Föreläsning 19

Heap-sort, merge-sort. Lower limit for sorting. Sorting in linear time?

TDDD86: DALP

Utskriftsversion av Föreläsing i *Datastrukturer, algoritmer och programmeringsparadigm*
27 november 2023

IDA, Linköpings universitet

19.1

Content

Contents

1 Sorting **1**
1.1 Heap-sort . . . 1
1.2 Merge-sort . . . 5
1.3 Summary . . . 10

2 A lower limit for comparison based sorting **11**

3 Sorting in linear time? **13**
3.1 Counting-sort . . . 14
3.2 Bucket-sort . . . 22
3.3 Radix-sort . . . 23

19.2

1 Sorting

1.1 Heap-sort

Sorting with a priority queue

- Use a priority queue to sort a number of comparable elements
  - Insert the elements in the priority queue
  - Remove the elements in a sorted order using removeMin-operations
- Execution time depends on the priority queue implementation:
  - Unsorted sequence corresponds to a selection sort and an $O(n^2)$ time
  - Sorted sequence gives insertion sort and an $O(n^2)$ time
- Can we achieve better?

procedure PQSORT(S)
  $P \leftarrow$ empty priority queue
  while $\neg S$.ISEMPTY() **do**
    $e \leftarrow S$.REMOVE(S.FIRST())
    P.INSERT(e)
  while $\neg P$.ISEMPTY() **do**
    $e \leftarrow P$.REMOVEMIN()
    S.INSERTLAST(e)

19.3

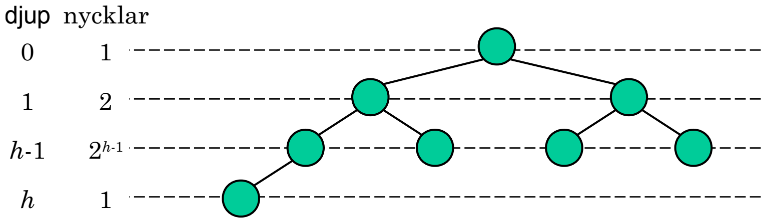
Height of a heap

Proposition 1. A heap with n keys has height $O(\log n)$

Proof. The heap is a represented with a complete tree.

- Let h be the height of a heap with n keys
- There are 2^i keys at depths $i = 0, \ldots h-1$ and at least a key at depth h. Therefore, $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
- Hence, $n \geq 2^h$ and $h \leq log_2 n$

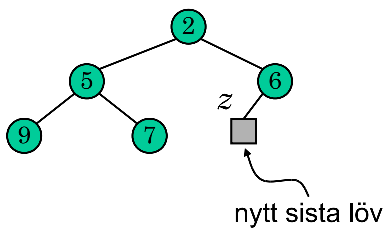
□

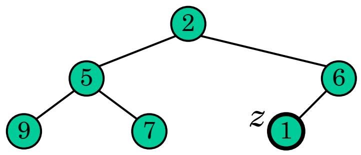


19.4

Insertion in a heap

- Method insert in ADT priority queue inserts key k in the heap
- Insertion algorithm involves three steps:
  - Find location for inserting node z (new last leaf)
  - Store k in z
  - Restore heap property

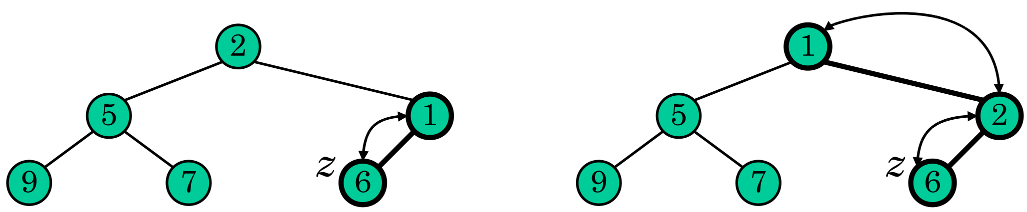




19.5

Upheap (bubble up)

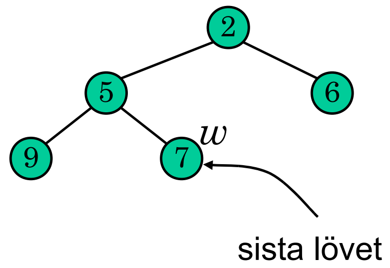
- Insertion of a key k might violate the heap property
- Method upheap restores the heap property by moving the key k upwards along the path to the root
- upheap terminates when key k reaches the root or a node whose parent is not larger than k
- Since the height of the heap is $O(\log n)$, the upheap method is in $O(\log n)$ time

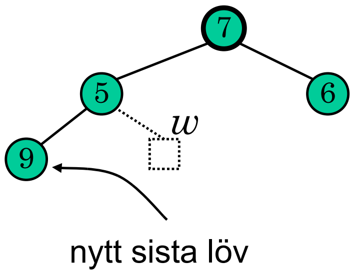


19.6

Removal from a heap

- Method removeMin in ADT priority queue removes the root key from the heap
- Removal algorithm consists in 3 steps:
  - Replace root key with the key from the last leaf w
  - Remove w
  - Restore heap property





19.7

Downheap (bubble down)

- Replacing root key with key k from last leaf might violate the heap property
- Method downheap restores the heap property by moving k downwards
- downheap terminate when key k reaches a leaf or a node where none of the children has a key smaller than k
- Since the height of the tree is $O(\log n)$, the downheap method is in $O(\log n)$ time

19.8

Heap-sort

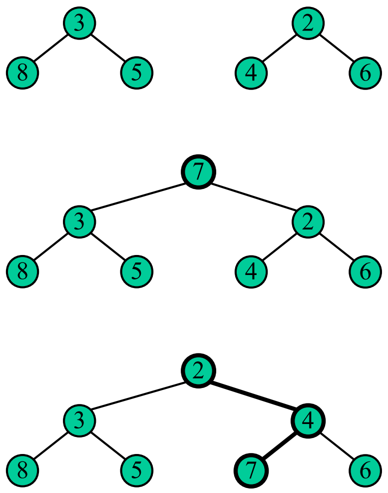
- Consider a priority queue with n elements implemented with a heap. For each one of the n elements:
  - insert and removeMin take $O(\log n)$ time
  - size, isEmpty and min take $O(1)$ time

- With a heap based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is faster than a quadratic sorting algorithm.

19.9

Merging two heaps

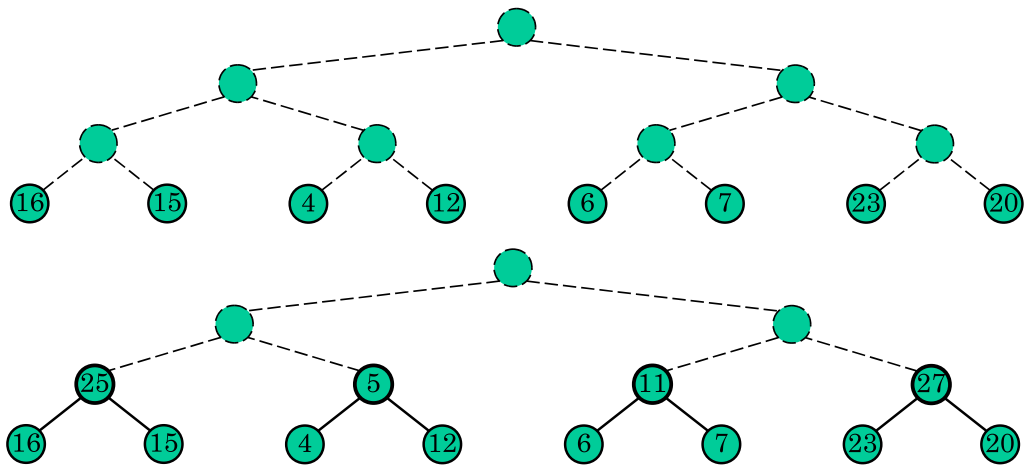
- Given two heaps and a key k
- Create a new heap where the root node stores key k with the two heaps as sub-trees
- Run downheap to restore the heap property



