# SORTING TECHNIQUES

Hanan Samet

Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
e-mail: hjs@umiacs.umd.edu

Copyright © 1997 Hanan Samet

These notes may not be reproduced by any means (mechanical or electronic or any other) without the express written permission of Hanan Samet

ISSUES

1. Nature of storage media
   - fast memory
   - secondary storage
2. Random versus sequential access of the data
3. Space
   - can we use space in excess of the space occupied by the data?
4. Stability
   - is the relative order of duplicate occurrences of a data value preserved?
   - useful as adds a dimension of priority (e.g., time stamps in a scheduling application) for free

Copyright © 1998 by Hanan Samet

1
b
st2

## SELECTION SORT

- Find the smallest element and output it

113 400 818 612 411 →

113
113
113
113
113

Copyright © 1998 by Hanan Samet

st2

## SELECTION SORT

- Find the smallest element and output it

113 400 818 612 411

400 818 612 411

113

113 400

113 400

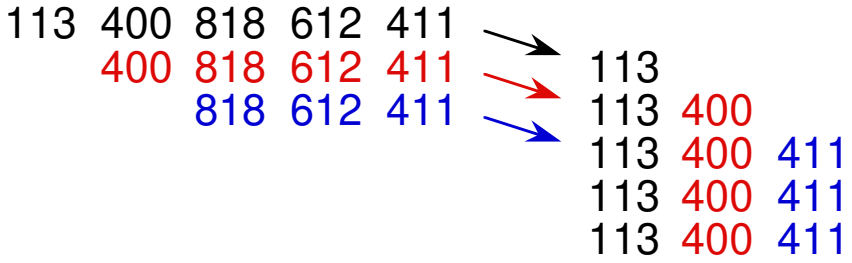
113 400

113 400

Copyright © 1998 by Hanan Samet

# SELECTION SORT

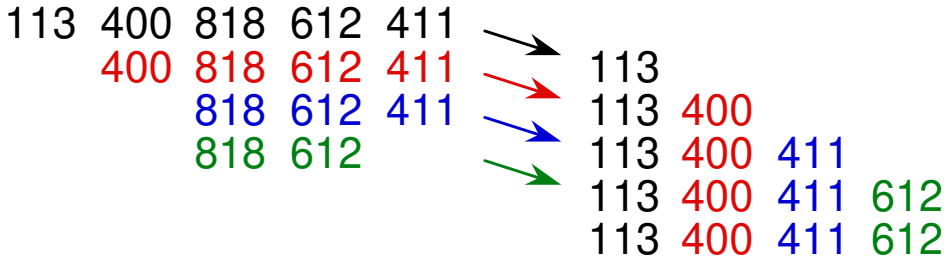
- Find the smallest element and output it




Copyright © 1998 by Hanan Samet

# SELECTION SORT

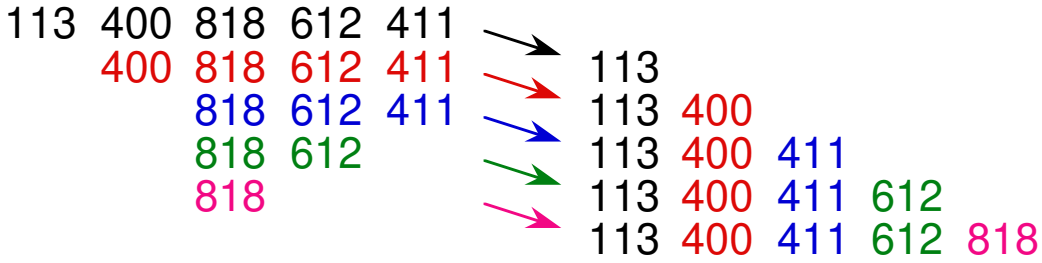
- Find the smallest element and output it




Copyright © 1998 by Hanan Samet

# SELECTION SORT

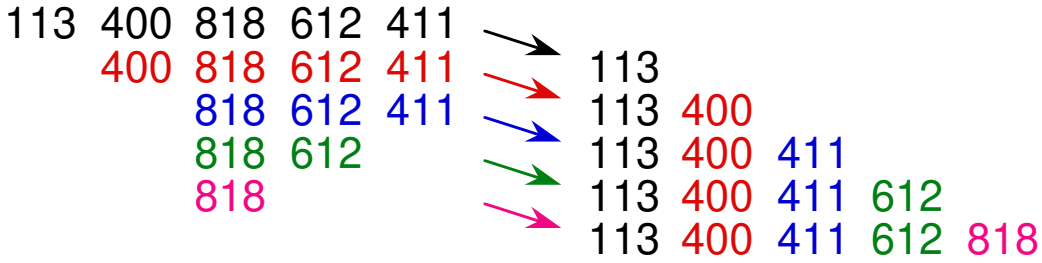
- Find the smallest element and output it




Copyright © 1998 by Hanan Samet

# SELECTION SORT

- Find the smallest element and output it




- $O(n^2)$ time as always $n \cdot (n-1)$ comparisons
- Needs extra space for output
- In-place variant is not stable
- Ex:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 113 | $400_1$ | 818 | $400_2$ | 411 | 311 | 412 | 420 |

  Step 1 causes no change as 113 is the smallest element

  Step 2 causes 311 in position 6 to be exchanged with 400 in position 2 resulting in the first instance of 400 becoming the second instance

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 113 | 311 | 818 | $400_2$ | 411 | $400_1$ | 412 | 420 |

- In-place variant can be made stable:
  1. use linked lists
     - only $n \cdot (n-1)/2$ comparisons
  2. exchange adjacent elements

Copyright © 1998 by Hanan Samet

# BUBBLE SORT

- Smaller values filter their way to the start of the list while the larger values filter their way to the end of the list
- Data is moved one position at a time and hence method is stable
- In-place

| | | | | | |
|---|---|---|---|---|---|
| Initially: | 113 | 400 | 818 | 612 | 411 |
| First pass: | 113 — | 400 | 818 | 612 | 411 |
| | 113 | 400 — | 818 | 612 | 411 |
| | 113 | 400 | 818 ↔ | 612 | 411 |
| | 113 | 400 | 612 | 818 ↔ | 411 |
| | 113 | 400 | 612 | 411 | 818* |
| Second pass: | 113 — | 400 | 612 | 411 | 818* |
| | 113 | 400 — | 612 | 411 | 818* |
| | 113 | 400 | 612 ↔ | 411 | 818* |
| | 113 | 400 | 411 | 612* | 818* |

* denotes final position
— denotes a key comparison
↔ denotes a key comparison and a need to swap

- Procedure starts with the values at the right end of the input list and works its way to the left in the sense that larger values are placed in their proper position before (timewise) the smaller value
- $O(n^2)$ comparison and exchange operations

Copyright © 1998 by Hanan Samet

## INSERTION SORT

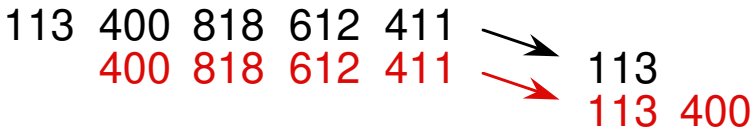
- Values on the input list are taken in sequence and put in their proper position in the output list
- Previously moved values may have to be moved again
- Makes only one pass on the input list at cost of move operations

113 400 818 612 411 → 113

Copyright © 1998 by Hanan Samet

# INSERTION SORT

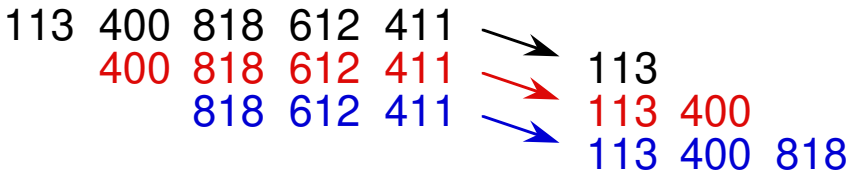
- Values on the input list are taken in sequence and put in their proper position in the output list
- Previously moved values may have to be moved again
- Makes only one pass on the input list at cost of move operations




Copyright © 1998 by Hanan Samet

# INSERTION SORT

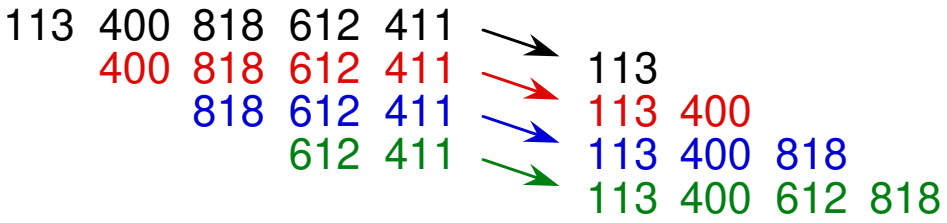
- Values on the input list are taken in sequence and put in their proper position in the output list
- Previously moved values may have to be moved again
- Makes only one pass on the input list at cost of move operations




Copyright © 1998 by Hanan Samet

# INSERTION SORT

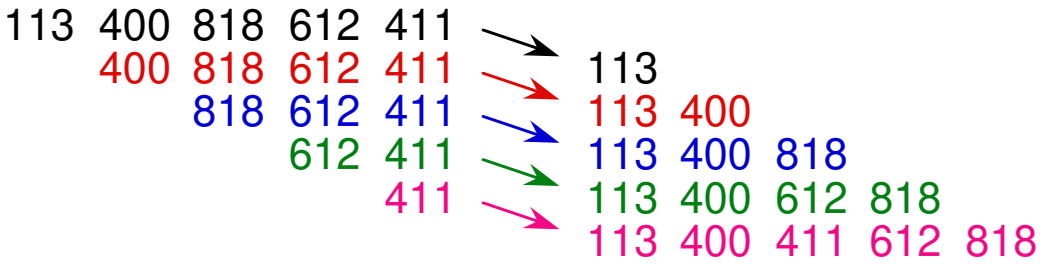
- Values on the input list are taken in sequence and put in their proper position in the output list
- Previously moved values may have to be moved again
- Makes only one pass on the input list at cost of move operations




Copyright © 1998 by Hanan Samet

# INSERTION SORT

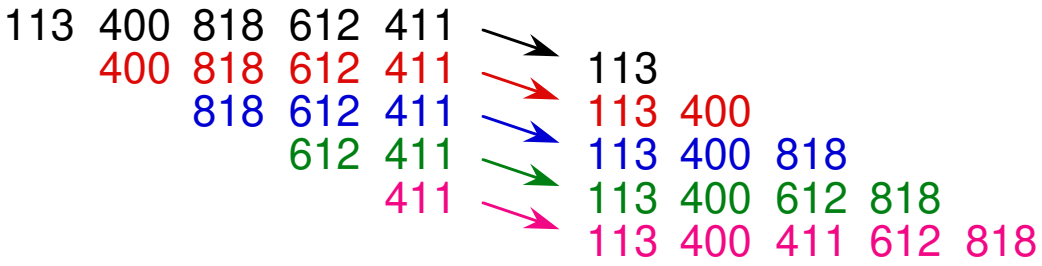
- Values on the input list are taken in sequence and put in their proper position in the output list
- Previously moved values may have to be moved again
- Makes only one pass on the input list at cost of move operations




Copyright © 1998 by Hanan Samet

# INSERTION SORT

- Values on the input list are taken in sequence and put in their proper position in the output list
- Previously moved values may have to be moved again
- Makes only one pass on the input list at cost of move operations




- $O(n^2)$ but almost always smaller number of operations than selection sort
- Needs extra space for output
- Stable
- In-place variant exists and can also be made stable

