$
- Olikt heap-sort
  - använder den inte en priokö till hjälp
  - accessar den data på ett sekvensiellt sätt (lämpligt för att sortera data på disk)

21.16

Merge-sort

Merge-sort på en indatasekvens S med n element består av tre steg:

- Söndra: dela S i två sekvenser S_1 och S_2 med vardera ca $n/2$ element
- Härska: sortera S_1 och S_2 rekursivt
- Kombinera: slå ihop (merge) S_1 och S_2 till en unik sorterad sekvens

```
procedure MERGESORT(S)
    if S.SIZE() > 1 then
        (S1, S2) ←PARTITION(S.SIZE()/2)
        MERGESORT(S1)
        MERGESORT(S2)
        S ←MERGE(S1, S2)
```

21.17

Slå ihop två sorterade sekvenser

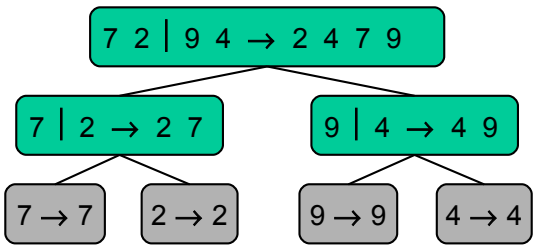
- Kombineringssteget: slå ihop två sekvenser A och B till en sorterad sekvens S innehållande unionen av elementen i A och B
- Att slå ihop två sorterade sekvenser, var och en med $n/2$ element implementerade med dubbellänkade listor, tar $O(n)$ tid

```
function MERGE(A, B)
    S ← tom sekvens
    while ¬A.ISEMPTY() ∧ ¬B.ISEMPTY() do
        if A.FIRST.ELEMENT() < B.FIRST.ELEMENT() then
            S.INSERTLAST(A.REMOVE(A.FIRST()))
        else
            S.INSERTLAST(B.REMOVE(B.FIRST()))
    while ¬A.ISEMPTY() do
        S.INSERTLAST(A.REMOVE(A.FIRST()))
    while ¬B.ISEMPTY() do
        S.INSERTLAST(B.REMOVE(B.FIRST()))
    return S
```

21.18

Merge-sortträdet

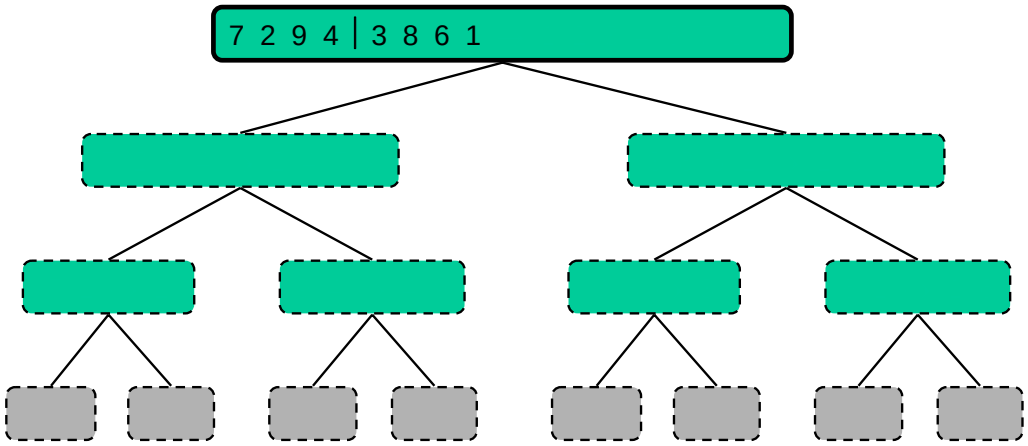
- Exekveringen av merge-sort kan visualiseras som ett binärt träd
  - Varje nod representerar ett rekursivt anrop till merge-sort och lagrar
    - Osorterad sekvens före exekveringen och dess partition
    - Sorterad sekvens efter exekveringen
  - Roten är ursprungsanropet
  - Löven är anrop på delsekvenser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