19.10

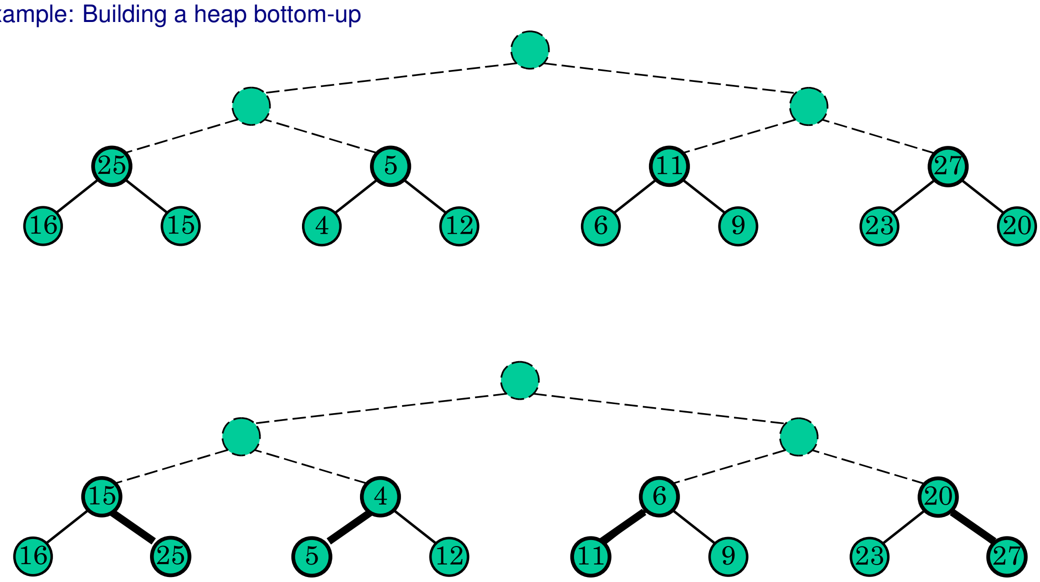
Example: Building a heap bottom-up

10	7	8	25	5	11	27	16	15	4	12	6	7	23	20



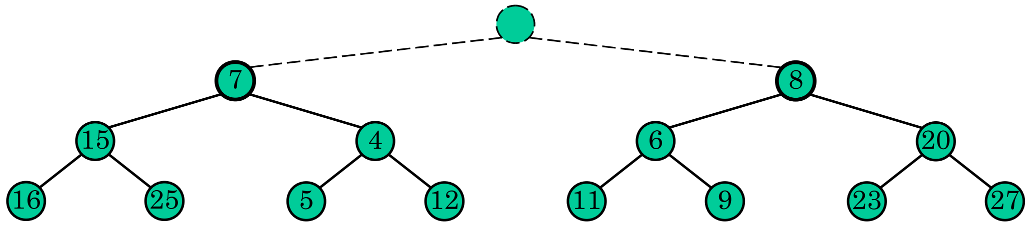
19.11

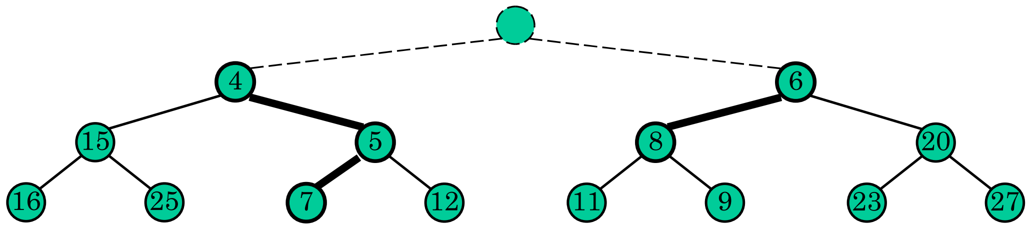
Example: Building a heap bottom-up



19.12

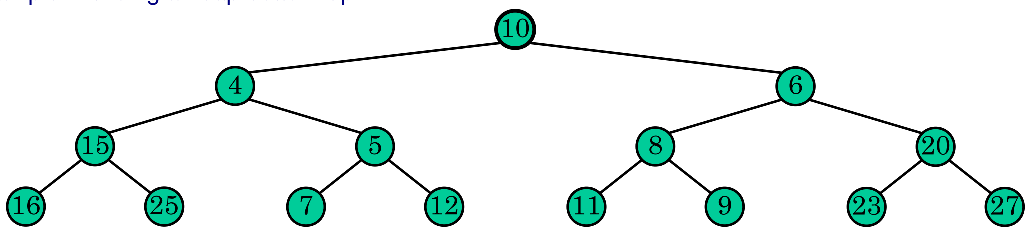
Example: Building a heap bottom-up

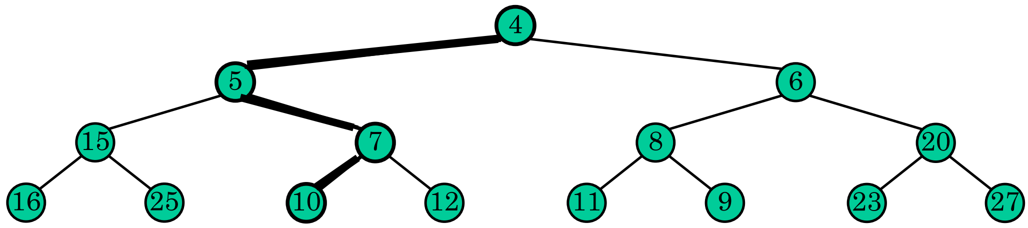




19.13

Example: Building a heap bottom-up

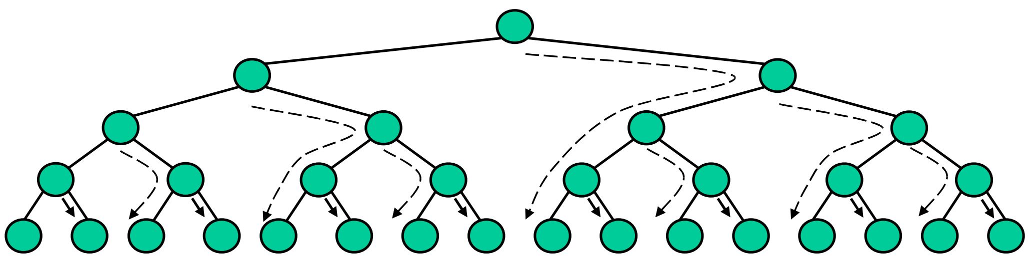




19.14

Analysis

- We visualize a worst-case calls to downheap with paths that start right then continue left until the heap bottom.
- Since each node is traversed at most twice, the total number of such paths is $O(n)$
- Hence building the heap bottom-up requires at most $O(n)$ steps
- This is faster than n calls to insert in the first phase of heap-sort



19.15

1.2 Merge-sort

Back to divide-and-conquer

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Similar to heap-sort:
  - has an execution time in $O(n \log n)$

- Unlike heap-sort
  - does not use a priority queue
  - accesses data in a sequential fashion (adapted for sorting data on disk)