Copyright © 1998 by Hanan Samet

## MECHANICS OF INSERTION

- Assume inserting *x*, the $i^{th}$ element
- Must search for position to make the insertion
- Two methods:
  1. search in increasing order
     - Ex: insert 411 in [113 400 612 818] →
  2. search in decreasing order
     - Ex: insert 411 in [113 400 612 818] ←

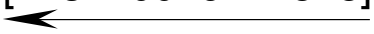
Copyright © 1998 by Hanan Samet

## SEARCH IN INCREASING ORDER

- Ex: insert 411 in [113 400 612 818] $\longrightarrow$
- Initialize the content of the output position $i$ to $\infty$ so that we don't have to test if we are at end of current list
- Start at $j$=1 and find first position $j$ where $x$ belongs — i.e., $x <$ out[$j$]
- Move all elements at positions $k$ ($k \geq j$) one to right (i.e., to $k$+1)
- $j$ comparisons and $i-j+1$ moves
- Total of $i$+1 operations for step $i$
- Total of $(n+1)(n+2)/2-1 = n(n+3)/2$ operations independent of whether the file is initially sorted, not sorted, or sorted in reverse order

Copyright © 1998 by Hanan Samet

# SEARCH IN DECREASING ORDER

- Ex: insert 411 in [113 400 612 818]
  
- Initialize the contents of the output position 0 to $-\infty$ so that we don't have to test if we are at end (actually the start) of the current list
- Start at $j = i$ and find first position $j$ where $x$ belongs — i.e., out[$j-1$] $\leq$ x
- Each time test is false, move the element tested (i.e., at $j-1$) one to the right (i.e., $j$ )
- $i-j+1$ comparisons and $i-j$ moves for step $i$
- Total of as many as $n(n+1)/2$ comparisons and $n(n-1)/2$ moves when data is sorted in reverse order
- Total of as few as $n$ comparisons and 0 moves when the data is already sorted in increasing order
- Superior to a search increasing order

Copyright © 1998 by Hanan Samet

# SORTING BY DISTRIBUTION COUNTING

- Insertion sort must perform move operations because when a value is inserted in the output list, we don't know its final position
- If we know the distribution of the various values, then we can calculate their final position in advance
- If range of values is finite, use an extra array of counters
- Algorithm:
  1. count frequency of each value
  2. calculate the destination address for each value by making a pass over the array of counters in reverse order and reuse the counter field
  3. place the values in the right place in the output list
- $O(m+n)$ time for $n$ values in the range 0 … $m$–1 and stable
- Not in-place because on the final step we can't tell if a value is in its proper location as values are not inserted in the output list in sorted order
- Ex:

| input list: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 3a | 0a | 8a | 2a | 1a | 1b | 2b | 0b |

| counter: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| value: | 2 | 2 | 2 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| position: | 0 | 2 | 4 | 6 | 7 | 7 | 7 | 7 | 7 | 8 |

| output list: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 0a | 0b | 1a | 1b | 2a | 2b | 3a | 8a |

Copyright © 1998 by Hanan Samet

# SHELLSORT

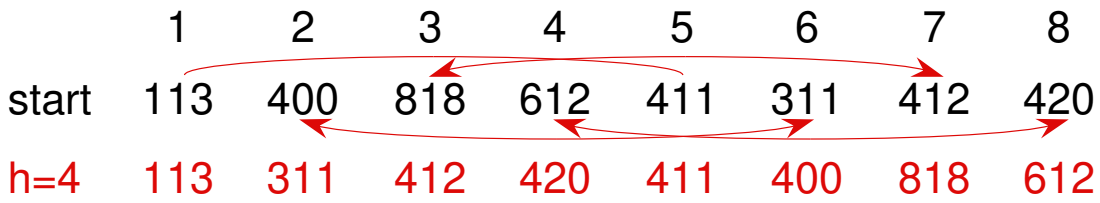
- Drawback of insertion and bubble sort is that move operations only involve adjacent elements
- If largest element is initially at the start, then by the end of the sort it will have been the subject of *n* move or exchange operations
- Can reduce number of move operations by allowing values to make larger jumps
- Sort successively larger sublists
  1. e.g., sort every fourth element
     - elements 1, 5, 9, 13, 17 form one sorted list
     - elements 2, 6, 10, 14, 18 form a second sorted list
     - elements 3, 7, 11, 15, 19 form a third sorted list
     - elements 4, 8, 12, 16, 20 form a fourth sorted list
  2. apply same technique with successively shorter increments
  3. final step uses an increment of size 1
- Usually implement by using insertion sort (that searches for position to make the insertion in decreasing order) at each step
- Ex: with increment sequence of 4, 2, 1

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |

Copyright © 1998 by Hanan Samet

# SHELLSORT

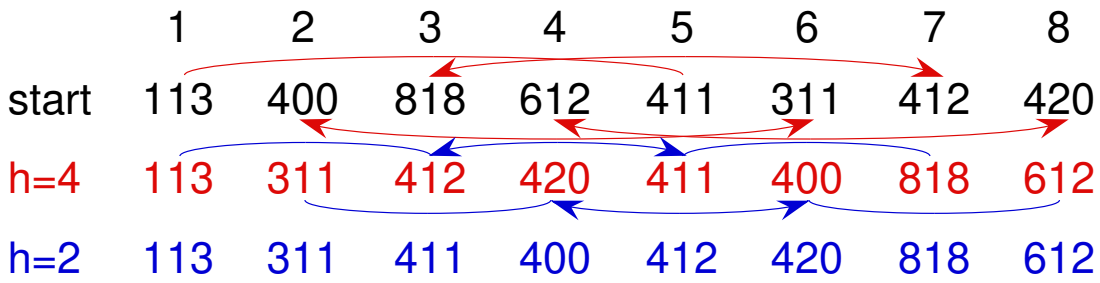
- Drawback of insertion and bubble sort is that move operations only involve adjacent elements
- If largest element is initially at the start, then by the end of the sort it will have been the subject of *n* move or exchange operations
- Can reduce number of move operations by allowing values to make larger jumps
- Sort successively larger sublists
  1. e.g., sort every fourth element
     - elements 1, 5, 9, 13, 17 form one sorted list
     - elements 2, 6, 10, 14, 18 form a second sorted list
     - elements 3, 7, 11, 15, 19 form a third sorted list
     - elements 4, 8, 12, 16, 20 form a fourth sorted list
  2. apply same technique with successively shorter increments
  3. final step uses an increment of size 1
- Usually implement by using insertion sort (that searches for position to make the insertion in decreasing order) at each step
- Ex: with increment sequence of 4, 2, 1




Copyright © 1998 by Hanan Samet

# SHELLSORT

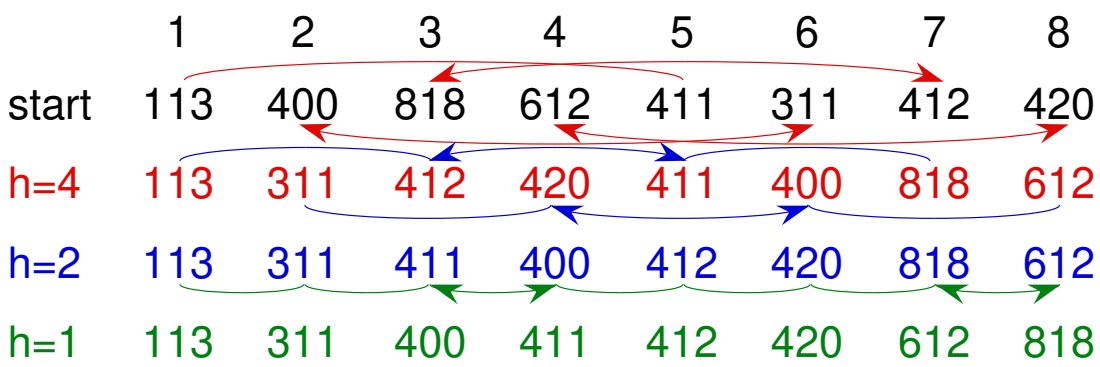
- Drawback of insertion and bubble sort is that move operations only involve adjacent elements
- If largest element is initially at the start, then by the end of the sort it will have been the subject of *n* move or exchange operations
- Can reduce number of move operations by allowing values to make larger jumps
- Sort successively larger sublists
  1. e.g., sort every fourth element
     - elements 1, 5, 9, 13, 17 form one sorted list
     - elements 2, 6, 10, 14, 18 form a second sorted list
     - elements 3, 7, 11, 15, 19 form a third sorted list
     - elements 4, 8, 12, 16, 20 form a fourth sorted list
  2. apply same technique with successively shorter increments
  3. final step uses an increment of size 1
- Usually implement by using insertion sort (that searches for position to make the insertion in decreasing order) at each step
- Ex: with increment sequence of 4, 2, 1




Copyright © 1998 by Hanan Samet

# SHELLSORT

- Drawback of insertion and bubble sort is that move operations only involve adjacent elements
- If largest element is initially at the start, then by the end of the sort it will have been the subject of *n* move or exchange operations
- Can reduce number of move operations by allowing values to make larger jumps
- Sort successively larger sublists
  1. e.g., sort every fourth element
     - elements 1, 5, 9, 13, 17 form one sorted list
     - elements 2, 6, 10, 14, 18 form a second sorted list
     - elements 3, 7, 11, 15, 19 form a third sorted list
     - elements 4, 8, 12, 16, 20 form a fourth sorted list
  2. apply same technique with successively shorter increments
  3. final step uses an increment of size 1
- Usually implement by using insertion sort (that searches for position to make the insertion in decreasing order) at each step
- Ex: with increment sequence of 4, 2, 1




Copyright © 1998 by Hanan Samet

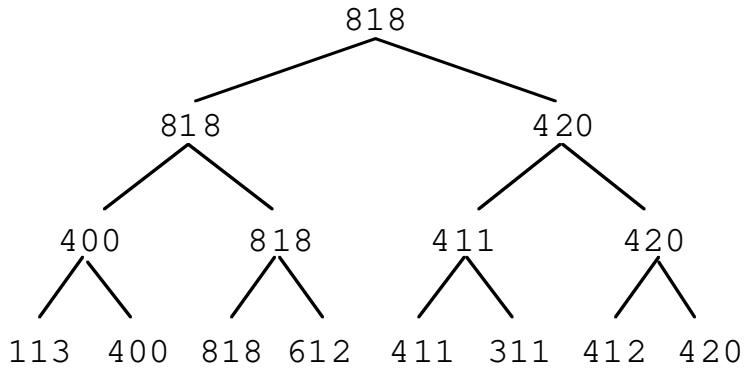
# PROPERTIES OF SHELLSORT

- Difficult to analyze execution time but is good due to being simple, having a fast inner loop, and no bad cases
- Some increment sequences yield $O(n\sqrt{n})$
- Others of form $2^p3^q$ (e.g., 1, 2, 3, 4, 6, 9, 16, 18, …) yield $O(n \log_2 n)$
- Some increment sequences are better than others
- Not stable but in-place
- 1, 2, 4, 8, … is not so good as doesn't allow for interaction between values in odd and even positions until final step
- Desirable for $\log_2 n$ increments in order to reduce the number of passes that need to be made over the data
- Not stable as order of equal values is destroyed due to application of sorting to overlapping lists
- In-place