21.19

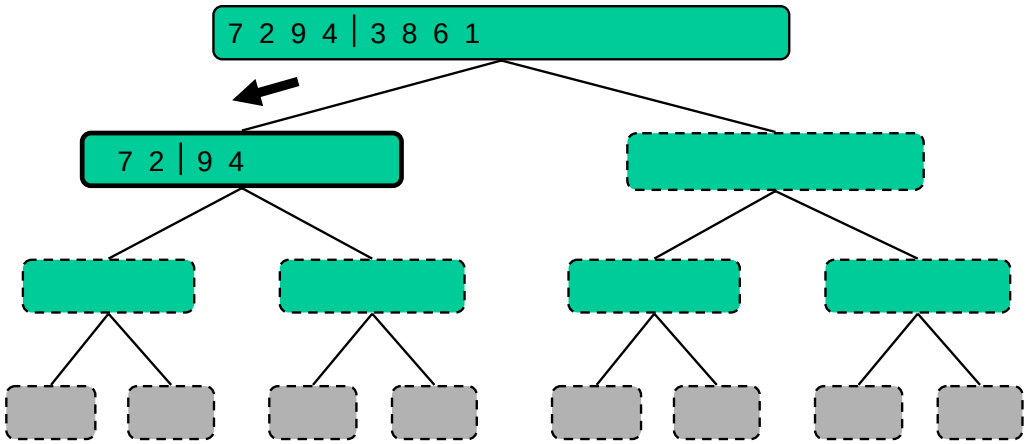
Exempel: Exekvering av merge-sort

- Partitionera



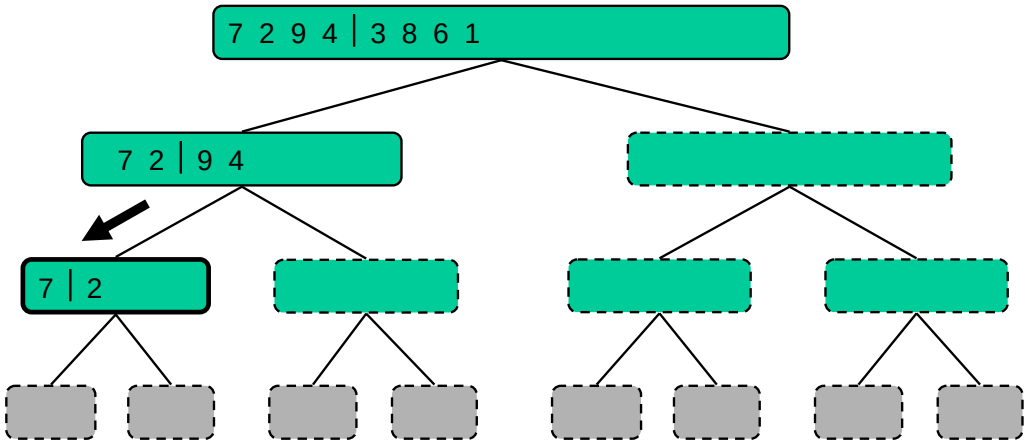
Exempel: Exekvering av merge-sort

- Rekursivt anrop, partitionera



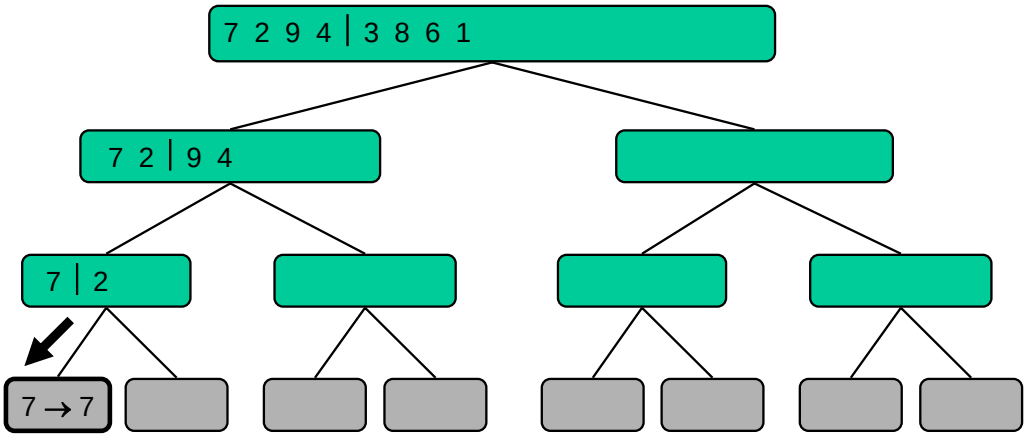
Exempel: Exekvering av merge-sort

- Rekursivt anrop, partitionera



Exempel: Exekvering av merge-sort

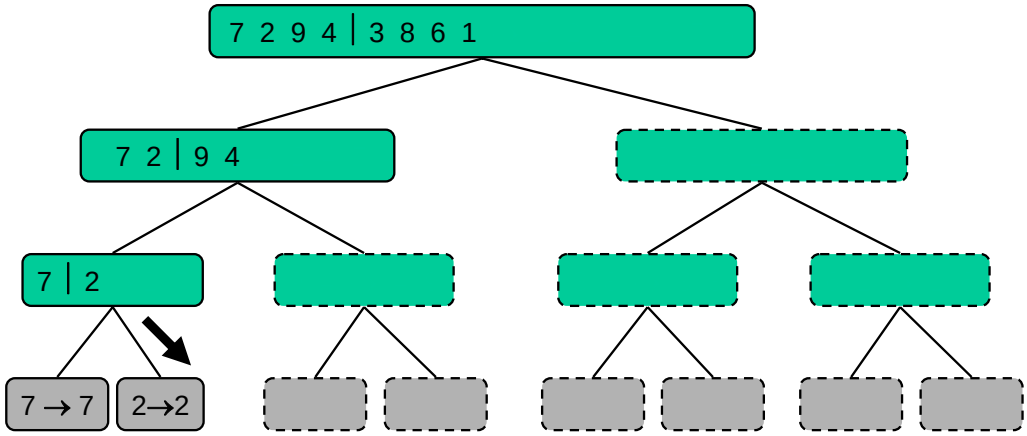
- Rekursivt anrop, basfall



21.23

Exempel: Exekvering av merge-sort

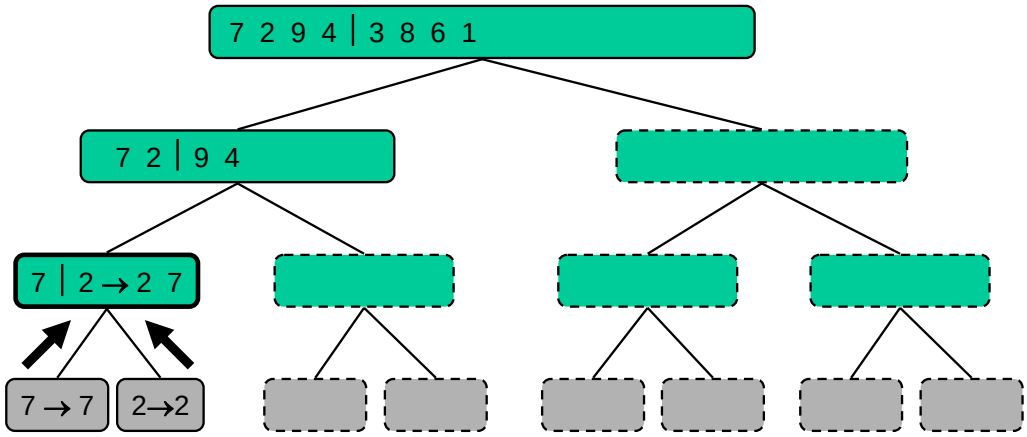
- Rekursivt anrop, basfall



21.24

Exempel: Exekvering av merge-sort

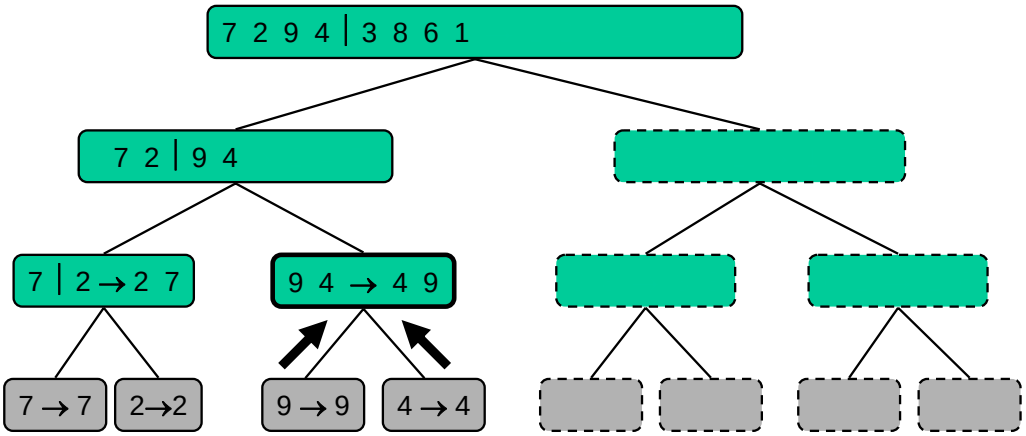
- Slå ihop



21.25

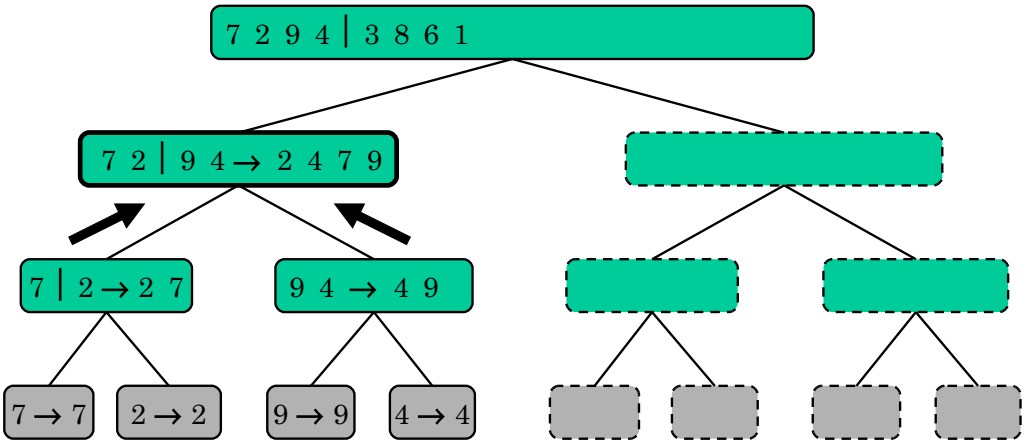
Exempel: Exekvering av merge-sort

- Rekursivt anrop, . . . , basfall



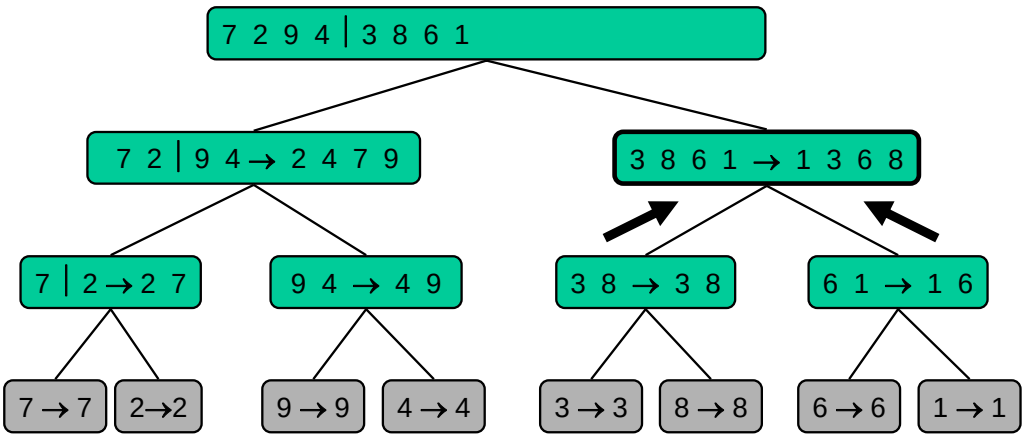
Exempel: Exekvering av merge-sort

- Slå ihop



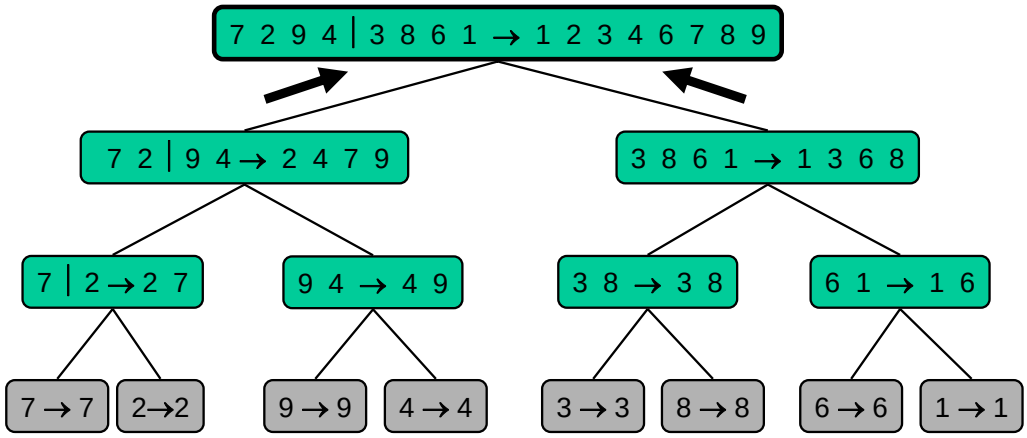
Exempel: Exekvering av merge-sort

- Rekursivt anrop, . . . , slå ihop, slå ihop



Exempel: Exekvering av merge-sort

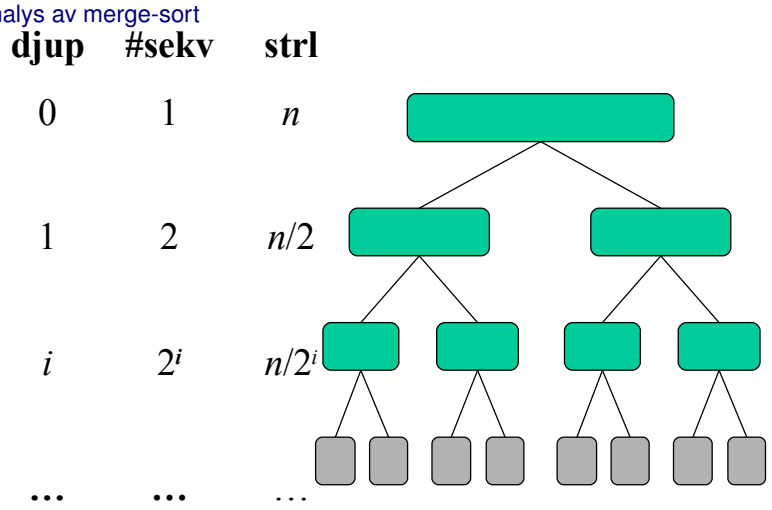
- Slå ihop



Analys av merge-sort

- Höjden h av merge-sortträdet är $O(\log n)$
  - vid varje rekursivt anrop delar vi sekvensen på mitten
- Det totala arbetet som utförs i noderna på djup i är $O(n)$
  - vi partitionerar och slår ihop 2^i sekvenser av storlek $n/2^i$
  - vi gör 2^{i+1} rekursiva anrop
- Alltså är den totala exekveringstiden för merge-sort $O(n \log n)$

Analys av merge-sort



1.3 Sammanfattning

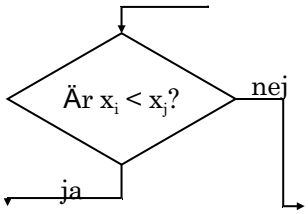