19.16

Merge-sort

Merge-sort on an input sequence S with n elements consists in 3 steps:

- Divide: partition S in two sequences S_1 and S_2, each with $n/2$ elements
- Conquer: sort S_1 and S_2 recursively
- Combine: merge S_1 and S_2 into a sorted sequence

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

19.17

Merge two sorted sequences

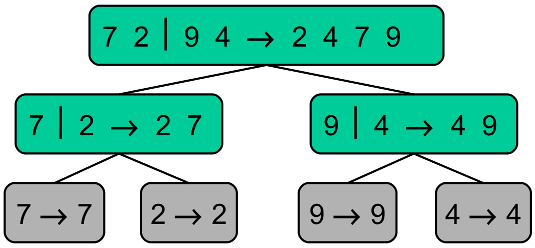
- Combination step: merge two sequences A and B into a sorted sequence S containing the union of elements in A and B
- Merging two sorted sequences, each with $n/2$ elements implemented with doubly linked lists takes $O(n)$ time

```
function MERGE(A, B)
    S ← empty sequence
    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
```

19.18

Merge-sort tree

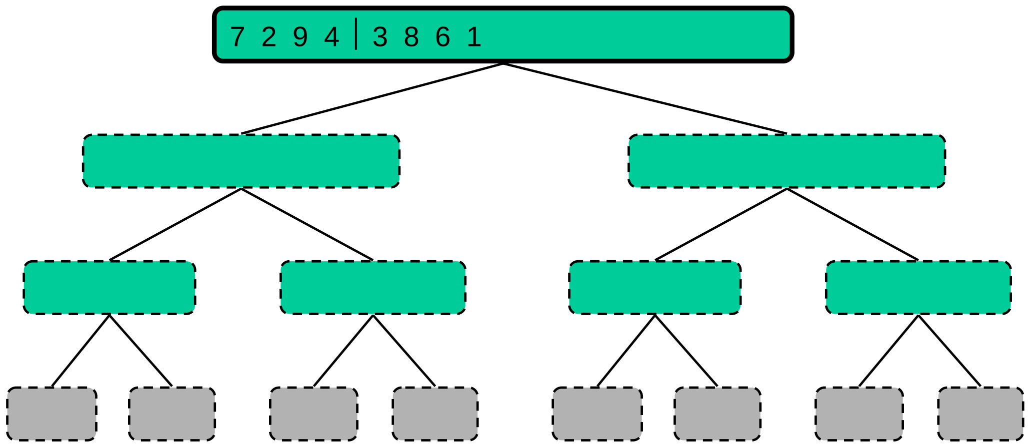
- Execution of merge-sort can be visualized with a binary tree
  - Each node represents a recursive call to merge sort and represents
    - Unsorted sequence before execution and its partition
    - Sorted sequence after execution
  - Root is the original call
  - Leaves are calls on sequences with lengths 0 or 1