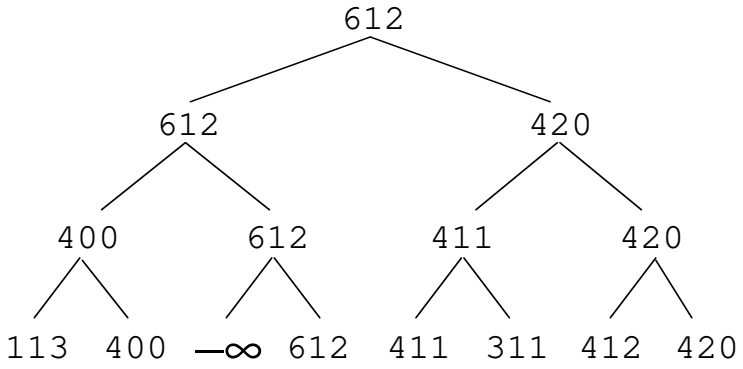
Copyright © 1998 by Hanan Samet

# TOURNAMENT SORT

- Like a ladder




- Output value at the root, replace it by $-\infty$, and rebuild the ladder

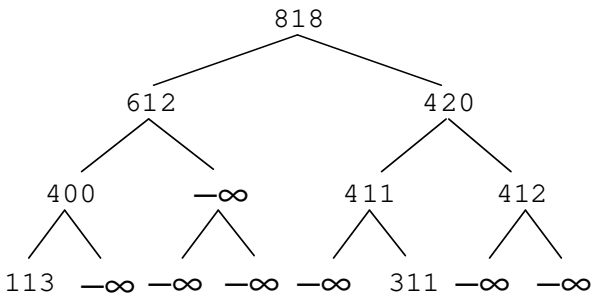



- Sorting takes $O(n \log_2 n)$ once the ladder has been built

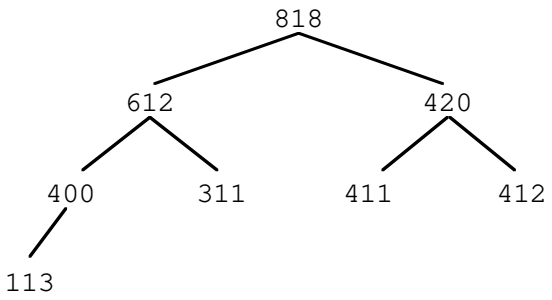
Copyright © 1998 by Hanan Samet

# HEAPSORT (TREESORT)

- Tournament ladder is wasteful of space as values are repeated
- Need space to store the output of the sort
- Use a modified tournament ladder where only the winners are retained




- Rearrange values so get a reasonably complete binary tree




- Result is a heap
  1. each value is greater than its two sons
  2. use a complete binary tree array representation
     - sons of node at location $x$ are at $2x$ and $2x+1$

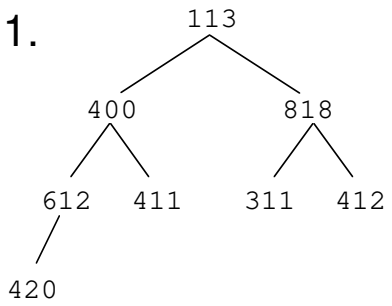
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 818 | 612 | 420 | 400 | 311 | 411 | 412 | 113 |

Copyright © 1998 by Hanan Samet

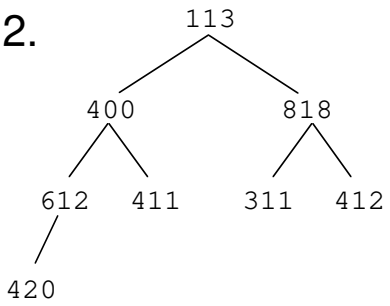
# BUILDING A HEAP

- Start at bottom level of tree
  1. compare maximum of each pair of brother values with their father
  2. if father < maximum of sons, then exchange father with maximum son and repeat as necessary with two sons
- Repeat process for remaining values on the same level, and the remaining levels while ascending the tree

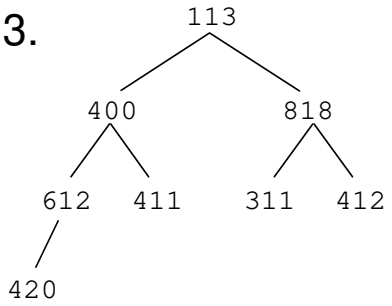
1.



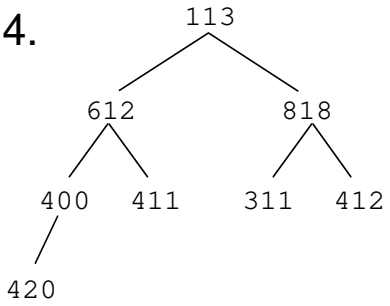

2.



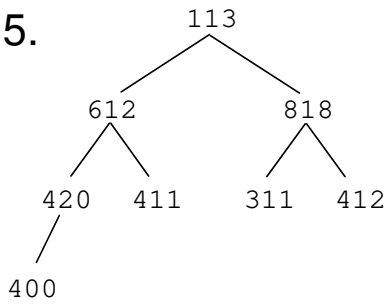

3.



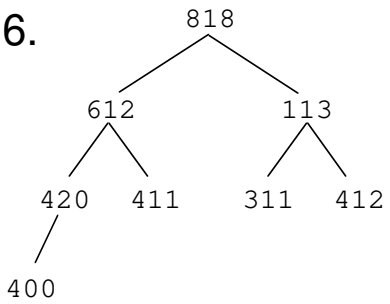

4.




5.




6.

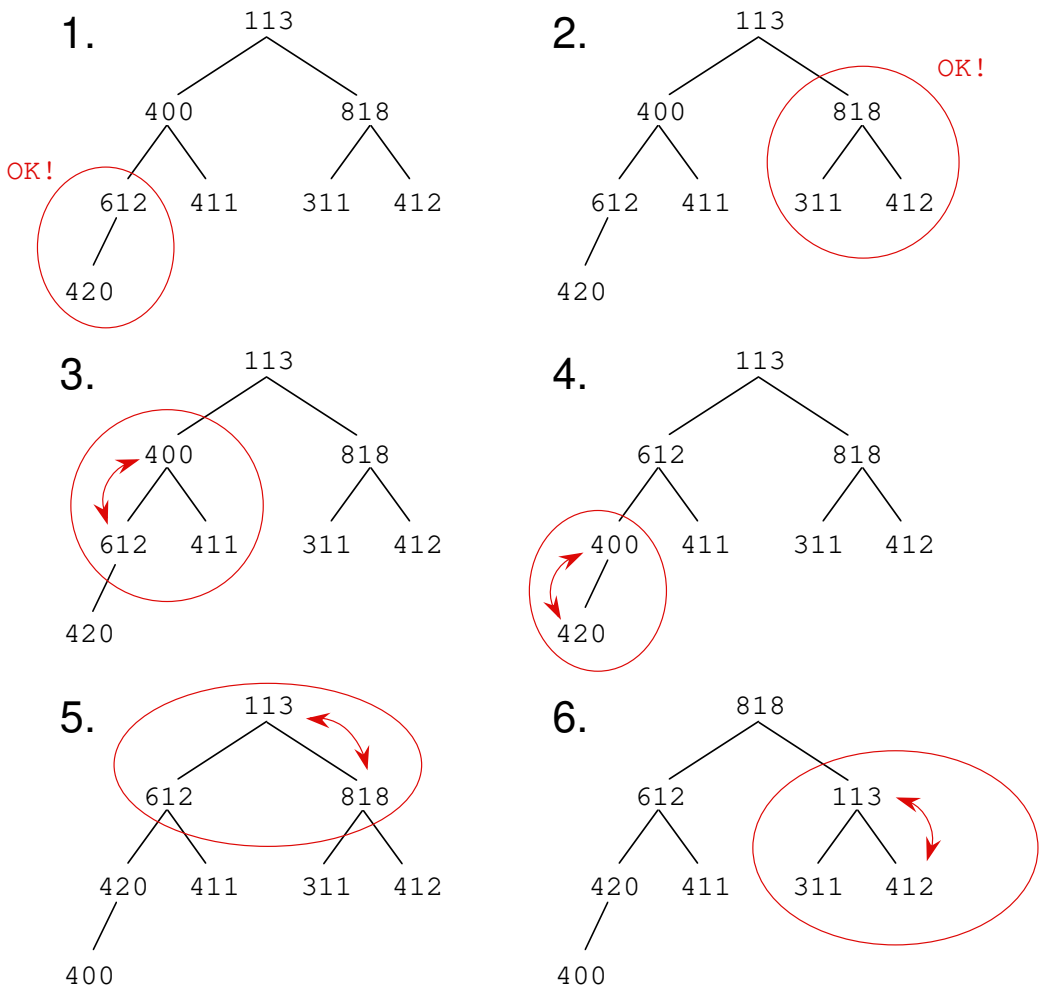



- Process is not stable

Copyright © 1998 by Hanan Samet

# BUILDING A HEAP

- Start at bottom level of tree
  1. compare maximum of each pair of brother values with their father
  2. if father < maximum of sons, then exchange father with maximum son and repeat as necessary with two sons
- Repeat process for remaining values on the same level, and the remaining levels while ascending the tree

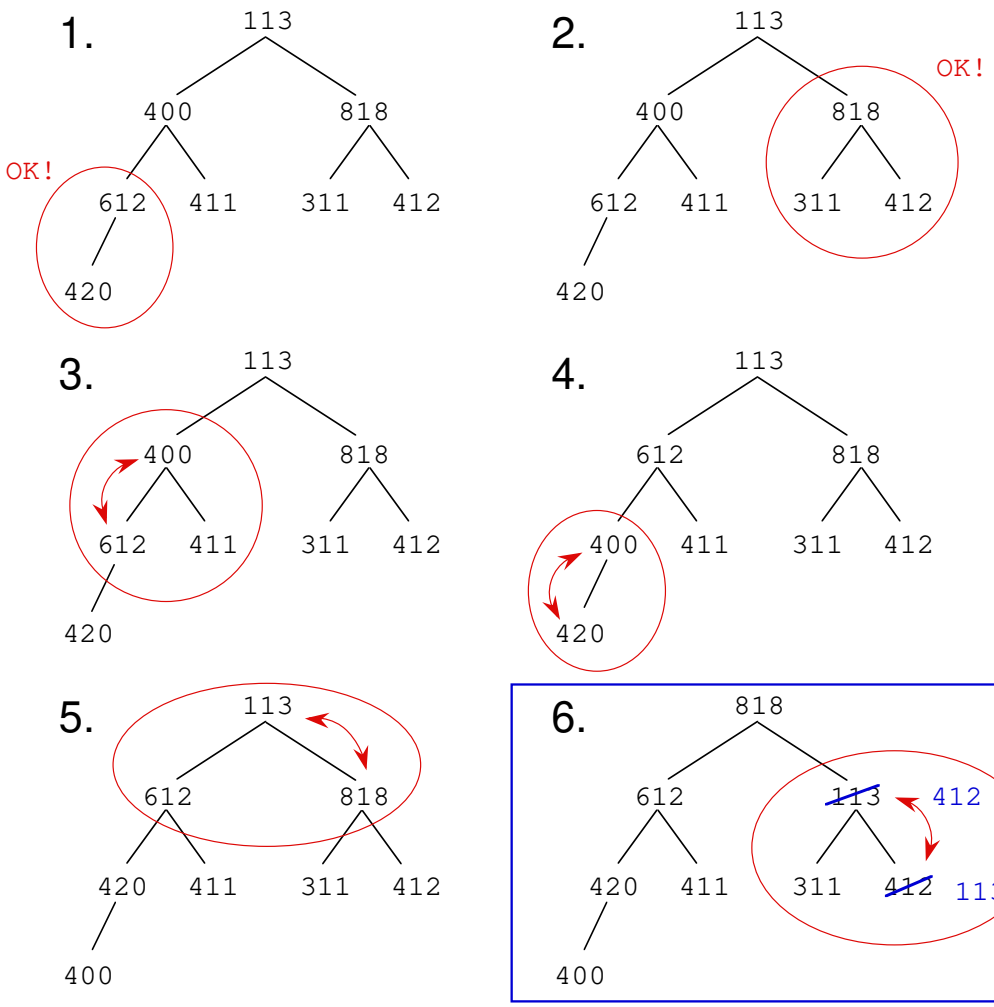



- Process is not stable

Copyright © 1998 by Hanan Samet

# BUILDING A HEAP

- Start at bottom level of tree
  1. compare maximum of each pair of brother values with their father
  2. if father < maximum of sons, then exchange father with maximum son and repeat as necessary with two sons
- Repeat process for remaining values on the same level, and the remaining levels while ascending the tree