Sammanfattning så långt

Algoritm	Tid	Noteringar
selection-sort	$O(n^2)$	• in-place • långsam (bra för små indata)
insertion-sort	$O(n^2)$	• in-place • långsam (bra för små indata)
quick-sort	$O(n \log n)$ förväntad	• in-place, randomiserad • snabbast (bra för stora indata)
heap-sort	$O(n \log n)$	• in-place • snabb (bra för stora indata)
merge-sort	$O(n \log n)$	• sekvensiell dataaccess • snabb (bra för enorma indata)

2 En undre gräns för jämförelsebaserad sortering

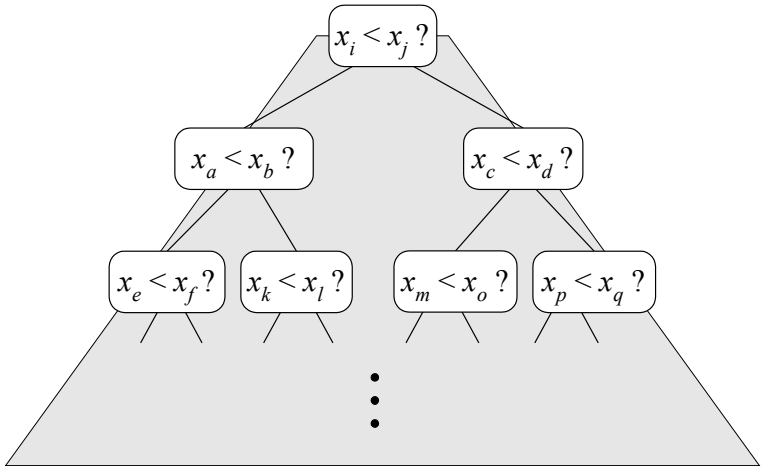
Jämförelsebaserad sortering

- Många sorteringsalgoritmer är *jämförelsebaserade*
  - De sorterar genom att göra jämförelser mellan par av objekt
  - Exempel: insertion-sort, selection-sort, heap-sort, merge-sort, quick-sort, ...
- Låt oss därför försöka härleda en undre gräns för exekveringstiden i värsta fall för varje algoritm som använder jämförelse för att sortera n element $x_1, x_2, \ldots, x_n$



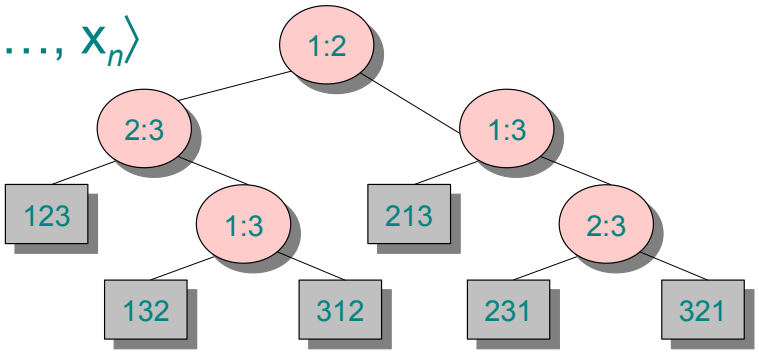
Räkna jämförelser

- Låt oss enbart räkna jämförelser
- Varje möjlig körning av en algoritm motsvaras av en rot-till-lövstig i ett *beslutsträd*



Exempel: Beslutsträd

Sort $\langle x_1, x_2, \ldots, x_n \rangle$

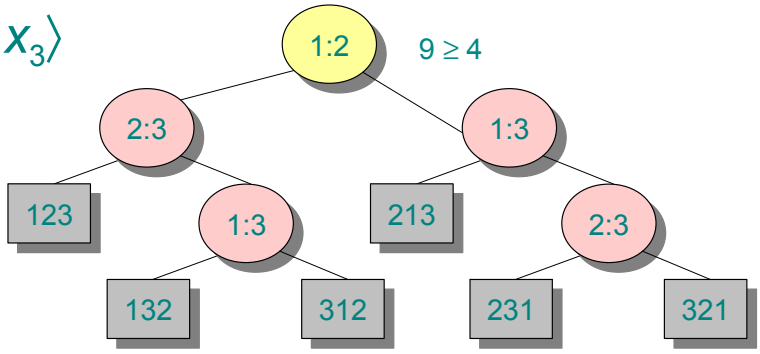


Varje intern nod är märkt $i : j$ för $i, j \in \{1, 2, \ldots, n\}$

- Vänster delträd visar efterföljande jämförelser om $x_i \leq x_j$
- Höger delträd visar efterföljande jämförelser om $x_i \geq x_j$

Exempel: Beslutsträd

Sort $\langle x_1, x_2, x_3 \rangle$ $= \langle\ 9, 4, 6\ \rangle$:

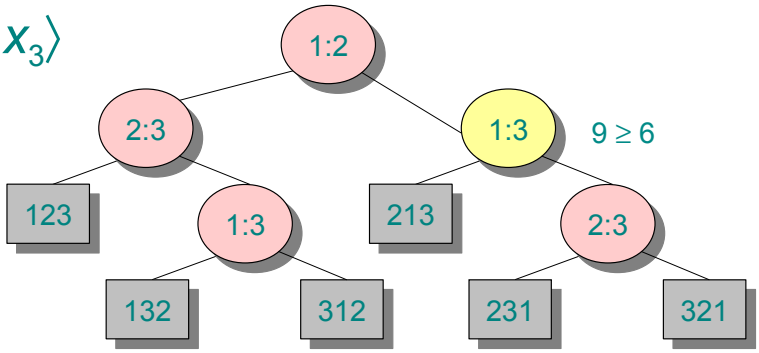


Varje intern nod är märkt $i : j$ för $i, j \in \{1, 2, \ldots, n\}$

- Vänster delträd visar efterföljande jämförelser om $x_i \leq x_j$
- Höger delträd visar efterföljande jämförelser om $x_i \geq x_j$

Exempel: Beslutsträd

Sort $\langle x_1, x_2, x_3 \rangle$ $= \langle\ 9, 4, 6\ \rangle$:

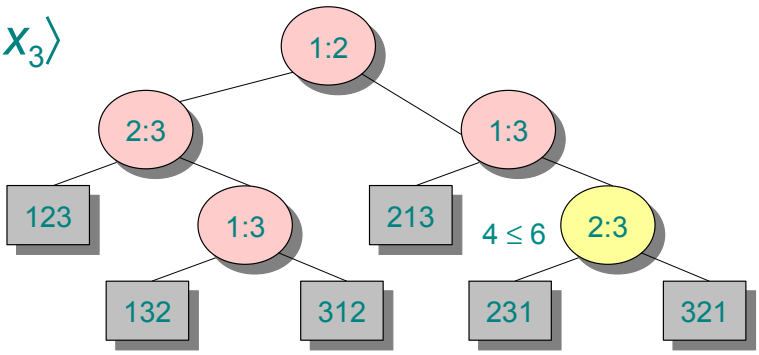


Varje intern nod är märkt $i : j$ för $i, j \in \{1, 2, \ldots, n\}$

- Vänster delträd visar efterföljande jämförelser om $x_i \leq x_j$
- Höger delträd visar efterföljande jämförelser om $x_i \geq x_j$

Exempel: Beslutsträd

Sort $\langle x_1, x_2, x_3 \rangle$ $= \langle\ 9, 4, 6\ \rangle$:



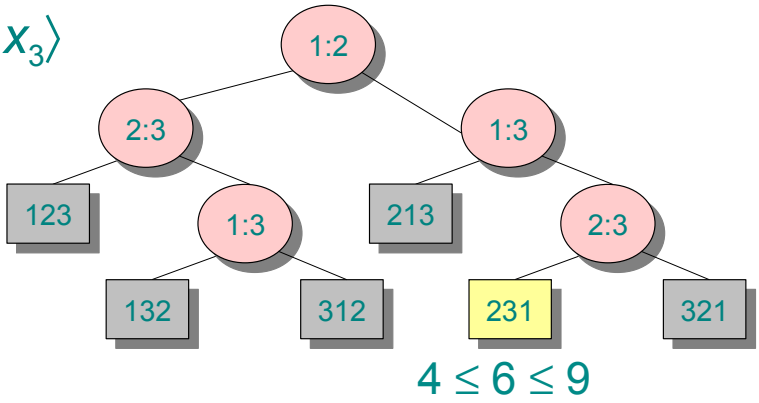
Varje intern nod är märkt $i : j$ för $i, j \in \{1, 2, \ldots, n\}$

- Vänster delträd visar efterföljande jämförelser om $x_i \leq x_j$
- Höger delträd visar efterföljande jämförelser om $x_i \geq x_j$

21.38

Exempel: Beslutsträd

Sort $\langle x_1, x_2, x_3 \rangle$ $= \langle 9, 4, 6 \rangle$:



Varje löv innehåller en permutation $\langle \pi(i), \pi(2), \ldots, \pi(n) \rangle$ för att indikera att ordningen $x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)}$ har etablerats

21.39

Beslutsträdsmodellen

Ett beslutsträd kan modellera exekveringen av vilken jämförelsebaserad sorteringsalgoritm som helst:

- Ett träd för varje storlek på indata
- Betrakta algoritmen som att exekveringen delas närhelst två element jämförs
- Trädet innehåller alla jämförelser längs alla tänkbara följder av instruktioner
- Körtiden för algoritmen = längden av stigen som traverseras
- Körtiden i värsta fall = höjden av trädet

21.40

Beslutsträdets höjd

- Höjden av beslutsträdet är en undre gräns för exekveringstiden i värsta fallet
- Varje tänkbar permutation av indata måste leda till ett separat utdatalöv
  - Om det inte vore så skulle något indata $\ldots 4 \ldots 5 \ldots$ ha samma utdataordning som $\ldots 5 \ldots 4 \ldots$ vilket skulle vara fel
- Eftersom det finns $n! = 1 \cdot 2 \cdot \ldots \cdot n$ löv är höjden av trädet minst $\log(n!)$

21.41

Den undre gränsen

- Varje jämförelsebaserad sorteringsalgoritm använder minst $\log(n!)$ tid i värsta fallet
- Därför använder en sådan algoritm minst tid

$$\log(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = (n/2)\log(n/2)$$

- Alltså är exekveringstiden för alla jämförelsebaserade sorteringsalgoritmer $\Omega(n \log n)$ i värsta fallet

21.42

3 Sortering i linjär tid?

Några fall då man kan sortera snabbare än $n \log n$

- Bara ett konstant antal *olika* element ska sorteras
  - $\Theta(n)$ med räknesortering
- Elementen som ska sorteras är tal som är jämnt fördelade i ett visst intervall
  - $\Theta(n)$ med bucket-sort
- Elementen som ska sorteras är strängar som består av d ”siffror” ($S[i] = s_{i,1}s_{i,2}\ldots s_{i,d}$)
  - $\Theta(nd)$ med radix-sort
  - Om d är konstant får vi linjär tidskomplexitet
  - Om vi räknar antalet siffror i indata får vi linjär tidskomplexitet $\Theta(N)$, där $N = nd$

21.43

3.1 Counting-sort

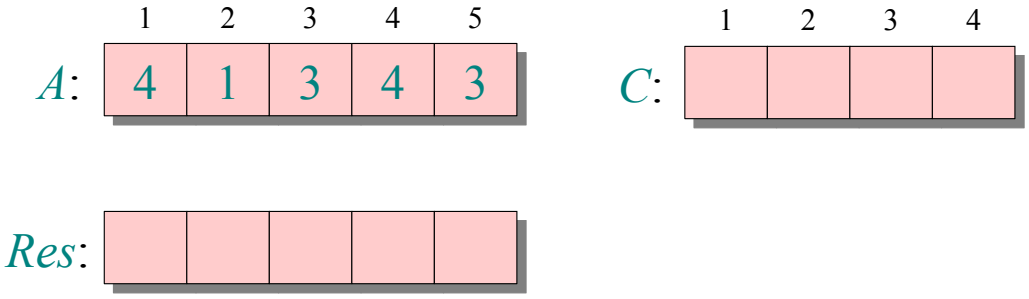
Räknesortering

Require: $A[1,\ldots,n]$, där $A[j] \in \{1,2,\ldots,k\}$
  function COUNTINGSORT(A)
    Hjälparray för räkning: $C[1,\ldots,k]$
    Hjälparray för lagring av resultatet: $Res[1,\ldots,n]$
    for $i \leftarrow 1$ **to** k **do**
      $C[i] \leftarrow 0$
    for $j \leftarrow 1$ **to** n **do**
      $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{nyckel = i\}|$
    for $i \leftarrow 2$ **to** k **do**
      $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{nyckel \leq i\}|$
    for $j \leftarrow n$ **downto** i **do**
      $Res[C[A[j]]] \leftarrow A[j]$
      $C[A[j]] \leftarrow C[A[j]] - 1$
    return Res