19.19

Example: Execution of merge-sort

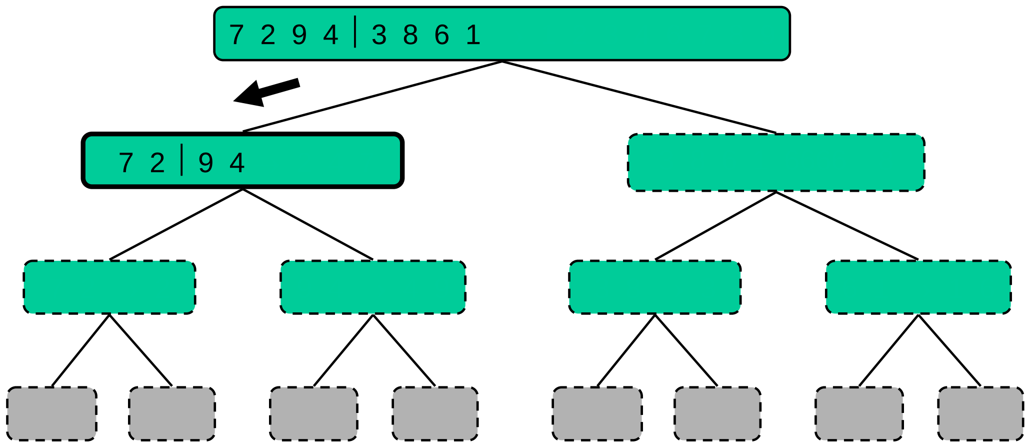
- Partition



19.20

Example: Execution of merge-sort

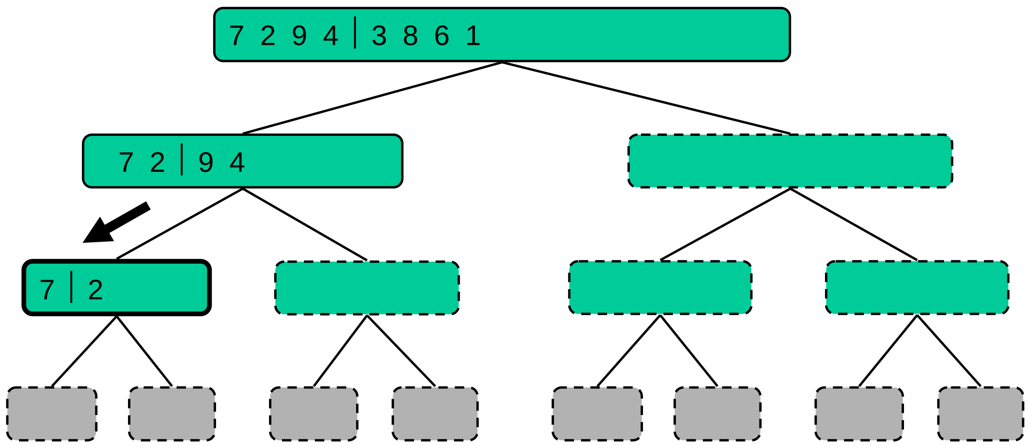
- recursive call, partition



19.21

Example: Execution of merge-sort

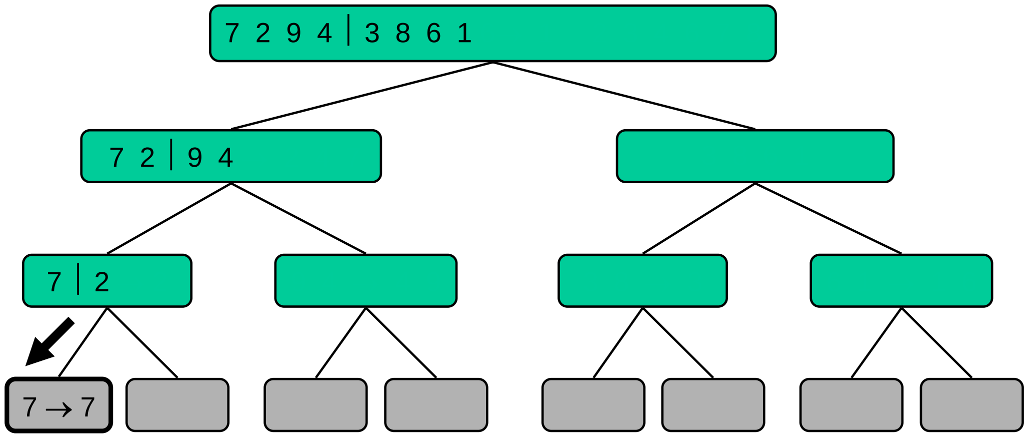
- recursive call, partition



19.22

Example: Execution of merge-sort

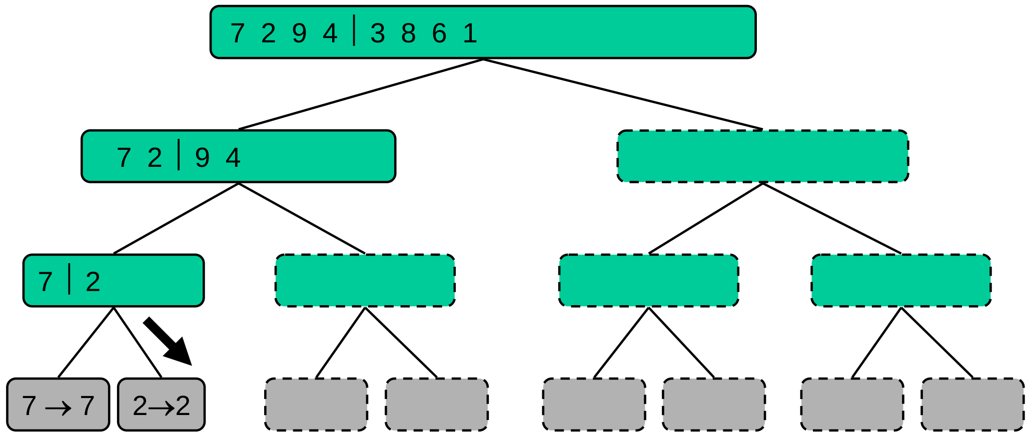
- recursive call, base case



19.23

Example: Execution of merge-sort

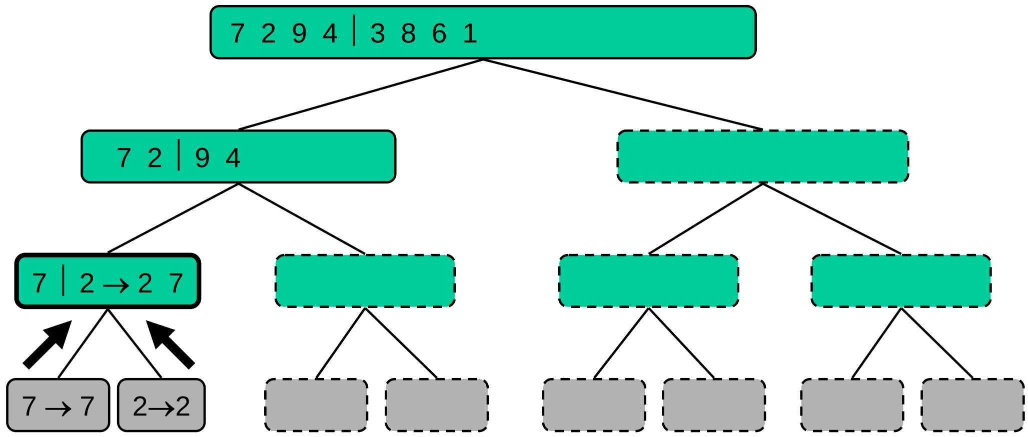
- Recursive call, base case



19.24

Example: Execution of merge-sort

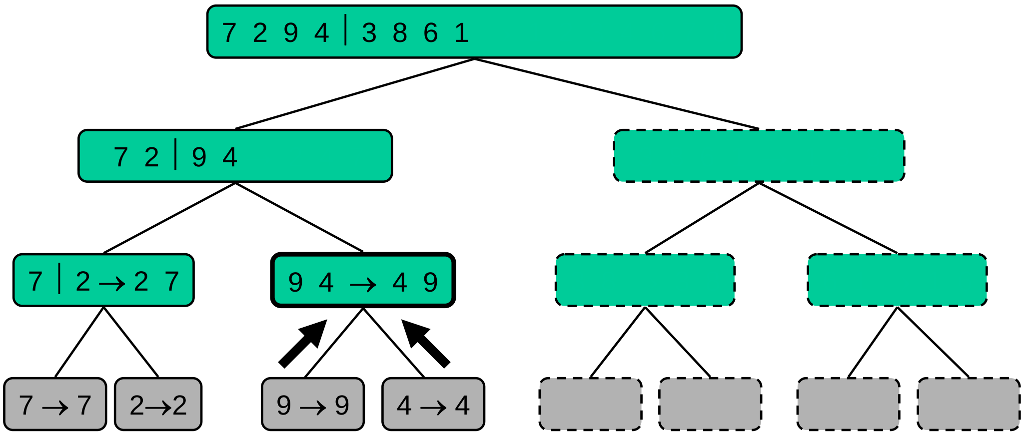
- merge



19.25

Example: Execution of merge-sort

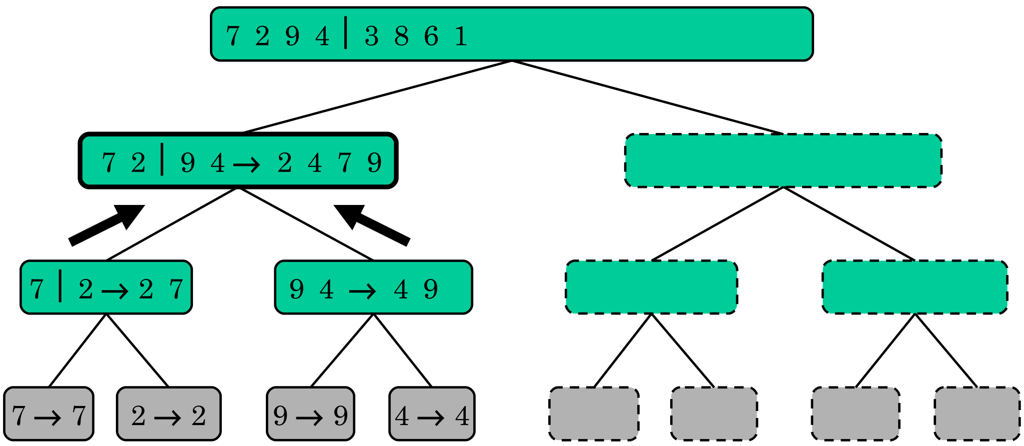
- recursive call, . . . , base case



19.26

Example: Execution of merge-sort

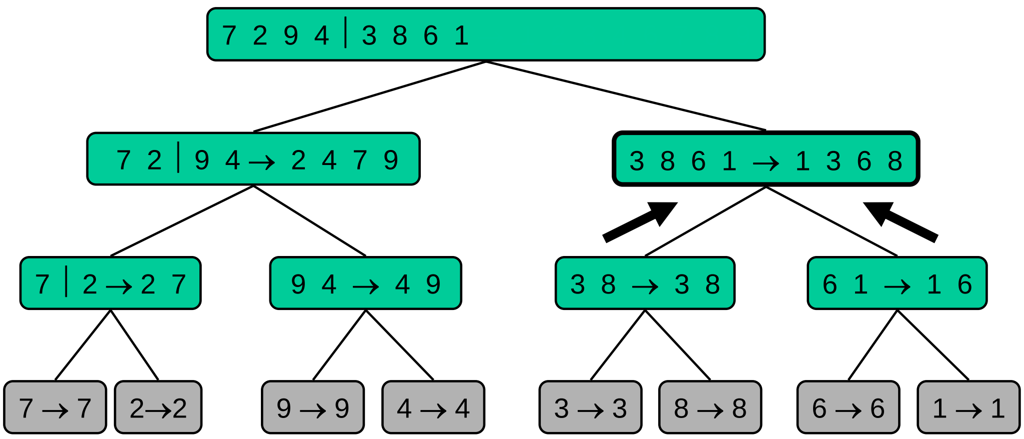
- Merge



19.27

Example: Execution of merge-sort

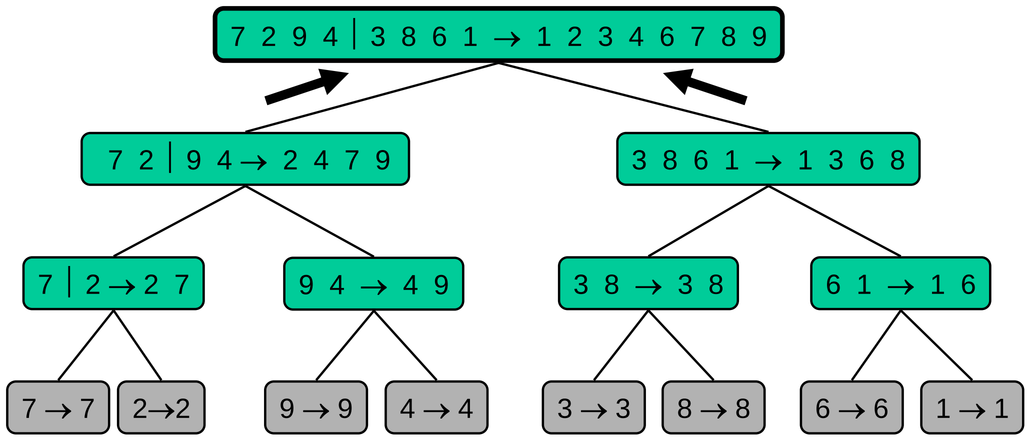
- Recursive call, . . . , merge



19.28

Example: Execution of merge-sort

- Merge



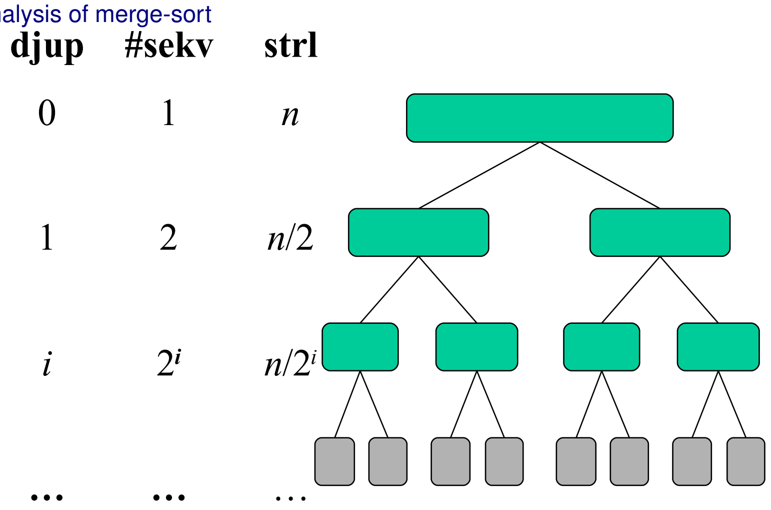
19.29

Analysis of merge-sort

- Height h of merge-sort tree is $O(\log n)$
  - at each recursive call, the sequence is divided in the middle
- The total amount of work performed at depth i is $O(n)$
  - we partition and merge 2^i sequences of lengths $n/2^i$