- Process is not stable
- Final result

Copyright © 1998 by Hanan Samet

## OUTPUTTING A HEAP

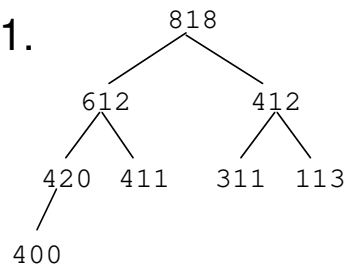
1. Exchange root with node at the end of the unsorted set
2. Compare root with roots of left and right subtrees
3. If root > left and root > right, then done
4. If root > max(left,right) and max(left,right)=left, then exchange root with left son and reapply steps 1–5 to left subtree
5. Else exchange root with right son and reapply steps 1–5 to right subtree

Copyright © 1998 by Hanan Samet

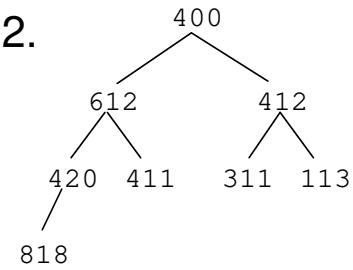
1 b st15

# EXAMPLE OF OUTPUTTING A HEAP

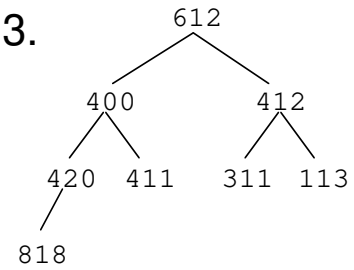
1.



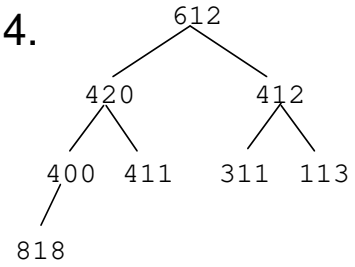

2.



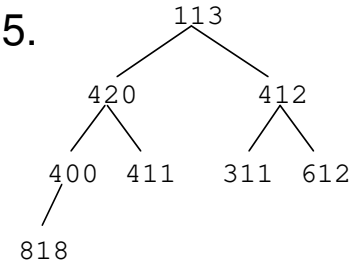

3.



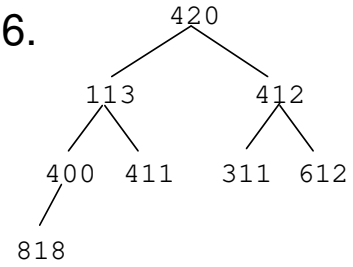

4.



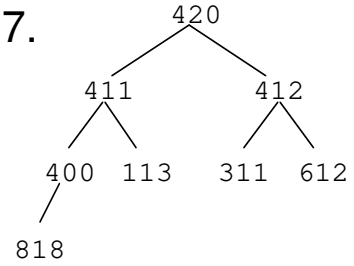

5.



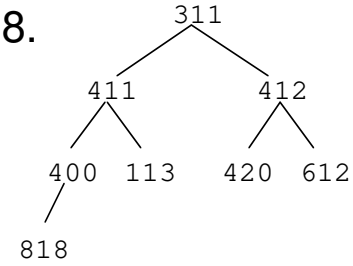

6.



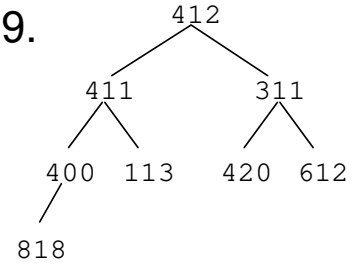

7.



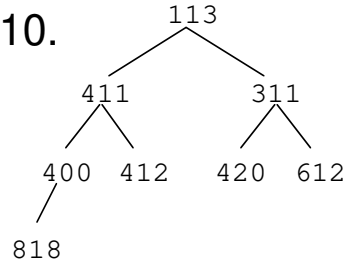

8.



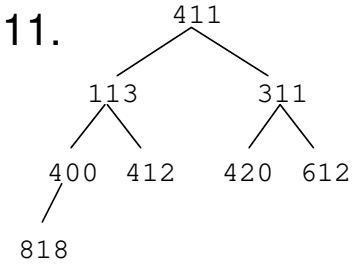

9.



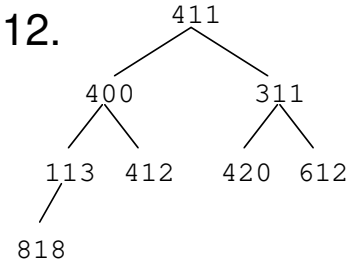

10.



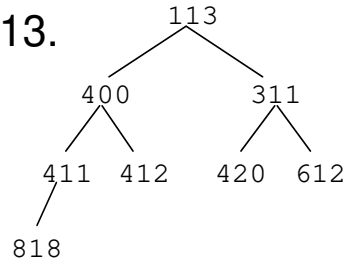

11.



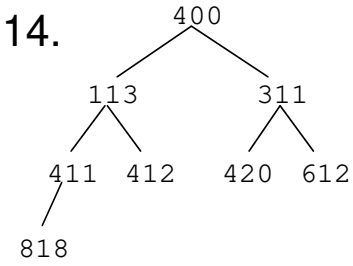

12.



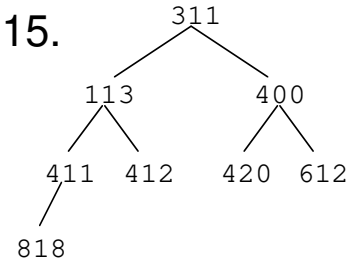

13.




14.

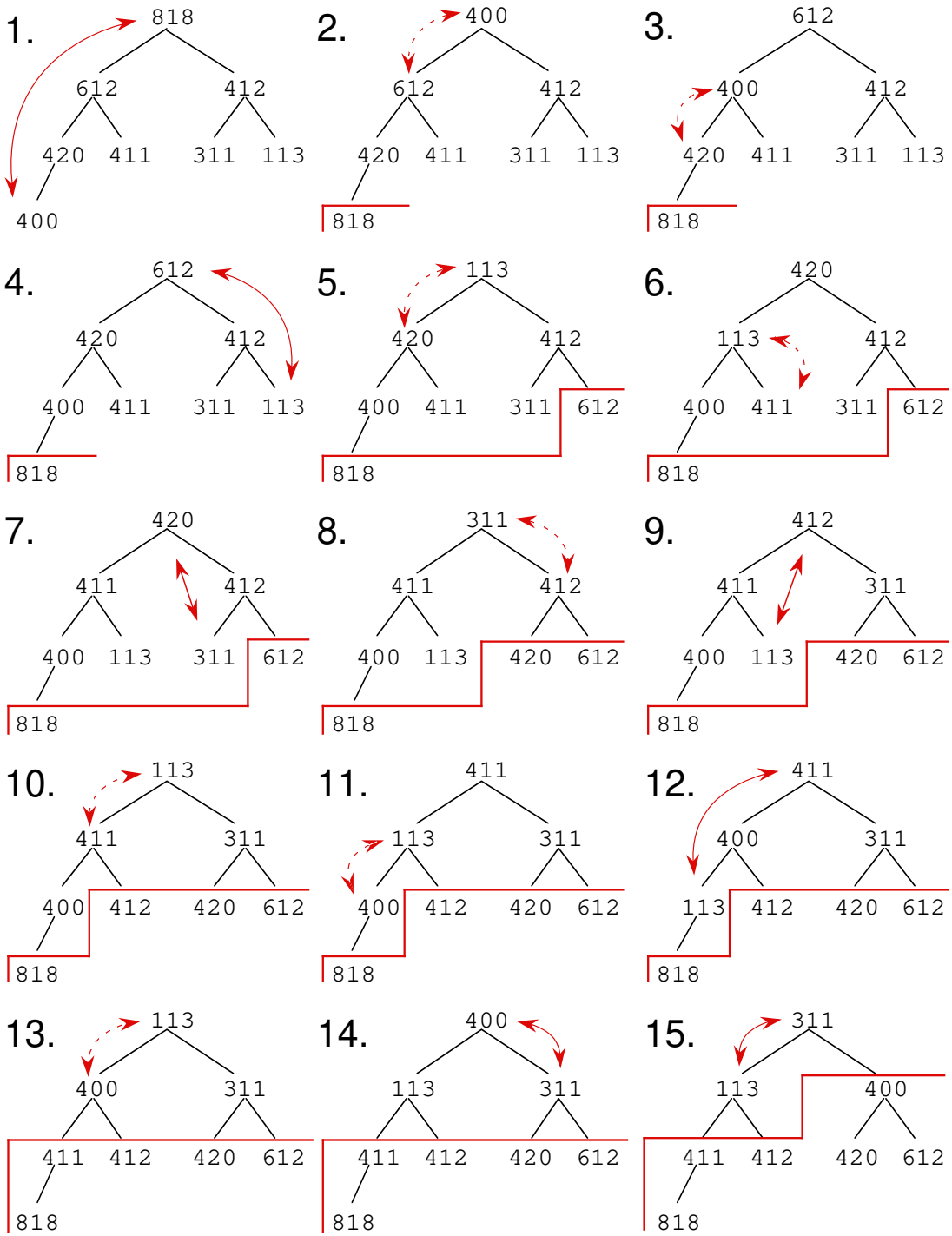



15.