21.44

Exempel

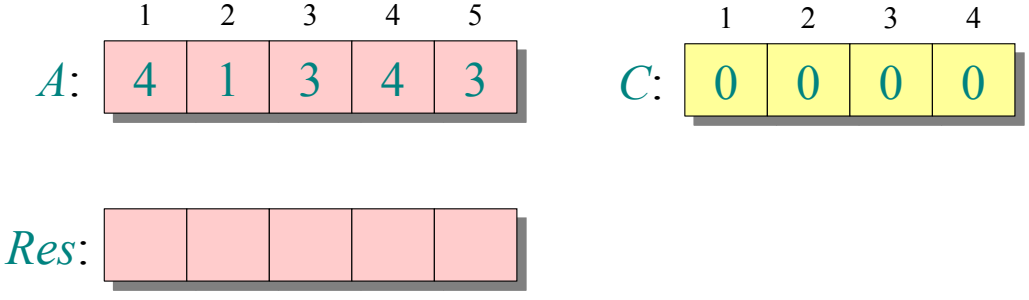
Counting-sort



21.45

Exempel

Loop 1

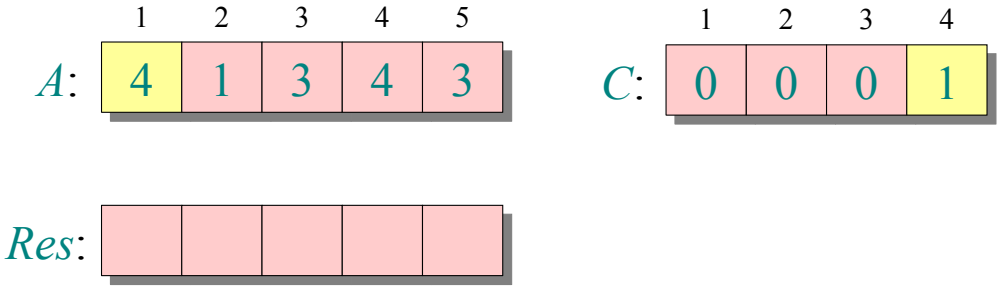


for $i \leftarrow 1$ **to** k **do**
  $C[i] \leftarrow 0$

21.46

Exempel

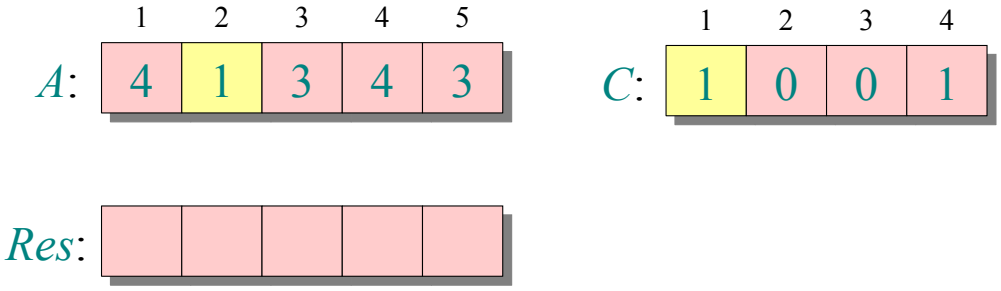
Loop 2



for $j \leftarrow 1$ **to** n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{nyckel} = i\}|$

Exempel

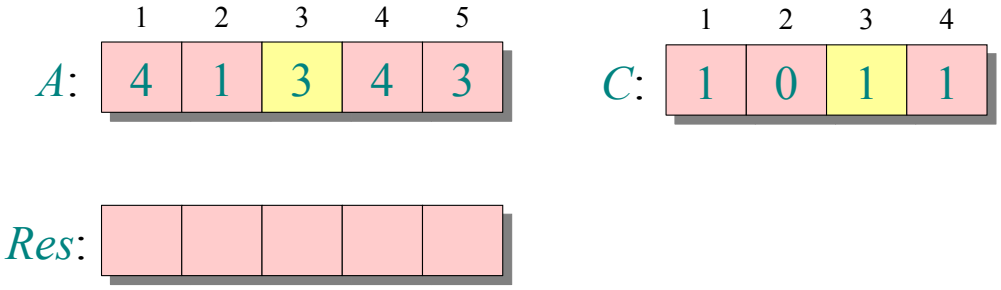
Loop 2



for $j \leftarrow 1$ **to** n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{nyckel} = i\}|$

Exempel

Loop 2

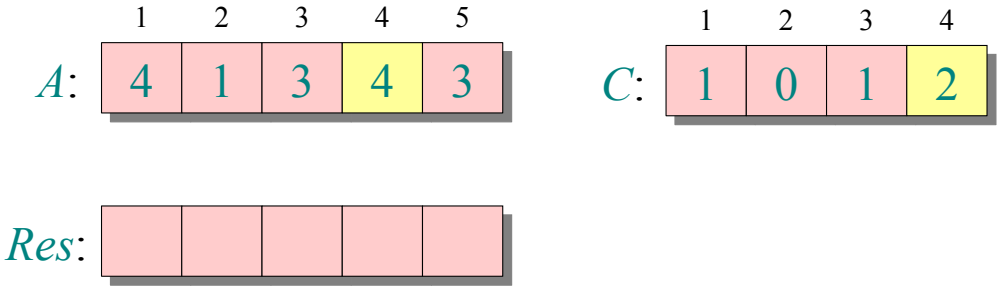


for $j \leftarrow 1$ **to** n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{nyckel} = i\}|$

21.49

Exempel

Loop 2

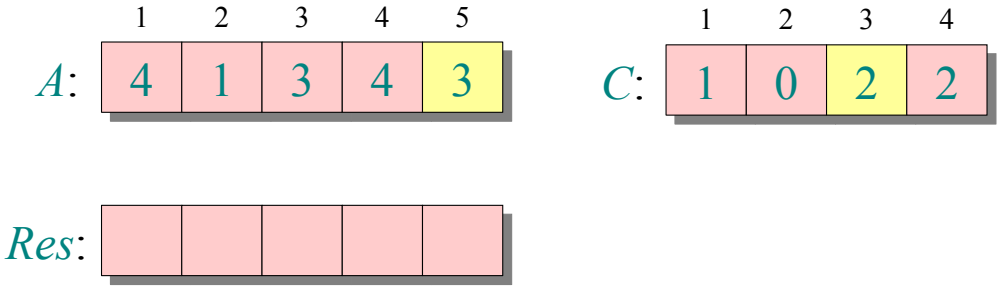


for $j \leftarrow 1$ **to** n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{nyckel} = i\}|$

21.50

Exempel

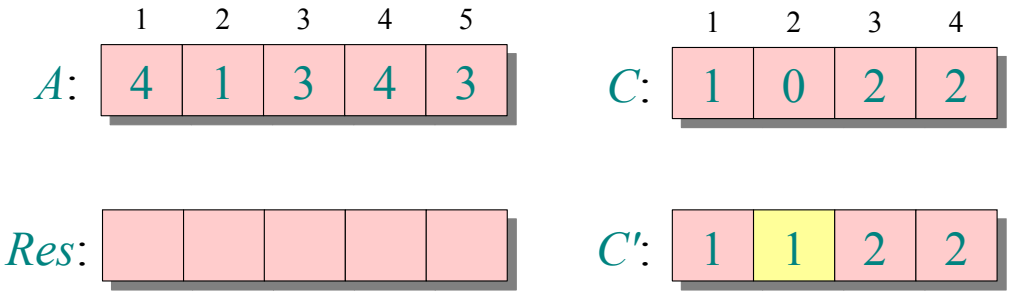
Loop 2



for $j \leftarrow 1$ **to** n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{nyckel} = i\}|$

Exempel

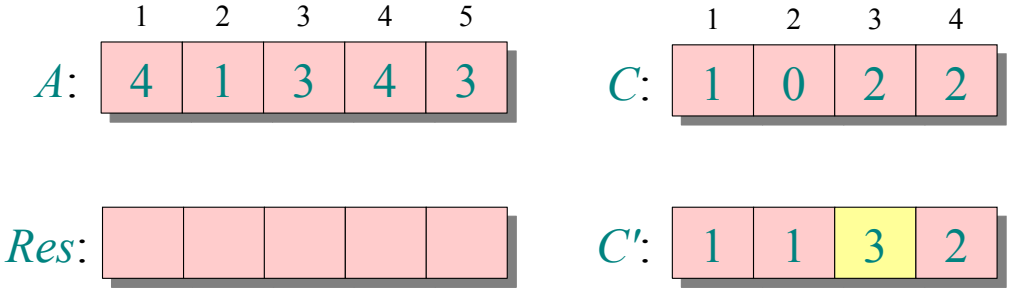
Loop 3



for $i \leftarrow 2$ **to** k **do**
  $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{nyckel} \leq i\}|$