  - we perform 2^{i+1} recursive calls
- The total execution time for merge-sort is $O(n \log n)$

19.30

Analysis of merge-sort



19.31

1.3 Summary

Summary so far

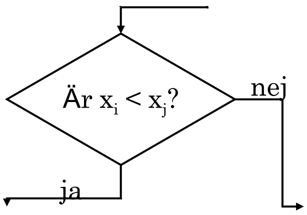
Algoritm	Tid	Noteringar
selection-sort	$O(n^2)$	• in-place • långsam (bra för små indata)
insertion-sort	$O(n^2)$	• in-place • långsam (bra för små indata)
quick-sort	$O(n \log n)$ förväntad	• in-place, randomiserad • snabbast (bra för stora indata)
heap-sort	$O(n \log n)$	• in-place • snabb (bra för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

19.32

2 A lower limit for comparison based sorting

Comparison based sorting

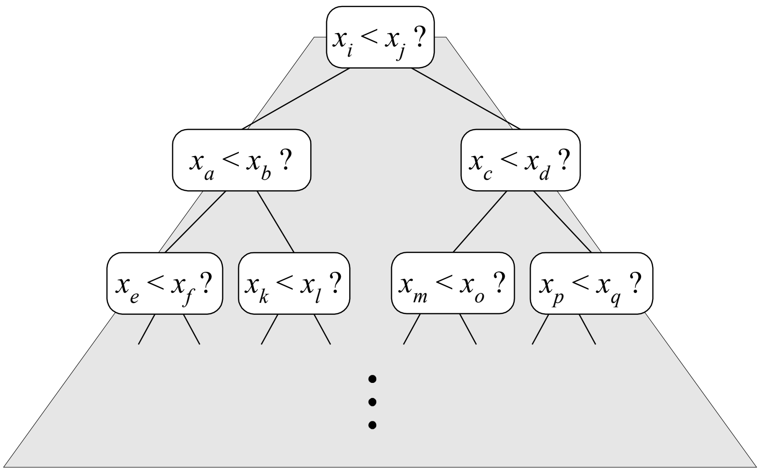
- Many sorting algorithms are *comparison based*
  - They sort by comparing pairs of elements
  - Example: insertion-sort, selection-sort, heap-sort, merge-sort, quick-sort, ...
- Let's deduce a lower limit for the worst-case execution time of any comparison-based algorithm that sorts a sequence of n elements $x_1, x_2, \ldots, x_n$



19.33

Count comparisons

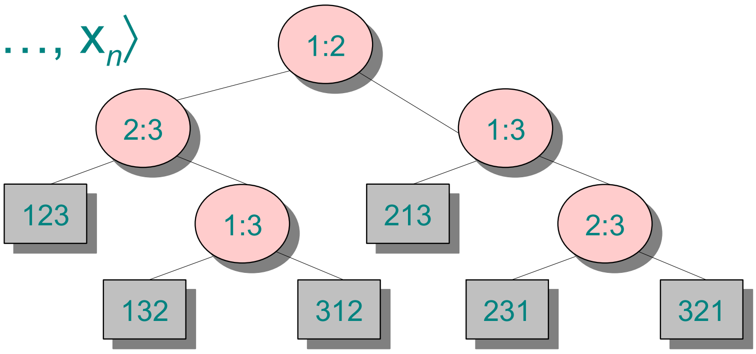
- Let us just count the number of comparisons
- Each execution of the algorithm corresponds to a path from the root to a leaf in a *decision tree*



19.34

Example: Decision tree

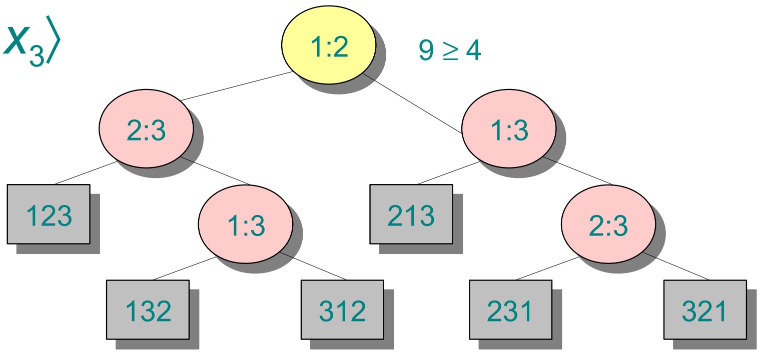
Sort $\langle x_1, x_2, \ldots, x_n \rangle$