Copyright © 1998 by Hanan Samet

# EXAMPLE OF OUTPUTTING A HEAP

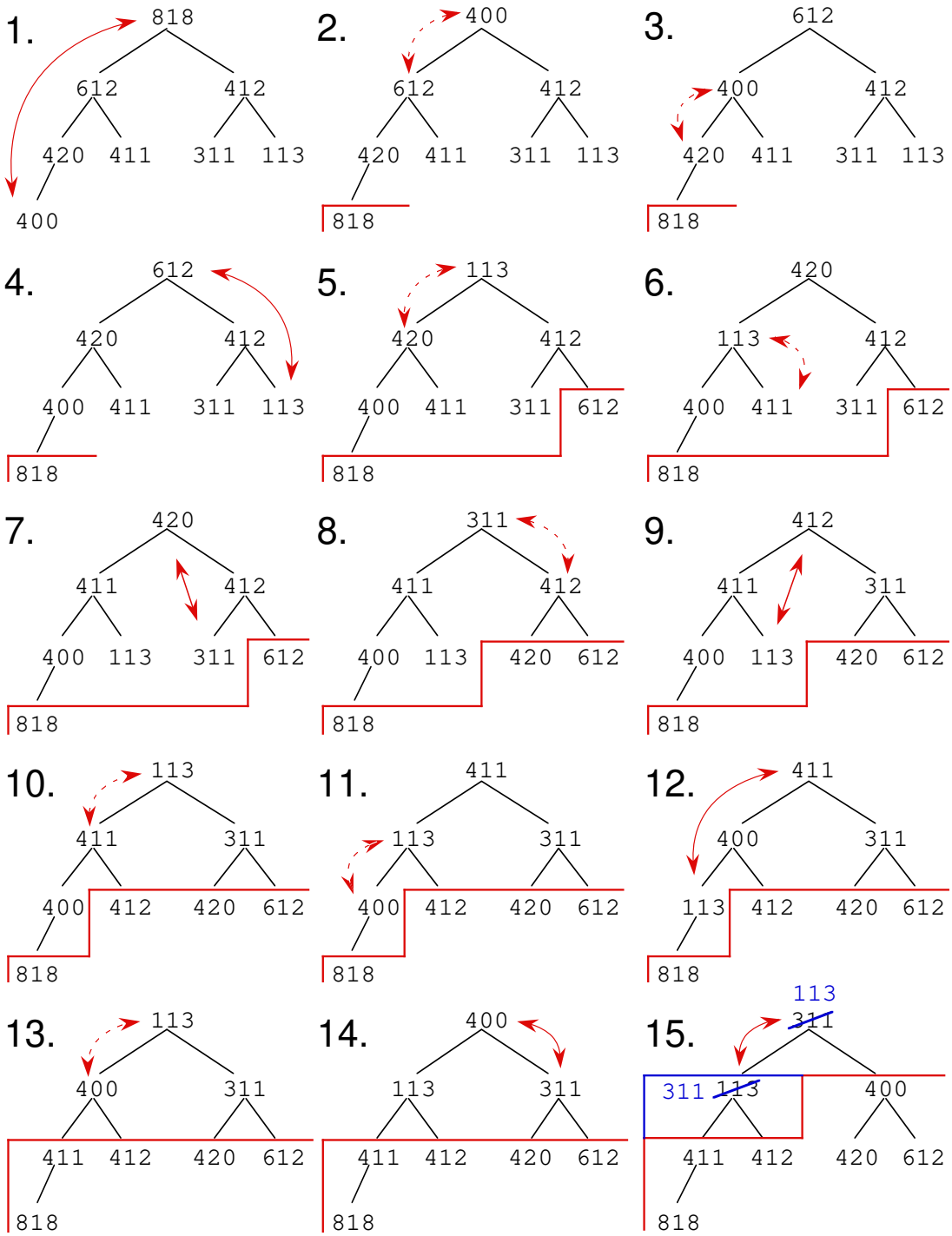



exchange and output

compare and exchange

Copyright © 1998 by Hanan Samet

# EXAMPLE OF OUTPUTTING A HEAP




exchange and output

compare and exchange

- Final result

Copyright © 1998 by Hanan Samet

# ANALYSIS OF HEAPSORT

1. Outputting the heap is $O(n \log_2 n)$
2. Building the heap
   - based on counting the number of comparisons when each position in the heap serves as the starting point of a comparison process
     a. root is at level $\lfloor \log_2 n \rfloor$
     b. between 1 and $\lceil n / 2 \rceil$ nodes at level 0
     c. root is the start of at most $2 \lfloor \log_2 n \rfloor$ comparison operations
     d. nodes at level 1 are start of at most 2 comparison operations, and there are at most $\lceil n / 4 \rceil$ of them
     e. nodes at level $i$ are the start of at most $2i$ comparison operations, and there are at most $\left\lceil n / 2^{i+1} \right\rceil$ of them

   - $C$ = number of comparisons = $\sum_{i=1}^{\lfloor \log_2 n \rfloor} \frac{n \cdot i}{2^i}$.

$$\sum_{i=1}^{a} \frac{i}{2^i} = \frac{1}{2} \cdot \sum_{i=0}^{a-1} \frac{i+1}{2^i} = \frac{1}{2} \cdot \sum_{i=0}^{a-1} \frac{i}{2^i} + \frac{1}{2} \cdot \sum_{i=0}^{a-1} \frac{1}{2^i}$$

$$= \frac{1}{2} \cdot \sum_{i=1}^{a-1} \frac{i}{2^i} + \frac{1}{2} \cdot \frac{1 - \frac{1}{2^a}}{1 - \frac{1}{2}} = \frac{1}{2} \cdot \sum_{i=1}^{a} \frac{i}{2^i} - \frac{a}{2^{a+1}} + 1 - \frac{1}{2^a}$$

or, $\sum_{i=1}^{a} \frac{i}{2^i} = 2 - \frac{a+2}{2^a}$

Substituting into $C$ yields

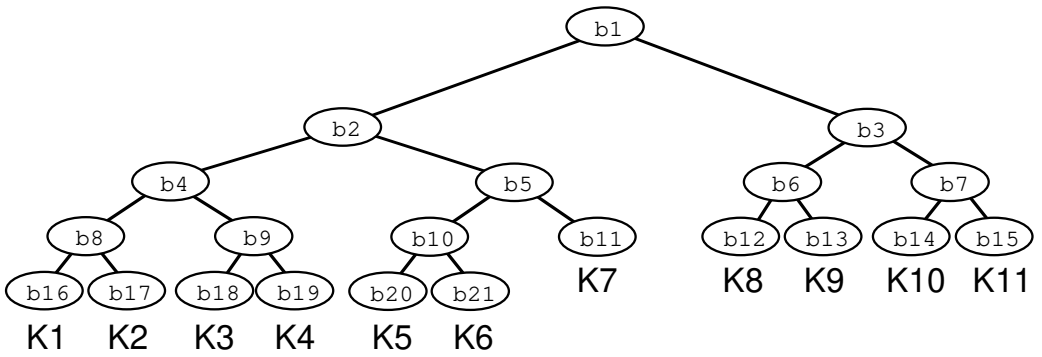
$$C = n \cdot \left( 2 - \frac{\lfloor \log_2 n \rfloor + 2}{2^{\lfloor \log_2 n \rfloor}} \right) \approx 2 \cdot n$$

Copyright © 1998 by Hanan Samet

# PRIORITY QUEUES

- Properties
  1. locate the maximum (minimum) element in O(1) time
  2. insert and delete an arbitrary element in O($\log_2 n$) time
- Can implement using a heap
- If set of possible elements is known a priori, then can use a bit representation consisting of an array corresponding to a complete binary tree (Abel)
  1. all information is stored at the bottom two levels
  2. for $n$ elements, need just $2n-1$ bits
  3. node $b_i$ is 1 if one of the elements below it is present
- Ex: set of possible keys

$$\{K_1, K_2, K_3, K_4, K_5, K_6, K_7, K_8, K_9, K_{10}, K_{11}\}$$

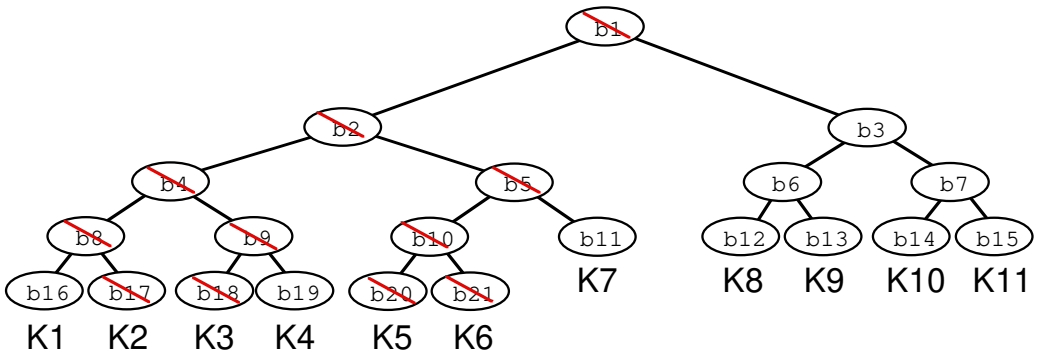



Copyright © 1998 by Hanan Samet

# PRIORITY QUEUES

- Properties
  1. locate the maximum (minimum) element in O(1) time
  2. insert and delete an arbitrary element in O($\log_2 n$) time
- Can implement using a heap
- If set of possible elements is known a priori, then can use a bit representation consisting of an array corresponding to a complete binary tree (Abel)
  1. all information is stored at the bottom two levels
  2. for $n$ elements, need just $2n-1$ bits
  3. node $b_i$ is 1 if one of the elements below it is present
- Ex: set of possible keys

$$\{K_1, K_2, K_3, K_4, K_5, K_6, K_7, K_8, K_9, K_{10}, K_{11}\}$$




For the set $S = \{K_2, K_3, K_5, K_6\}$ we have

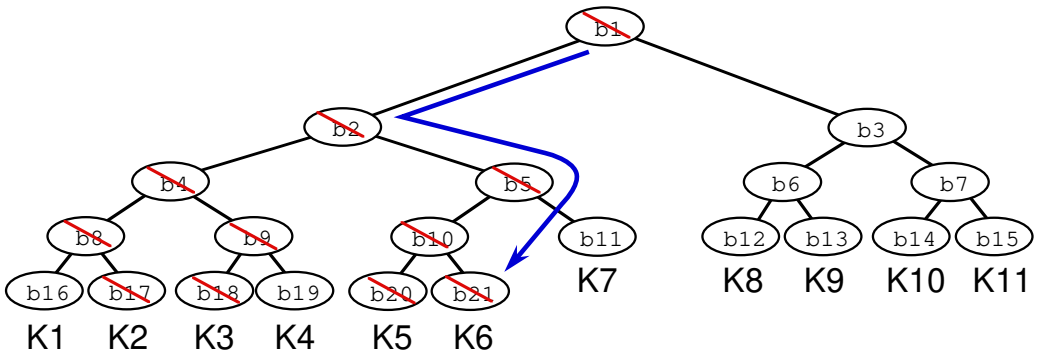
$b_1 = b_2 = b_4 = b_5 = b_8 = b_9 = b_{10} = b_{17} = b_{18} = b_{20} = b_{21} = 1$

Copyright © 1998 by Hanan Samet

# PRIORITY QUEUES

- Properties
  1. locate the maximum (minimum) element in O(1) time
  2. insert and delete an arbitrary element in O($\log_2 n$) time
- Can implement using a heap
- If set of possible elements is known a priori, then can use a bit representation consisting of an array corresponding to a complete binary tree (Abel)
  1. all information is stored at the bottom two levels
  2. for $n$ elements, need just $2n-1$ bits
  3. node $b_i$ is 1 if one of the elements below it is present
- Ex: set of possible keys

$$\{K_1, K_2, K_3, K_4, K_5, K_6, K_7, K_8, K_9, K_{10}, K_{11}\}$$




For the set $S = \{K_2, K_3, K_5, K_6\}$ we have

$b_1 = b_2 = b_4 = b_5 = b_8 = b_9 = b_{10} = b_{17} = b_{18} = b_{20} = b_{21} = 1$

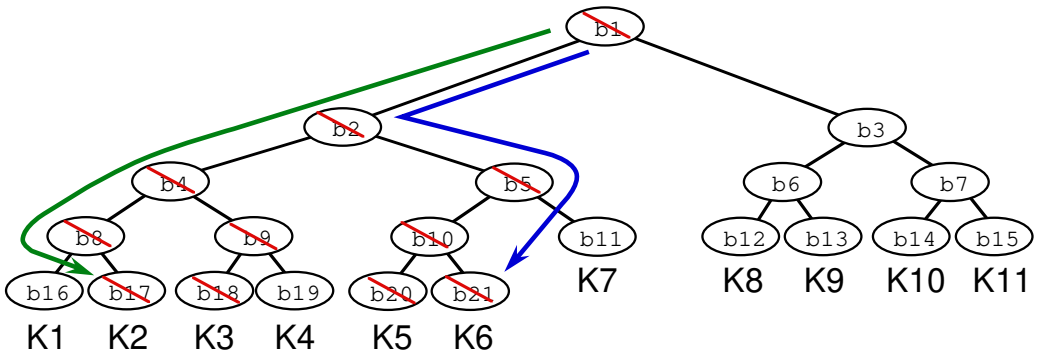
- To locate the maximum element:
  1. $j \leftarrow 1$
  2. while $j < n$ do $j \leftarrow 2j + b_{2j+1}$