Exempel

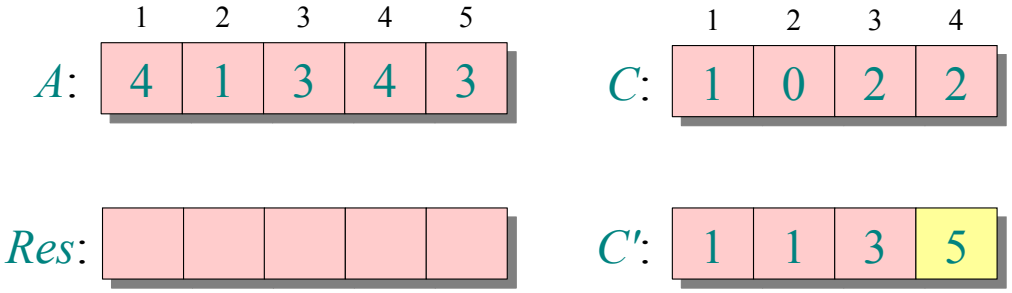
Loop 3



for $i \leftarrow 2$ **to** k **do**
  $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{nyckel} \leq i\}|$

Exempel

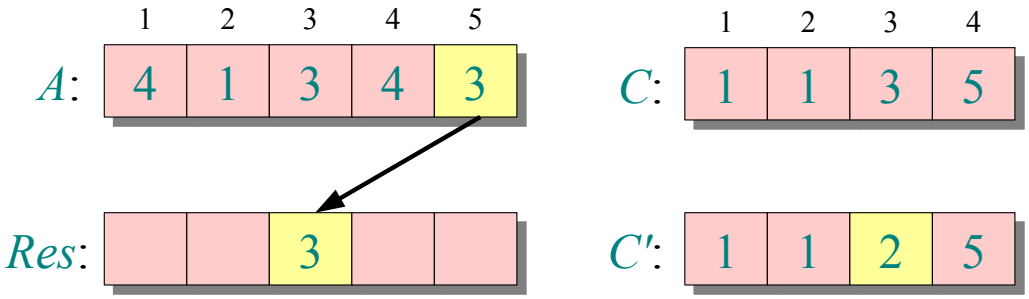
Loop 3



for $i \leftarrow 2$ **to** k **do**
  $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{nyckel} \leq i\}|$

Exempel

Loop 4

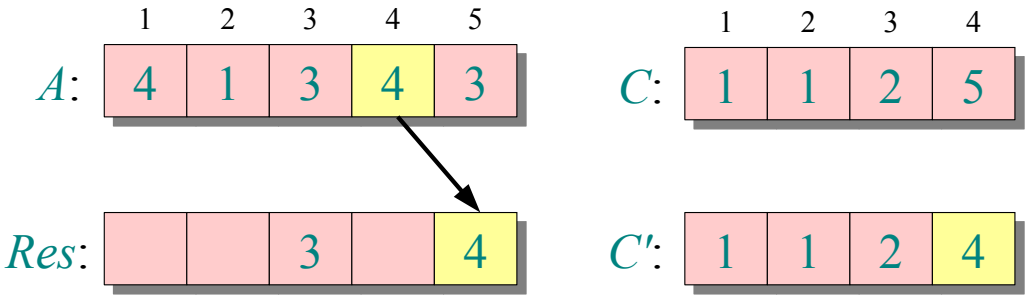


for $j \leftarrow n$ **downto** 1**do**
  $Res[C[A[j]]] \leftarrow A[j]$
  $C[A[j]] \leftarrow C[A[j]] - 1$

21.55

Exempel

Loop 4

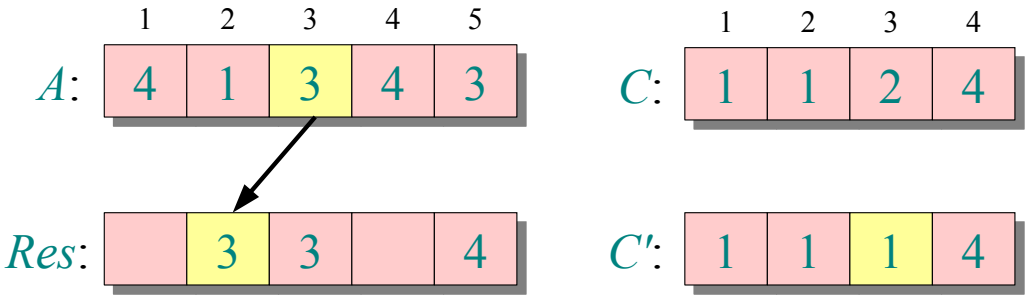


for $j \leftarrow n$ **downto** 1**do**
  $Res[C[A[j]]] \leftarrow A[j]$
  $C[A[j]] \leftarrow C[A[j]] - 1$

21.56

Exempel

Loop 4

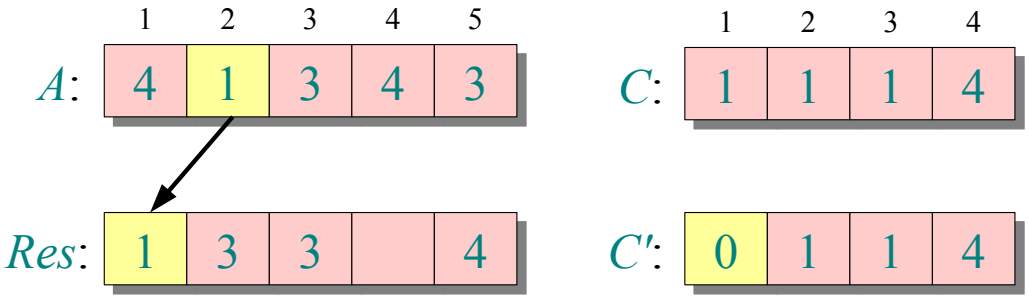


for $j \leftarrow n$ **downto** 1**do**
  $Res[C[A[j]]] \leftarrow A[j]$
  $C[A[j]] \leftarrow C[A[j]] - 1$

21.57

Exempel

Loop 4

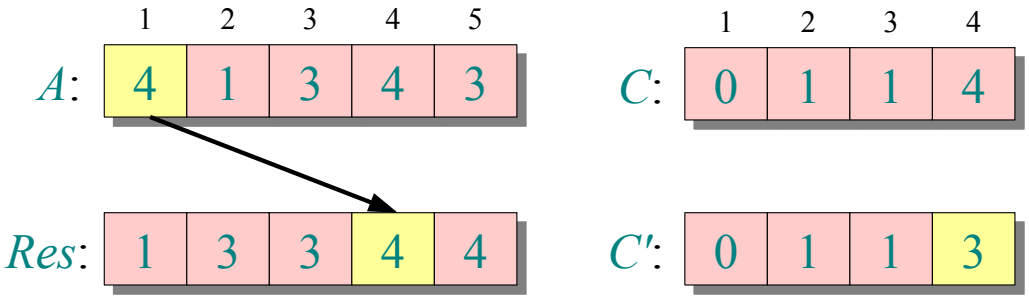


for $j \leftarrow n$ **downto** 1**do**
  $Res[C[A[j]]] \leftarrow A[j]$
  $C[A[j]] \leftarrow C[A[j]] - 1$

21.58

Exempel

Loop 4



for $j \leftarrow n$ **downto** 1**do**
  $Res[C[A[j]]] \leftarrow A[j]$
  $C[A[j]] \leftarrow C[A[j]] - 1$

Analys

$\Theta(k)$ **for** $i \leftarrow 1$ **to** k **do**
  $C[i] \leftarrow 0$

$\Theta(n)$ **for** $j \leftarrow 1$ **to** n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1$

$\Theta(k)$ **for** $i \leftarrow 2$ **to** k **do**
  $C[i] \leftarrow C[i] + C[i-1]$

$\Theta(n)$ **for** $j \leftarrow n$ **downto** 1 **do**
  $Res[C[A[j]]] \leftarrow A[j]$
  $C[A[j]] \leftarrow C[A[j]] - 1$

$\Theta(n + k)$

Exekveringstid

Om $k \in O(n)$ tar räknesortering $\Theta(n)$ tid

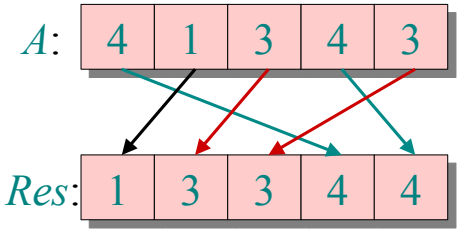
- Men sortering tar ju $\Omega(n \log n)$ tid!
- Vad är det som inte stämmer?