Each node is marked with indices $i:j$ for $i, j \in \{1, 2, \ldots, n\}$
- Left sub-tree shows remaining comparisons if $x_i \leq x_j$
- Right sub-tree shows remaining comparisons if $x_i > x_j$

Example: Decision tree

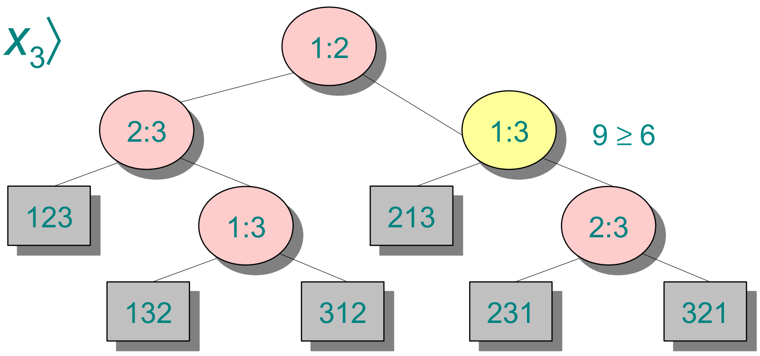
Sort $\langle x_1, x_2, x_3 \rangle$
$= \langle\ 9, 4, 6\ \rangle$:



Each node is marked with indices $i:j$ for $i, j \in \{1, 2, \ldots, n\}$
- Left sub-tree shows remaining comparisons if $x_i \leq x_j$
- Right sub-tree shows remaining comparisons if $x_i > x_j$

Example: Decision tree

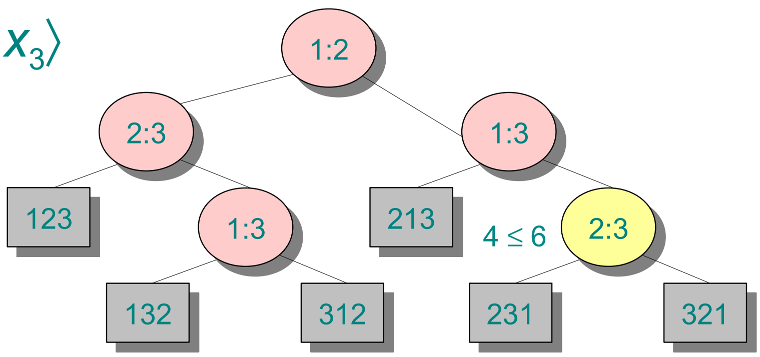
Sort $\langle x_1, x_2, x_3 \rangle$
$= \langle\ 9, 4, 6\ \rangle$:



Each node is marked with indices $i:j$ for $i, j \in \{1, 2, \ldots, n\}$
- Left sub-tree shows remaining comparisons if $x_i \leq x_j$
- Right sub-tree shows remaining comparisons if $x_i > x_j$

Example: Decision tree

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



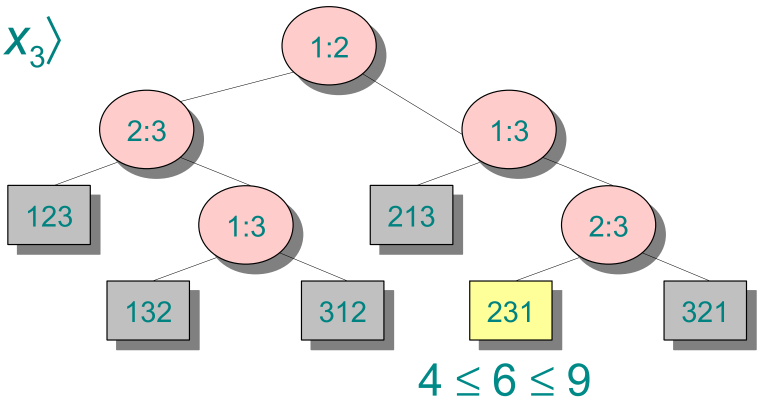
Each node is marked with indices $i:j$ for $i, j \in \{1, 2, \ldots, n\}$

- Left sub-tree shows remaining comparisons if $x_i \leq x_j$
- Right sub-tree shows remaining comparisons if $x_i > x_j$ 19.38

Example: Decision tree

Sort $\langle x_1, x_2, x_3 \rangle$

$= \langle 9, 4, 6 \rangle$:



Each leaf corresponds to a permutation $\langle \pi(i), \pi(2), \ldots, \pi(n) \rangle$ to indicate that $x_{\pi(1)} \leq x_{\pi(2)} \leq \ldots \leq x_{\pi(n)}$ was established 19.39

Decision tree model

Decision trees can model executions of any comparison based sorting algorithm:

- A tree for each input size
- Consider that execution is forked in two each time two elements are compared
- Tree contains all comparisons along all possible executions
- Execution time for the algorithm = length of the path to be traversed
- Execution time in worst case = height of the tree 19.40

Height of decision tree

- Height of decision tree is a lower limit to the worst case execution time
- Each possible permutation of input data need to result in a separate output leaf
  - Otherwise, some input sequence $\ldots 4 \ldots 5 \ldots$ would result in the same output as $\ldots 5 \ldots 4 \ldots$, which would be wrong
- Since there are $n! = 1 \cdot 2 \cdot \ldots \cdot n$ leaves, the height of the tree is at least $\log(n!)$ 19.41

Lower limit

- Each comparison based sorting algorithm uses at least $\log(n!)$ steps in the worst case
- Such an algorithm would therefore use at least

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

- The worst-case execution time of any comparison based sorting algorithm is therefore in $\Omega(n \log n)$ 19.42

3 Sorting in linear time?

Some cases where sorting can be faster than $n \log n$

- Only a constant number of *different* elements to sort
  - $\Theta(n)$ with Counting sort
- The elements to be sorted are uniformly distributed in a given interval
  - $\Theta(n)$ with bucket-sort
- Elements to be sorted are strings with d ”digits” ($S[i] = s_{i,1}s_{i,2}\ldots s_{i,d}$)
  - $\Theta(nd)$ with radix-sort
  - If d is constant we get linear time complexity
  - If we count the number of digits in the input sequence, we get a linear time complexity $\Theta(N)$, with $N = nd$ 19.43

3.1 Counting-sort

Counting sort

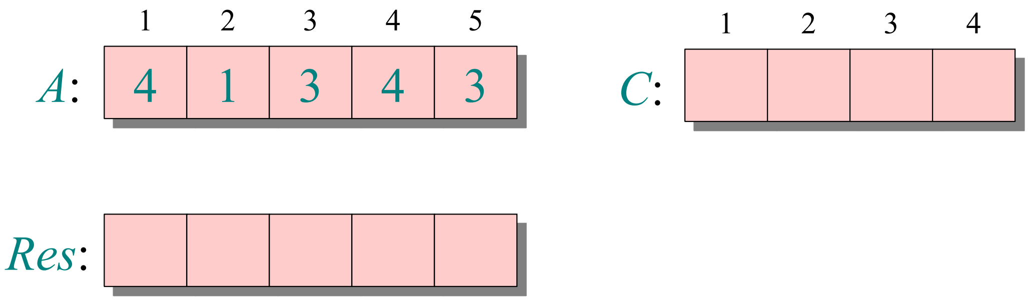
Require: $A[1,\ldots,n]$, with $A[j] \in \{1,2,\ldots,k\}$

  function COUNTINGSORT(A)
    an array for counting: $C[1,\ldots,k]$
    an array for storing the result: $Res[1,\ldots,n]$
    for $i \leftarrow 1$ **to** k **do**
      $C[i] \leftarrow 0$
    for $j \leftarrow 1$ **to** n **do**
      $C[A[j]] \leftarrow C[A[j]]+1$ $\triangleright\ C[i] = |\{key = i\}|$
    for $i \leftarrow 2$ **to** k **do**
      $C[i] \leftarrow C[i]+C[i-1]$ $\triangleright\ C[i] = |\{key \leq i\}|$
    for $j \leftarrow n$ **downto** i **do**
      $Res[C[A[j]]] \leftarrow A[j]$
      $C[A[j]] \leftarrow C[A[j]]-1$
    return Res