Copyright © 1998 by Hanan Samet

# PRIORITY QUEUES

- Properties
  1. locate the maximum (minimum) element in O(1) time
  2. insert and delete an arbitrary element in O($\log_2 n$) time
- Can implement using a heap
- If set of possible elements is known a priori, then can use a bit representation consisting of an array corresponding to a complete binary tree (Abel)
  1. all information is stored at the bottom two levels
  2. for $n$ elements, need just $2n-1$ bits
  3. node $b_i$ is 1 if one of the elements below it is present
- Ex: set of possible keys

$$\{K_1, K_2, K_3, K_4, K_5, K_6, K_7, K_8, K_9, K_{10}, K_{11}\}$$




For the set $S = \{K_2, K_3, K_5, K_6\}$ we have

$b_1 = b_2 = b_4 = b_5 = b_8 = b_9 = b_{10} = b_{17} = b_{18} = b_{20} = b_{21} = 1$

- To locate the maximum element:
  1. $j \leftarrow 1$
  2. while $j<n$ do $j \leftarrow 2j + b_{2j+1}$
- To locate the minimum element:
  1. $j \leftarrow 1$
  2. while $j<n$ do $j \leftarrow 2j+1 - b_{2j}$

Copyright © 1998 by Hanan Samet

# QUICKSORT

- Key:
  1. break file into successive halves such that each half contains all keys greater than a certain value
  2. recursive application of this partitioning scheme to halves yields a sorted list once a partition has only one element
- Algorithm:
  1. pick an item in the array at random, say $x$
  2. scan the array of keys from the left until $a_i \geq x$ is found
  3. scan the array of keys from the right until $a_j \leq x$ is found
  4. exchange $a_i$ and $a_j$ and continue the scan and swap process until the two scans meet (note that $x$ itself may also be moved)
- Ex:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |

pick item 5 at random (i.e. 411) and scan starting at position 1

Copyright © 1998 by Hanan Samet

# QUICKSORT

- Key:
  1. break file into successive halves such that each half contains all keys greater than a certain value
  2. recursive application of this partitioning scheme to halves yields a sorted list once a partition has only one element
- Algorithm:
  1. pick an item in the array at random, say $x$
  2. scan the array of keys from the left until $a_i \geq x$ is found
  3. scan the array of keys from the right until $a_j \leq x$ is found
  4. exchange $a_i$ and $a_j$ and continue the scan and swap process until the two scans meet (note that $x$ itself may also be moved)
- Ex:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |

pick item 5 at random (i.e. 411) and scan starting at position 1

exchange 818 and 311

| 113 | 400 | 311 | 612 | 411 | 818 | 412 | 420 |
|---|---|---|---|---|---|---|---|

Copyright © 1998 by Hanan Samet

# QUICKSORT

- Key:
  1. break file into successive halves such that each half contains all keys greater than a certain value
  2. recursive application of this partitioning scheme to halves yields a sorted list once a partition has only one element
- Algorithm:
  1. pick an item in the array at random, say $x$
  2. scan the array of keys from the left until $a_i \geq x$ is found
  3. scan the array of keys from the right until $a_j \leq x$ is found
  4. exchange $a_i$ and $a_j$ and continue the scan and swap process until the two scans meet (note that $x$ itself may also be moved)
- Ex:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |

pick item 5 at random (i.e. 411) and scan starting at position 1

exchange 818 and 311

| 113 | 400 | 311 | 612 | 411 | 818 | 412 | 420 |
|---|---|---|---|---|---|---|---|

exchange 612 and 411

| 113 | 400 | 311 | 411 | 612 | 818 | 412 | 420 |
|---|---|---|---|---|---|---|---|

stop with $j = 4$ and $i = 5$

now, partition the two subhalves $(1, j)$ and $(i, 8)$…

Copyright © 1998 by Hanan Samet

# QUICKSORT

- Key:
  1. break file into successive halves such that each half contains all keys greater than a certain value
  2. recursive application of this partitioning scheme to halves yields a sorted list once a partition has only one element
- Algorithm:
  1. pick an item in the array at random, say $x$
  2. scan the array of keys from the left until $a_i \geq x$ is found
  3. scan the array of keys from the right until $a_j \leq x$ is found
  4. exchange $a_i$ and $a_j$ and continue the scan and swap process until the two scans meet (note that $x$ itself may also be moved)
- Ex:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |

pick item 5 at random (i.e. 411) and scan starting at position 1

exchange 818 and 311

| 113 | 400 | 311 | 612 | 411 | 818 | 412 | 420 |
|---|---|---|---|---|---|---|---|

exchange 612 and 411

| 113 | 400 | 311 | 411 | 612 | 818 | 412 | 420 |
|---|---|---|---|---|---|---|---|

stop with $j = 4$ and $i = 5$

now, partition the two subhalves $(1,j)$ and $(i, 8)$…

- Average $O(n \log_2 n)$: $\log_2 n$ passes to partition the file into singletons, each pass makes $n$ comparison operations

Copyright © 1998 by Hanan Samet

## DRAWBACKS OF QUICKSORT

1. Taking random elements may be overkill
   - solution: always work with first or last element in a sublist
   - problem: worst case of the algorithm arises when the file is initially in sorted order
   - solution: take a small subset and use its median as the partition value
2. Reduce stack requirements by processing the shortest partition first
3. Use insertion sort for small partitions
4. Not stable due to scan-and-swap process

Copyright © 1998 by Hanan Samet

# COMPUTING MEDIANS

- More generally, find the $k^{th}$ value in increasing order from a set containing $n$ values
- No need to sort the entire list
- Process partition containing entry $a_k$
- Algorithm:
  1. Partition set: $x$ = splitting value

$$\left.\begin{array}{l} a_h \le x \quad \forall h < i \\ a_h \ge x \quad \forall h > j \end{array}\right\} \; i > j$$

  2. if $k \ge i$, then partition right subhalf
     if $k \le j$, then partition left subhalf
     otherwise, $k^{th}$ value is $a_k$ since $j<k<i$

- Ex: want 6th largest element

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |

pick item 5 at random (i.e. 411):

| 113 | 400 | 311 | 411 | 612 | 818 | 412 | 420 |
|---|---|---|---|---|---|---|---|

$k = 6 \ge 5 = i$, so partition right half
use item 8 as the splitting value (i.e., 420):

| 420 | 412 | 818 | 612 |
|---|---|---|---|

$k = 6 = j$, so partition left half
use item 6 as the splitting value (i.e., 412)

| 412 | 420 |
|---|---|

420 is the 6th value in increasing order, since $k = 6 \ge i = 6$

Copyright © 1998 by Hanan Samet

# ANALYSIS OF MEDIAN COMPUTATION

1. Average $\log_2 n$ partitions
2. Only process one partition
3. $i^{\text{th}}$ partition has an average of $n/2^i$ elements
   - i.e., $n/2^i$ comparisons
4. Total average running time is

$$n + \sum_{i=1}^{\lceil \log_2 n \rceil} \frac{n}{2^i} = n + n \cdot \sum_{i=1}^{\lceil \log_2 n \rceil} \frac{1}{2^i} = n + n \cdot 1 = 2n$$

Copyright © 1998 by Hanan Samet

# RADIX-EXCHANGE SORTING

- Quicksort sorts a list on the basis of its values
- Can also partition on the basis of the bits (more generally the characters or digits) that make up the values
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Algorithm:
  1. test most significant bit and partition into two sublists (i.e., $b_{M-1}=0$ and $b_{M-1}=1$)
  2. recur on sublists testing successive bit positions
  3. terminate when no bits are left to test, or sublist is empty
- Ex:

| step | partition values | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | – | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |

Copyright © 1998 by Hanan Samet

# RADIX-EXCHANGE SORTING

- Quicksort sorts a list on the basis of its values
- Can also partition on the basis of the bits (more generally the characters or digits) that make up the values
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Algorithm:
  1. test most significant bit and partition into two sublists (i.e., $b_{M-1}$=0 and $b_{M-1}$=1)
  2. recur on sublists testing successive bit positions
  3. terminate when no bits are left to test, or sublist is empty
- Ex:

| step | partition values | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | – | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 9 | 512 | 113 | 400 | 420 | 412 | 411 | 311 | 612 | 818 |

Copyright © 1998 by Hanan Samet

# RADIX-EXCHANGE SORTING

- Quicksort sorts a list on the basis of its values
- Can also partition on the basis of the bits (more generally the characters or digits) that make up the values
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Algorithm:
  1. test most significant bit and partition into two sublists (i.e., $b_{M-1}$=0 and $b_{M-1}$=1)
  2. recur on sublists testing successive bit positions
  3. terminate when no bits are left to test, or sublist is empty
- Ex:

| step | partition values | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | – | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 9 | 512 | 113 | 400 | 420 | 412 | 411 | 311 | 612 | 818 |
| bit 8 | 256,768 | 113 | 400 | 420 | 412 | 411 | 311 | 612 | 818 |

Copyright © 1998 by Hanan Samet

# RADIX-EXCHANGE SORTING

- Quicksort sorts a list on the basis of its values
- Can also partition on the basis of the bits (more generally the characters or digits) that make up the values
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Algorithm:
  1. test most significant bit and partition into two sublists (i.e., $b_{M-1}$=0 and $b_{M-1}$=1)
  2. recur on sublists testing successive bit positions
  3. terminate when no bits are left to test, or sublist is empty
- Ex:

| step | partition values | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | – | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 9 | 512 | 113 | 400 | 420 | 412 | 411 | 311 | 612 | 818 |
| bit 8 | 256,768 | 113 | 400 | 420 | 412 | 411 | 311 | 612 | 818 |
| bit 7 | 384 | | 311 | 420 | 412 | 411 | 400 | | |

Copyright © 1998 by Hanan Samet

# RADIX-EXCHANGE SORTING

- Quicksort sorts a list on the basis of its values
- Can also partition on the basis of the bits (more generally the characters or digits) that make up the values
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Algorithm:
  1. test most significant bit and partition into two sublists (i.e., $b_{M-1}$=0 and $b_{M-1}$=1)
  2. recur on sublists testing successive bit positions
  3. terminate when no bits are left to test, or sublist is empty
- Ex:

| step | partition values | element number 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| start | – | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 9 | 512 | 113 | 400 | 420 | 412 | 411 | 311 | 612 | 818 |
| bit 8 | 256,768 | 113 | 400 | 420 | 412 | 411 | 311 | 612 | 818 |
| bit 7 | 384 | | 311 | 420 | 412 | 411 | 400 | | |
| bit 6 | 448 | | | 420 | 412 | 411 | 400 | | |

Copyright © 1998 by Hanan Samet

# RADIX-EXCHANGE SORTING

- Quicksort sorts a list on the basis of its values
- Can also partition on the basis of the bits (more generally the characters or digits) that make up the values
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Algorithm:
  1. test most significant bit and partition into two sublists (i.e., $b_{M-1}$=0 and $b_{M-1}$=1)
  2. recur on sublists testing successive bit positions
  3. terminate when no bits are left to test, or sublist is empty
- Ex:

| step | partition values | element number 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| start | – | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 9 | 512 | 113 | 400 | 420 | 412 | 411 | 311 \| | 612 | 818 |
| bit 8 | 256,768 | 113 \| | 400 | 420 | 412 | 411 | 311 \| | 612 \| | 818 |
| bit 7 | 384 | | 311 \| | 420 | 412 | 411 | 400 | | |
| bit 6 | 448 | | | 420 | 412 | 411 | 400 | | |
| bit 5 | 416 | | | 400 | 412 | 411 \| | 420 | | |