Svar

- *Jämförelsebaserad sortering* tar $\Omega(n \log n)$ tid
- Counting-sort är *inte* jämförelsebaserad
- Faktum är att inte en enda jämförelse mellan några element utförs!

Stabil sortering

Counting-sort är en stabil sorteringsmetod: den bevarar indataordningen mellan lika element



Att fundera på:
Vilka andra metoder för sortering är stabila?

21.62

3.2 Bucket-sort

Bucket-sort

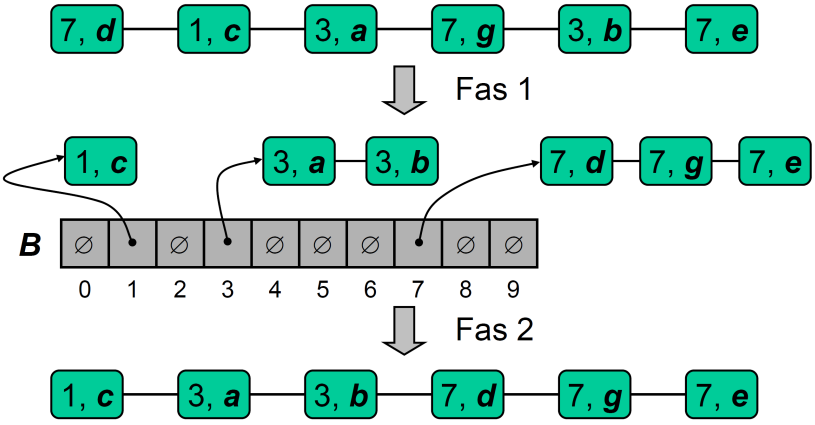
- Låt S vara en sekvens av n poster (nyckel, värde) med nycklar från $[0, N-1]$
- Bucket-sort använder nycklarna som index i en hjälparray B av sekvenser
  - Fas 1: Töm sekvensen S genom att flytta varje post (k, v) sist i sin bucket $B[k]$
  - Fas 2: För $i = 0, \ldots, N-1$ flytta posterna i bucket $B[i]$ till slutet av sekvensen S
- Analys:
  - Fas 1 tar $O(n)$ tid
  - Fas 2 tar $O(n+N)$ tid

  Bucket-sort tar $O(n+N)$ tid

```
procedure BUCKETSORT(S, N)
    B ← array med N tomma sekvenser
    while ¬S.ISEMPTY() do
        f ← S.FIRST()
        (k, o) ← S.REMOVE(f)
        B[k].INSERTLAST((k, o))
    for i ← 0 to N − 1 do
        while ¬B[i].ISEMPTY() do
            f ← B[i].FIRST()
            (k, o) ← B[i].REMOVE(f)
            S.INSERTLAST((k, o))
```

21.63

Exempel: Nycklar från $[0,9]$



21.64

Egenskaper och utökningar

Nycklarnas typ

- Nycklarna används som index i en array och kan inte vara objekt av godtycklig typ

Stabil sortering

- Den relativa ordningen av alla par av data med samma nycklar är bevarad efter exekveringen av algoritmen

Utökningar

- Heltalsnycklar från $[a,b]$
  - Sätt in (k,v) i bucket $B[k-a]$
- Strängnycklar från en ändlig mängd strängar D
  - Sortera D och beräkna rangen $r(k)$ för varje sträng $k \in D$ i den sorterade sekvensen
  - Sätt in (k,v) i bucket $B[r(k)]$

21.65

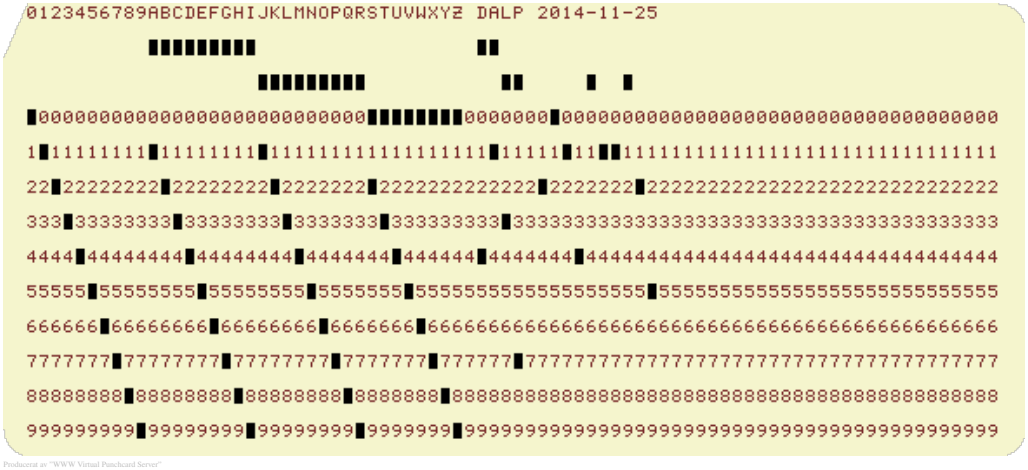
3.3 Radix-sort

Radix-sort

- Ursprung: Herman Holleriths kortsorteringsmaskin för folkräkningen 1890 i USA
- Siffra-för-siffrasortering
- Holleriths (dåliga) ursprungsidé: sortera på mest signifikant siffra först
- Bra idé: sortera på *minst signifikant siffra först* med en extern *stabil* sorteringsrutin

21.66

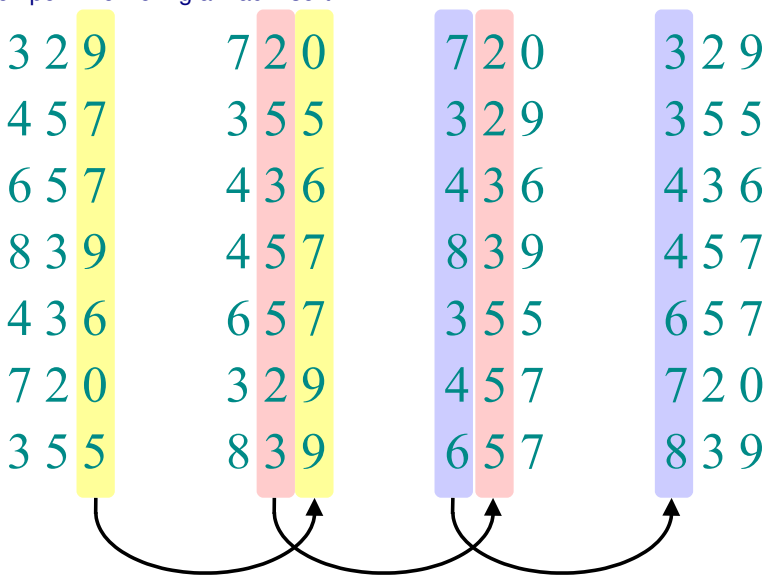
”Modernt” hålkort från IBM



Så det är därför skalfönster har 80 kolumner!