19.44

Example

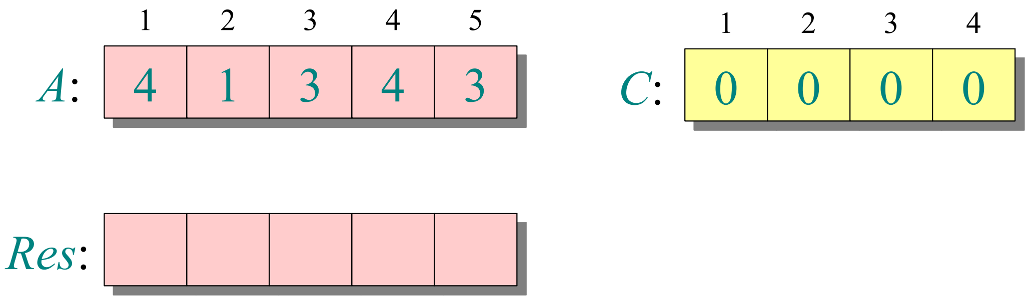
Counting-sort



19.45

Example

Loop 1



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

19.46

Example

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
0	0	0	1

Res:

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

19.47

Example

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	0	1

Res:

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

19.48

Example

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	1	1

Res:

for $j \leftarrow 1$ **to** n **do**

$C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright\ C[i] = |\{\text{nyckel} = i\}|$

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	1	2

Res:

for $j \leftarrow 1$ **to** n **do**

$C[A[j]] \leftarrow C[A[j]] + 1 \quad \triangleright\ C[i] = |\{\text{nyckel} = i\}|$

Loop 2

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

Res:

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

19.51

Example

Loop 3

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	0	2	2

Res:

C':

1	1	2	2

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

19.52

Example

Loop 3

	1	2	3	4	5
A:	4	1	3	4	3
Res:					

	1	2	3	4
C:	1	0	2	2
C':	1	1	3	2

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

19.53

Example

Loop 3

	1	2	3	4	5
A:	4	1	3	4	3
Res:					

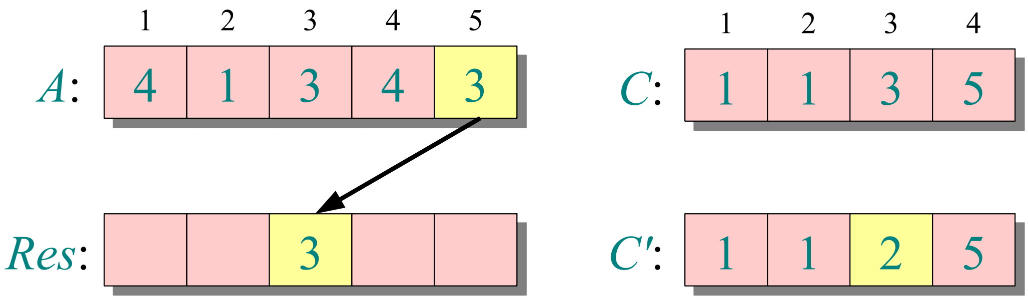
	1	2	3	4
C:	1	0	2	2
C':	1	1	3	5

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

19.54

Example

Loop 4



for $j \leftarrow n$ **downto** 1 **do**

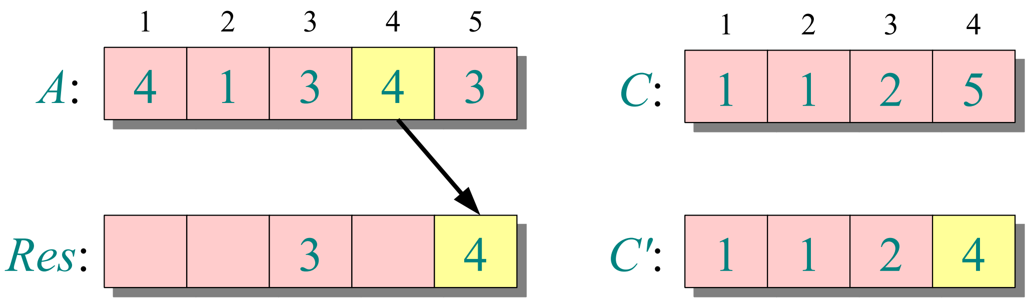
  $Res[C[A[j]]] \leftarrow A[j]$

  $C[A[j]] \leftarrow C[A[j]] - 1$

19.55

Example

Loop 4



for $j \leftarrow n$ **downto** 1 **do**

  $Res[C[A[j]]] \leftarrow A[j]$

  $C[A[j]] \leftarrow C[A[j]] - 1$

19.56

Example

Loop 4

	1	2	3	4	5
A:	4	1	3	4	3
Res:		3	3		4

	1	2	3	4
C:	1	1	2	4
C':	1	1	1	4

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

19.57

Example

Loop 4

	1	2	3	4	5
A:	4	1	3	4	3
Res:	1	3	3		4

	1	2	3	4
C:	1	1	1	4
C':	0	1	1	4

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

19.58

Example

Loop 4

	1	2	3	4	5
A:	4	1	3	4	3
Res:	1	3	3	4	4

	1	2	3	4
C:	0	1	1	4
C':	0	1	1	3

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

19.59

Analysis

$\Theta(k)$ { **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**
  $Res[C[A[j]]] \leftarrow A[j]$
  $C[A[j]] \leftarrow C[A[j]] - 1$

$\Theta(n + k)$

19.60

Execution time

If $k \in O(n)$ Counting sorting takes $\Theta(n)$ time

- But sorting takes $\Omega(n \log n)$ time!
- What is wrong?

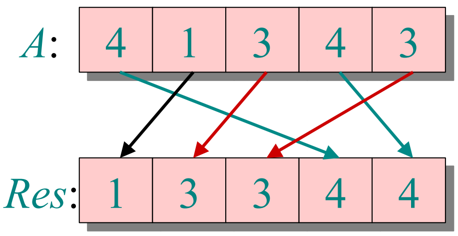
Answer:

- *Comparison based sorting* requires $\Omega(n \log n)$ steps
- Counting-sort is *not* comparison based
- No comparison between the elements!

19.61

Stable sorting

Counting-sort is a stable sorting algorithm: it preserves order among equal elements



To reflect:
Which other sorting algorithms are stable?