Copyright © 1998 by Hanan Samet

# RADIX-EXCHANGE SORTING

- Quicksort sorts a list on the basis of its values
- Can also partition on the basis of the bits (more generally the characters or digits) that make up the values
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Algorithm:
  1. test most significant bit and partition into two sublists (i.e., $b_{M-1}$=0 and $b_{M-1}$=1)
  2. recur on sublists testing successive bit positions
  3. terminate when no bits are left to test, or sublist is empty
- Ex:

| step | partition values | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | – | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 9 | 512 | 113 | 400 | 420 | 412 | 411 | 311 \| | 612 | 818 |
| bit 8 | 256,768 | 113 \| | 400 | 420 | 412 | 411 | 311 \| | 612 \| | 818 |
| bit 7 | 384 | | 311 \| | 420 | 412 | 411 | 400 | | |
| bit 6 | 448 | | | 420 | 412 | 411 | 400 | | |
| bit 5 | 416 | | | 400 | 412 | 411 \| | 420 | | |
| bit 4 | 400 | | | 400 | 412 | 411 | | | |

Copyright © 1998 by Hanan Samet

# RADIX-EXCHANGE SORTING

- Quicksort sorts a list on the basis of its values
- Can also partition on the basis of the bits (more generally the characters or digits) that make up the values
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Algorithm:
  1. test most significant bit and partition into two sublists (i.e., $b_{M-1}$=0 and $b_{M-1}$=1)
  2. recur on sublists testing successive bit positions
  3. terminate when no bits are left to test, or sublist is empty
- Ex:

| step | partition values | element number 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| start | – | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 9 | 512 | 113 | 400 | 420 | 412 | 411 | 311 \| | 612 | 818 |
| bit 8 | 256,768 | 113 \| | 400 | 420 | 412 | 411 | 311 \| | 612 \| | 818 |
| bit 7 | 384 | | 311 \| | 420 | 412 | 411 | 400 | | |
| bit 6 | 448 | | | 420 | 412 | 411 | 400 | | |
| bit 5 | 416 | | | 400 | 412 | 411 \| | 420 | | |
| bit 4 | 400 | | | 400 | 412 | 411 | | | |
| bit 3 | 408 | | | 400 \| | 412 | 411 | | | |

Copyright © 1998 by Hanan Samet

# RADIX-EXCHANGE SORTING

- Quicksort sorts a list on the basis of its values
- Can also partition on the basis of the bits (more generally the characters or digits) that make up the values
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Algorithm:
  1. test most significant bit and partition into two sublists (i.e., $b_{M-1}$=0 and $b_{M-1}$=1)
  2. recur on sublists testing successive bit positions
  3. terminate when no bits are left to test, or sublist is empty
- Ex:

| step | partition values | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | – | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 9 | 512 | 113 | 400 | 420 | 412 | 411 | 311 \| | 612 | 818 |
| bit 8 | 256,768 | 113 \| | 400 | 420 | 412 | 411 | 311 \| | 612 \| | 818 |
| bit 7 | 384 | | 311 \| | 420 | 412 | 411 | 400 | | |
| bit 6 | 448 | | | 420 | 412 | 411 | 400 | | |
| bit 5 | 416 | | | 400 | 412 | 411 \| | 420 | | |
| bit 4 | 400 | | | 400 | 412 | 411 | | | |
| bit 3 | 408 | | | 400 \| | 412 | 411 | | | |
| bit 2 | 412 | | | | 411 \| | 412 | | | |

- As in quicksort, not stable due to scan-and-swap process

Copyright © 1998 by Hanan Samet

# RADIX SORTING

- Radix-exchange sorting does not work too well when leading bits of the items being sorted are identical
- Alternative is to sort based on the trailing bits using a stable method such as distribution counting
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Ex: highlight elements where tested bit has value 1

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |

Copyright © 1998 by Hanan Samet

# RADIX SORTING

- Radix-exchange sorting does not work too well when leading bits of the items being sorted are identical
- Alternative is to sort based on the trailing bits using a stable method such as distribution counting
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Ex: highlight elements where tested bit has value 1

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 0 | 400 | 818 | 612 | 412 | 420 | 113 | 411 | 311 |

Copyright © 1998 by Hanan Samet

# RADIX SORTING

- Radix-exchange sorting does not work too well when leading bits of the items being sorted are identical
- Alternative is to sort based on the trailing bits using a stable method such as distribution counting
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Ex: highlight elements where tested bit has value 1

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 0 | 400 | 818 | 612 | 412 | 420 | 113 | 411 | 311 |
| bit 1 | 400 | 612 | 412 | 420 | 113 | 818 | 411 | 311 |

Copyright © 1998 by Hanan Samet

# RADIX SORTING

- Radix-exchange sorting does not work too well when leading bits of the items being sorted are identical
- Alternative is to sort based on the trailing bits using a stable method such as distribution counting
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Ex: highlight elements where tested bit has value 1

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 0 | 400 | 818 | 612 | 412 | 420 \| | 113 | 411 | 311 |
| bit 1 | 400 | 612 | 412 | 420 | 113 \| | 818 | 411 | 311 |
| bit 2 | 400 | 113 | 818 | 411 \| | 612 | 412 | 420 | 311 |

Copyright © 1998 by Hanan Samet

# RADIX SORTING

- Radix-exchange sorting does not work too well when leading bits of the items being sorted are identical
- Alternative is to sort based on the trailing bits using a stable method such as distribution counting
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Ex: highlight elements where tested bit has value 1

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 0 | 400 | 818 | 612 | 412 | 420 | 113 | 411 | 311 |
| bit 1 | 400 | 612 | 412 | 420 | 113 | 818 | 411 | 311 |
| bit 2 | 400 | 113 | 818 | 411 | 612 | 412 | 420 | 311 |
| bit 3 | 400 | 113 | 818 | 612 | 420 | 311 | 411 | 412 |

Copyright © 1998 by Hanan Samet

# RADIX SORTING

- Radix-exchange sorting does not work too well when leading bits of the items being sorted are identical
- Alternative is to sort based on the trailing bits using a stable method such as distribution counting
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Ex: highlight elements where tested bit has value 1

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 0 | 400 | 818 | 612 | 412 | 420 | 113 | 411 | 311 |
| bit 1 | 400 | 612 | 412 | 420 | 113 | 818 | 411 | 311 |
| bit 2 | 400 | 113 | 818 | 411 | 612 | 412 | 420 | 311 |
| bit 3 | 400 | 113 | 818 | 612 | 420 | 311 | 411 | 412 |
| bit 4 | 612 | 420 | 400 | 113 | 818 | 311 | 411 | 412 |

Copyright © 1998 by Hanan Samet

# RADIX SORTING

- Radix-exchange sorting does not work too well when leading bits of the items being sorted are identical
- Alternative is to sort based on the trailing bits using a stable method such as distribution counting
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Ex: highlight elements where tested bit has value 1

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 0 | 400 | 818 | 612 | 412 | 420 | 113 | 411 | 311 |
| bit 1 | 400 | 612 | 412 | 420 | 113 | 818 | 411 | 311 |
| bit 2 | 400 | 113 | 818 | 411 | 612 | 412 | 420 | 311 |
| bit 3 | 400 | 113 | 818 | 612 | 420 | 311 | 411 | 412 |
| bit 4 | 612 | 420 | 400 | 113 | 818 | 311 | 411 | 412 |
| bit 5 | 400 | 411 | 412 | 612 | 420 | 113 | 818 | 311 |

Copyright © 1998 by Hanan Samet

# RADIX SORTING

- Radix-exchange sorting does not work too well when leading bits of the items being sorted are identical
- Alternative is to sort based on the trailing bits using a stable method such as distribution counting
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Ex: highlight elements where tested bit has value 1

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 0 | 400 | 818 | 612 | 412 | 420 | 113 | 411 | 311 |
| bit 1 | 400 | 612 | 412 | 420 | 113 | 818 | 411 | 311 |
| bit 2 | 400 | 113 | 818 | 411 | 612 | 412 | 420 | 311 |
| bit 3 | 400 | 113 | 818 | 612 | 420 | 311 | 411 | 412 |
| bit 4 | 612 | 420 | 400 | 113 | 818 | 311 | 411 | 412 |
| bit 5 | 400 | 411 | 412 | 612 | 420 | 113 | 818 | 311 |
| bit 6 | 400 | 411 | 412 | 420 | 818 | 311 | 612 | 113 |

Copyright © 1998 by Hanan Samet

# RADIX SORTING

- Radix-exchange sorting does not work too well when leading bits of the items being sorted are identical
- Alternative is to sort based on the trailing bits using a stable method such as distribution counting
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Ex: highlight elements where tested bit has value 1

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 0 | 400 | 818 | 612 | 412 | 420 | 113 | 411 | 311 |
| bit 1 | 400 | 612 | 412 | 420 | 113 | 818 | 411 | 311 |
| bit 2 | 400 | 113 | 818 | 411 | 612 | 412 | 420 | 311 |
| bit 3 | 400 | 113 | 818 | 612 | 420 | 311 | 411 | 412 |
| bit 4 | 612 | 420 | 400 | 113 | 818 | 311 | 411 | 412 |
| bit 5 | 400 | 411 | 412 | 612 | 420 | 113 | 818 | 311 |
| bit 6 | 400 | 411 | 412 | 420 | 818 | 311 | 612 | 113 |
| bit 7 | 818 | 311 | 612 | 113 | 400 | 411 | 412 | 420 |

Copyright © 1998 by Hanan Samet

# RADIX SORTING

- Radix-exchange sorting does not work too well when leading bits of the items being sorted are identical
- Alternative is to sort based on the trailing bits using a stable method such as distribution counting
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Ex: highlight elements where tested bit has value 1

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 0 | 400 | 818 | 612 | 412 | 420 | 113 | 411 | 311 |
| bit 1 | 400 | 612 | 412 | 420 | 113 | 818 | 411 | 311 |
| bit 2 | 400 | 113 | 818 | 411 | 612 | 412 | 420 | 311 |
| bit 3 | 400 | 113 | 818 | 612 | 420 | 311 | 411 | 412 |
| bit 4 | 612 | 420 | 400 | 113 | 818 | 311 | 411 | 412 |
| bit 5 | 400 | 411 | 412 | 612 | 420 | 113 | 818 | 311 |
| bit 6 | 400 | 411 | 412 | 420 | 818 | 311 | 612 | 113 |
| bit 7 | 818 | 311 | 612 | 113 | 400 | 411 | 412 | 420 |
| bit 8 | 612 | 113 | 818 | 311 | 400 | 411 | 412 | 420 |

Copyright © 1998 by Hanan Samet

# RADIX SORTING

- Radix-exchange sorting does not work too well when leading bits of the items being sorted are identical
- Alternative is to sort based on the trailing bits using a stable method such as distribution counting
- Assume each value is represented by an $M$ bit number $b_{M-1}\ b_{M-2}\ \ldots\ b_1\ b_0$ (i.e., ranging from 0 to $2^M-1$)

| number | binary representation |
|---|---|
| 113 | 0 0 0 1 1 1 0 0 0 1 |
| 400 | 0 1 1 0 0 1 0 0 0 0 |
| 818 | 1 1 0 0 1 1 0 0 1 0 |
| 612 | 1 0 0 1 1 0 0 1 0 0 |
| 411 | 0 1 1 0 0 1 1 0 1 1 |
| 311 | 0 1 0 0 1 1 0 1 1 1 |
| 412 | 0 1 1 0 0 1 1 1 0 0 |
| 420 | 0 1 1 0 1 0 0 1 0 0 |