21.67

Exempel: Exekvering av radix-sort

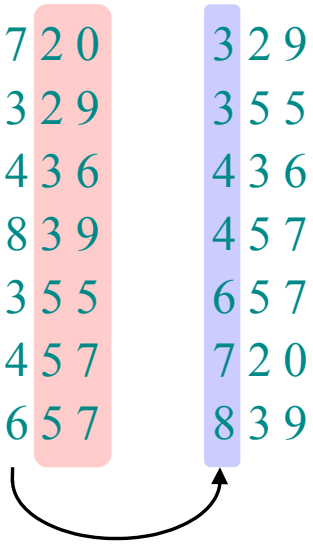


21.68

Korrekthet för radix-sort

Använd induktion över sifferposition

- Antag att talen är sorterade på sina $t-1$ lägsta siffror
- Sortera på siffra t

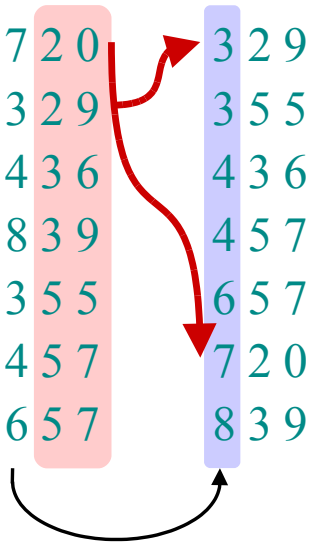


21.69

Korrekthet för radix-sort

Använd induktion över sifferposition

- Antag att talen är sorterade på sina $t-1$ lägsta siffror
- Sortera på siffra t
  - Två tal som skiljer sig i siffra t blir korrekt sorterade

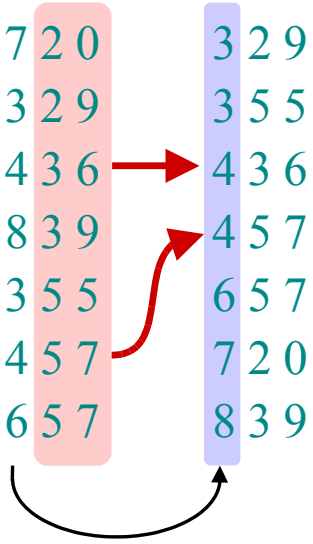


21.70

Korrekthet för radix-sort

Använd induktion över sifferposition

- Antag att talen är sorterade på sina $t-1$ lägsta siffror
- Sortera på siffra t
  - Två tal som skiljer sig i siffra t blir korrekt sorterade
  - Två tal som är lika i siffra t får samma ordning som i indata $\Rightarrow$ rätt ordning

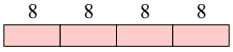


21.71

Analys av radix-sort

- Antag att counting-sort används som extern sorteringsrutin
- Sortera n maskinord om b bitar vardera
- Vi kan se det som att varje ord har b/r siffror i bas 2^r

Exempel:

32-bitars ord 

$r = 8 \Rightarrow b/r = 4$ pass av counting-sort på siffror i bas 2^8
eller $r = 16 \Rightarrow b/r = 2$ pass av counting-sort på siffror i bas 2^{16}

Hur många pass bör vi göra?

21.72

Analys av radix-sort

Kom ihåg: counting-sort tar tid $\Theta(n+k)$ för att sortera n tal från $[0, k-1]$. Om varje b-bitars ord bryts upp i r-bitars bitar tar varje pass av counting-sort $\Theta(n+2^r)$ tid. Eftersom det blir b/r pass får vi

$$T(n,b) = \Theta\left(\frac{b}{r}(n+2^r)\right)$$

Välj r för att minimera $T(n,b)$

- Att öka r ger färre pass, men när $r \gg \log n$ ökar tiden exponentiellt.

21.73

Att välja r

$$T(n,b) = \Theta\left(\frac{b}{r}(n+2^r)\right)$$

Minimera $T(n,b)$ genom att derivera och sätta till 0. Eller, observera att vi inte vill ha $2^r \gg n$ och att det inte skadar asymptotiskt att välja r så stort som möjligt givet detta villkor. Valet $r = \log n$ medför $T(n,b) = \Theta(bn/\log n)$.

- För tal i intervallet från 0 till $n^d - 1$ har vi $b = d \log n \Rightarrow$ radix-sort kör i tid $\Theta(dn)$.

21.74

Slutsatser

I praktiken är radix-sort snabb för stora indata, samt enkel att koda och underhålla.

Exempel: 32-bitars tal

- Som mest 3 pass när man sorterar ≥ 2000 tal.
- Merge-sort och quick-sort använder minst $\lceil \log 2000 \rceil = 11$ pass.

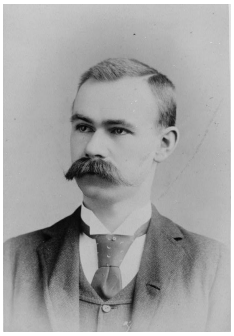
Nackdelar: Det går inte att sortera in-place med räknesortering. Det är också så att radix-sort inte har bra datalokalitet (vilket man kan se till att quick-sort har) vilket gör att en väl trimmad implementation av quick-sort kan vara snabbare på en modern processor med brant minneshierarki.

21.75

4 Hålkortsteknologi

Herman Hollerith (1860-1929)

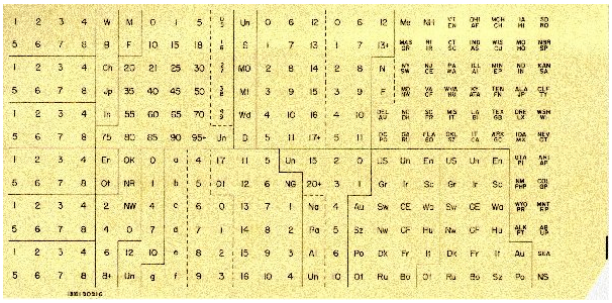
- 1880 års folkräkning i USA tog nästan 10 år att bearbeta.
- Under tiden arbetade en föreläsare vid MIT fram hålkortsteknologin.
- Hans maskiner, inklusive en ”kortsorterare”, gjorde att folkräkningen 1890 kunde slutföras på 6 veckor.
- Han grundade ”The Tabulating Machine Company” 1911, vilket gick samman med andra företag 1924 för att bilda ”International Business Machines”



21.76

Hålkort

- Hålkort = datapost
- Hål = värde
- Algoritm = maskin + människa



Replika av hålkort från folkräkningen 1900 [Howells 2000]

21.77

Holleriths tabuleringssytem

- Pantografisk hålmaskin
- Läsare manövrerad med handkraft
- Visare för räkneresultat
- Sorteringlåda

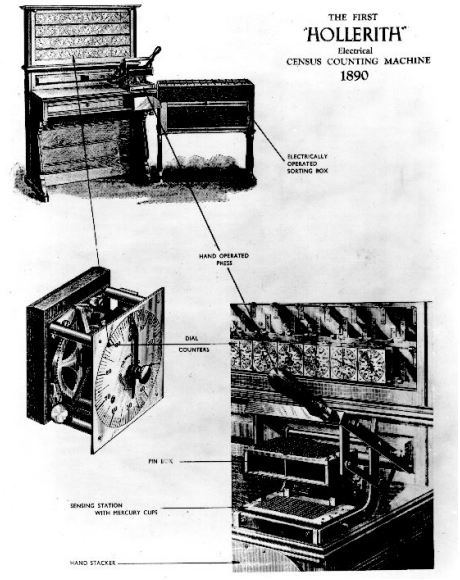


Bild från [Howells 2000]

21.78

Ursprunget till radix-sort

Holleriths patentansökan från 1899 talar om radix-sort med början från den mest signifikanta siffran:

> The most complicated combinations can readily be counted with comparatively few counters or relays by first assorting the cards according to the first items entering into the combinations, then reassorting each group according to the second item entering into the combination, and so on, and finally counting on a few counters the last item of the combination for each group of cards.

Att köra radix-sort med minst signifikant siffra först verkar vara en uppfinning av maskinoperatörerna. 21.79

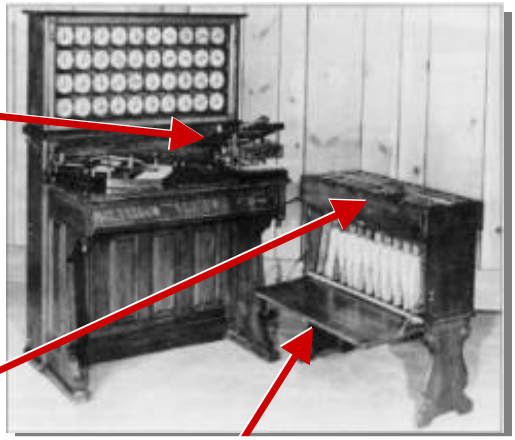
Sorteringsmaskinen

En operatör sätter in ett kort i pressen.

Stift på pressen når genom hålen i hålkortet och får kontakt med kvicksilverfyllda bägare under kortet.

När ett visst siffervärde slås in lyfts locket på motsvarande sorteringsfack.

Operatören lägger kortet i facket och stänger locket.



Holleriths Tabulator, Pantograph, Press och Sorter

När alla kort bearbetats öppnas frontpanelen och korten samlas in i ordning, vilket ger ett pass av en stabil sorteringsrutin. 21.80