19.62

3.2 Bucket-sort

Bucket-sort

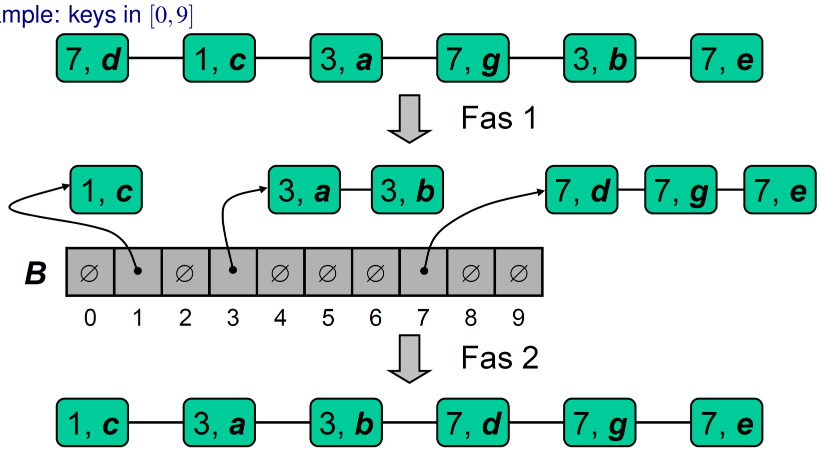
- Let S e a sequence of n pairs (key, value) with keys in $[0, N-1]$
- Bucket-sort uses keys as indices in an array B of sequences
  - Phase 1: Empty the sequence S by moving each pair (k, v) to the end of the bucket $B[k]$
  - Phase 2: For $i = 0, \ldots, N-1$ move the pairs in bucket $B[i]$ to the end of the sequence S
- Analysis:
  - Phase 1 takes $O(n)$ steps
  - Phase 2 takes $O(n+N)$ steps

  Bucket-sort has $O(n+N)$ time complexity

procedure BUCKETSORT(S, N)
    $B \leftarrow$ array with N empty sequences
    while $\neg S$.ISEMPTY() **do**
        $f \leftarrow S$.FIRST()
        $(k, o) \leftarrow S$.REMOVE(f)
        $B[k]$.INSERTLAST$((k, o))$
    for $i \leftarrow 0$ **to** $N-1$ **do**
        while $\neg B[i]$.ISEMPTY() **do**
            $f \leftarrow B[i]$.FIRST()
            $(k, o) \leftarrow B[i]$.REMOVE(f)
            S.INSERTLAST$((k, o))$

19.63

Example: keys in $[0, 9]$



19.64

Properties and extensions

Type of keys:

- Keys are used as indices in an array and can therefore not be of arbitrary types

Stable sorting

- The relative order among pairs with equal keys is preserved

Extensions

- Integers in $[a,b]$
  - Insert a pair (k,v) in bucket $B[k-a]$
- String keys from a finite set of strings D
  - Sort D and compute the range $r(k)$ for each string $k \in D$ in the sorted sequence
  - Insert pair (k,v) in bucket $B[r(k)]$

19.65

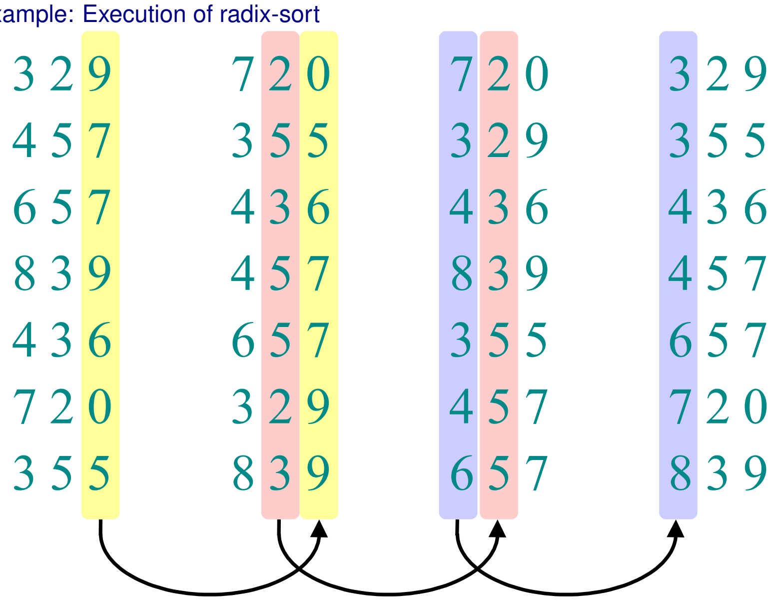
3.3 Radix-sort

Radix-sort

- Origin: Herman Holleriths sorting machine for 1890's census in USA
- digit-by-digit sorting
- Sort starting with the *least significant digit first* with an external *stable* sorting routine

19.66

Example: Execution of radix-sort

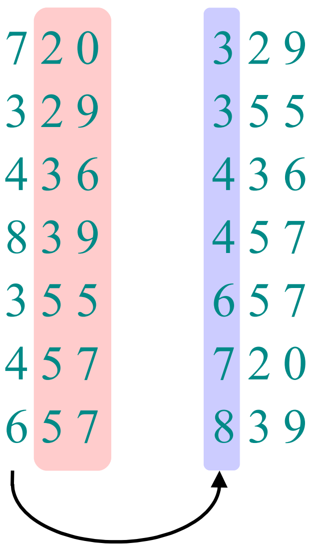


19.67

Correctness of radix-sort

Use induction over digit positions

- Assume the numbers are sorted according to the $t-1$ least significant digits
- Sort according to digit t



19.68

Correctness of radix-sort

Use induction over digit positions

- Assume the numbers are sorted according to the $t-1$ least significant digits
- Sort according to digit t
  - Two numbers that differ in the digit t are correctly sorted

720	329
329	355
436	436
839	457
355	657
457	720
657	839

19.69

Correctness for radix-sort

Use induction over digit positions

- Assume the numbers are sorted according to their $t-1$ least significant digits
- Sort according to digit t
  - Two numbers that differ in the digit t are correctly sorted
  - Two numbers with equal digit t keep their relative order $\Rightarrow$ correct ordering

720	329
329	355
436	436
839	457
355	657
457	720
657	839

19.70

Analysis of radix-sort

- Assume counting sort is used as the external sorting algorithm
- Sorting of n machine words with b bits each
- We can consider each word has b/r digits in base 2^r

Example:

	8	8	8	8
32-bits word				

$r = 8 \Rightarrow b/r = 4$: radix-sort with 4 counting-sort passes on digits in base 2^8
or $r = 16 \Rightarrow b/r = 2$: radix-sort with 2 passes on digits in base 2^{16}

How many passes?

19.71

Analysis of radix-sort

Recall: counting-sort takes $\Theta(n+k)$ execution time to sort n numbers from $[0, k-1]$. If each b-bits word is partitioned into r-words then each counting-sort pass takes $\Theta(n+2^r)$ time. With b/r parts, we get

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

Choose r to minimize $T(n,b)$

- Increasing r gives less passes but if $r \gg \log n$ the required time increases exponentially in r.

19.72

Choose r

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

Minimize $T(n,b)$ by deriving and finding a minimum. Or, observe that we want to avoid $2^r \gg n$ and that it does not hurt asymptotically to have a large r as long as we avoid $2^r \gg n$. Choosing $r = \log n$ gives $T(n,b) = \Theta(bn/\log n)$.

- for numbers in the interval 0 to $n^d - 1$, we get $b = d \log n \Rightarrow$ radix-sort runs in $\Theta(dn)$ time complexity.

19.73

Conclusions

In practice, radix-sort is fast for large input data and simple to encode and maintain

- for numbers in $[0, n^d - 1]$, we get $b = d \log n$ and radix-sort runs in $\Theta(dn)$ time complexity.

Example: 32-bit integers

- At most 3 passes when sorting ~ 2000 numbers.
- Merge-sort and quick-sort use at least $\lceil \log 2000 \rceil = 11$ passes

Disadvantages: You cannot sort in place with counting-sort. Radix sort does not exhibit good locality (quick-sort does) so that a fine tuned quick-sort implementation can be faster on a modern processor with a steep memory hierarchy.

19.74