- Ex: highlight elements where tested bit has value 1

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| bit 0 | 400 | 818 | 612 | 412 | 420 | 113 | 411 | 311 |
| bit 1 | 400 | 612 | 412 | 420 | 113 | 818 | 411 | 311 |
| bit 2 | 400 | 113 | 818 | 411 | 612 | 412 | 420 | 311 |
| bit 3 | 400 | 113 | 818 | 612 | 420 | 311 | 411 | 412 |
| bit 4 | 612 | 420 | 400 | 113 | 818 | 311 | 411 | 412 |
| bit 5 | 400 | 411 | 412 | 612 | 420 | 113 | 818 | 311 |
| bit 6 | 400 | 411 | 412 | 420 | 818 | 311 | 612 | 113 |
| bit 7 | 818 | 311 | 612 | 113 | 400 | 411 | 412 | 420 |
| bit 8 | 612 | 113 | 818 | 311 | 400 | 411 | 412 | 420 |
| bit 9 | 113 | 311 | 400 | 411 | 412 | 420 | 612 | 818 |

Copyright © 1998 by Hanan Samet

# PROPERTIES OF RADIX SORTING

- Slow when the set of numbers being sorted is small while the range of values is large
- Can speed up by using larger groups of bits such as 4 or even the base 10 digits of the numbers
- Ex:

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |

Copyright © 1998 by Hanan Samet

# PROPERTIES OF RADIX SORTING

- Slow when the set of numbers being sorted is small while the range of values is large
- Can speed up by using larger groups of bits such as 4 or even the base 10 digits of the numbers
- Ex:

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| digit 0 | 400 | 420 \| | 411 | 311 \| | 612 | 412 \| | 113 \| | 818 |

Copyright © 1998 by Hanan Samet

# PROPERTIES OF RADIX SORTING

- Slow when the set of numbers being sorted is small while the range of values is large
- Can speed up by using larger groups of bits such as 4 or even the base 10 digits of the numbers
- Ex:

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| digit 0 | 400 | 420 \| | 411 | 311 \| | 612 | 412 \| | 113 \| | 818 |
| digit 1 | 400 \| | 411 | 311 | 612 | 412 | 113 | 818 \| | 420 |

Copyright © 1998 by Hanan Samet

# PROPERTIES OF RADIX SORTING

- Slow when the set of numbers being sorted is small while the range of values is large
- Can speed up by using larger groups of bits such as 4 or even the base 10 digits of the numbers
- Ex:

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| digit 0 | 400 | 420 | 411 | 311 | 612 | 412 | 113 | 818 |
| digit 1 | 400 | 411 | 311 | 612 | 412 | 113 | 818 | 420 |
| digit 2 | 113 | 311 | 400 | 411 | 412 | 420 | 612 | 818 |

Copyright © 1998 by Hanan Samet

# PROPERTIES OF RADIX SORTING

- Slow when the set of numbers being sorted is small while the range of values is large
- Can speed up by using larger groups of bits such as 4 or even the base 10 digits of the numbers
- Ex:

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| digit 0 | 400 | 420 | 411 | 311 | 612 | 412 | 113 | 818 |
| digit 1 | 400 | 411 | 311 | 612 | 412 | 113 | 818 | 420 |
| digit 2 | 113 | 311 | 400 | 411 | 412 | 420 | 612 | 818 |

- Radix sorting differs from radix-exchange sorting in that we can't use the scan-and-swap process to rearrange the sublist currently being processed since it is not stable
- Need space for the output list and the counters for sorting by distribution counting
- $O(n \log_2 M)$ time
- Analogous to techniques used to sort punched cards in prior times
- $M$ is the range of values being sorted

Copyright © 1998 by Hanan Samet

# MERGE SORTING

- Bottom-up quicksort
- Small partitions are repeatedly merged to yield larger lists
- Ex:

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |

Copyright © 1998 by Hanan Samet

## MERGE SORTING

- Bottom-up quicksort
- Small partitions are repeatedly merged to yield larger lists
- Ex:

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| 1 | 113 | 400 | 612 | 818 | 311 | 411 | 412 | 420 |

Copyright © 1998 by Hanan Samet

## MERGE SORTING

- Bottom-up quicksort
- Small partitions are repeatedly merged to yield larger lists
- Ex:

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| 1 | 113 | 400 | 612 | 818 | 311 | 411 | 412 | 420 |
| 2 | 113 | 400 | 612 | 818 | 311 | 411 | 412 | 420 |

Copyright © 1998 by Hanan Samet

# MERGE SORTING

- Bottom-up quicksort
- Small partitions are repeatedly merged to yield larger lists
- Ex:

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| 1 | 113 | 400 | 612 | 818 | 311 | 411 | 412 | 420 |
| 2 | 113 | 400 | 612 | 818 | 311 | 411 | 412 | 420 |
| 3 | 113 | 311 | 400 | 411 | 412 | 420 | 612 | 818 |

Copyright © 1998 by Hanan Samet

# MERGE SORTING

- Bottom-up quicksort
- Small partitions are repeatedly merged to yield larger lists
- Ex:

| step | element number | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| start | 113 | 400 | 818 | 612 | 411 | 311 | 412 | 420 |
| 1 | 113 | 400 | 612 | 818 | 311 | 411 | 412 | 420 |
| 2 | 113 | 400 | 612 | 818 | 311 | 411 | 412 | 420 |
| 3 | 113 | 311 | 400 | 411 | 412 | 420 | 612 | 818 |

- Observations:
  1. $O(n \log_2 n)$ time as $\log_2 n$ passes and $n$ items
  2. stable as it retains duplicate items and in same order
  3. not in-place as need room to store the output on each pass but can be reused
  4. can avoid space for output list by using linked allocation instead of an array for the lists
  5. similar to Shellsort with increment sequence $2^k$, …, 2, 1
     - must modify merge sorting not to use consecutive items
     - Shellsort usually uses insertion sort while for this increment sequence it can also be implemented with merging
       a. result is still not stable as the lists being merged are overlapping rather than consecutive
       b. process is not in-place because of merging
     - yields a version of Shellsort running in $O(n \log_2 n)$ time but needs extra space
  6. used in tape sorting
     - number of records is large and can't keep all in memory
     - merge sorting needs little memory as usually comparing one element against an adjacent one

Copyright © 1998 by Hanan Samet

# POLYPHASE MERGE

- Merge sorting can be sped up by starting with larger ordered sequences of records (termed *runs* or *suites*)
- Ex: 12 42 44 55 | 6 18 67 90 | 72 88 | 15 37 98

  Suite #1 Suite #2 Suite #3 Suite #4
- *k*–1 tapes with *s* suites apiece may be merged onto a remaining tape to form *s* suites on it
- Each tape contains a different number of suites initially:

| tape # | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| # of suites | 9 | 13 | 6 | 15 | 0 |

  After the first merge (onto tape 5):

| | | | | | |
|---|---|---|---|---|---|
| | 3 | 7 | 0 | 9 | 6 |

  Rewinding tapes 3 and 5 and merging onto tape 3 yields:

| | | | | | |
|---|---|---|---|---|---|
| | 0 | 4 | 3 | 6 | 3 |

- Working backwards leads to an optimal distribution of suites on the tapes:

| | Tape # | | | | | |
|---|---|---|---|---|---|---|
| Pass | 1 | 2 | 3 | 4 | 5 | |
| *n* | 1 | 0 | 0 | 0 | 0 | |
| *n*–1 | 0 | 1 | 1 | 1 | 1 | Because on the next pass the |
| *n*–2 | 1 | 0 | 2 | 2 | 2 | → maximum number of suites on |
| *n*–3 | 3 | 2 | 0 | 4 | 4 | tapes 2, 3, 4 or 5 is 1 |
| *n*–4 | 7 | 6 | 4 | 0 | 8 | |
| *n*–5 | 15 | 14 | 12 | 8 | 0 | Pass *i*–1 must create the largest |
| *n*–6 | 0 | 29 | 27 | 23 | 15 | number of suites on pass *i*, so pass |
| *n*–7 | 29 | 0 | 56 | 54 | 44 | *i*–1 must merge onto this tape |

- Each of the distribution patterns provides a starting point for polyphase merge — i.e., initially require 1, 4, 7, 13, 25, 49, 92, 181, … suites
- These numbers can be obtained formally using recurrence relations

Copyright © 1998 by Hanan Samet

# INITIAL SUITE DISTRIBUTION

- Performance of polyphase merge methods can be improved when the initial suites contain more records (implies less suites!)
  1. fill the working store (WS) to capacity
  2. when a record is output (*R*), it is replaced by the next record from the input (*N*)
     - if *N* > *R*, then *N* will eventually be output in same suite
     - if *N* < *R*, then *N* is part of new suite being formed in WS
- Summary: there are 2 suites in the working store!
- Ex: |WS| = 3 [ ] = elements of new suite

| memory | | | output |
|---|---|---|---|
| 1 | 2 | 3 | |
| 44 | 55 | 12 | 12 |
| 44 | 55 | 42 | 42 |
| 44 | 55 | 94 | 44 |
| [6] | 55 | 94 | 55 |
| [6] | 67 | 94 | 67 |
| [6] | [18] | 94 | 94 |
| [6] | [18] | [70] | end of suite |
| 6 | 18 | 70 | 6 |
| 98 | 18 | 70 | 18 |
| 98 | 82 | 70 | 70 |
| 98 | 82 | [37] | 82 |
| | . | | |
| | . | | |
| | . | | |

- Theorem: the average length of a suite is 2·|WS| (see Knuth)

Copyright © 1998 by Hanan Samet

## COMPARISON

- Merge sorting is a bottom-up variant of quicksort
- Can also have top-down merge sorting
  1. quicksort is partition-and-divide
  2. top-down merge sorting is divide-and-merge at start
  3. key to comparison is time at which recursion (≡*divide*) takes place
     - quicksort: at end
     - top-down merge sorting: at start
- Merge sorting is best for external sorting (i.e., on disk or tape)
- Radix methods are generally impractical as the range of values is usually large and thus many counters or many steps are needed (when sorting by distribution counting is used to reduce the number of steps)
- Linked storage lessens the impact of exchange operations
  1. in-place selection sort can be made stable
  2. quicksort and radix-exchange sorting need doubly-linked lists as they process the lists in both directions
  3. top-down merge sorting is impractical as need to know how many elements are in each sublist
- Heapsort and Shellsort are in-place (don't require extra space) while quicksort and radix-exchange sorting are also in-place but need space for the stack

Copyright © 1998 by Hanan Samet

## SPEED CONSIDERATIONS

- Quicksort is $n \log_2 n$ on the average but can be as bad as $O(n^2)$
- Heapsort and merge sorting are always $O(n \log_2 n)$
- Quicksort has fastest inner loop
- Merge sorting lies somewhere inbetween quicksort and heapsort
- Shellsort is comparable to merge sorting and is not sensitive to the data being in sorted order unlike quicksort
- When data set is small, use one of basic sorting methods
  1. insertion sort or selection sort are comparable
  2. insertion sort is preferable
     - easy to implement a stable in-place variant
     - selection sort always requires $n^2$ or $n^2/2$ operations
     - requires between $n$ or $n(n+1)/2$ operations
     - only $n$ operations when input is sorted in increasing order

Copyright © 1998 by Hanan Samet