Syrian Private University

Algorithms & Data Structure I

Instructor: Dr. Mouhib Alnoukari

The Role of Algorithms in Computing

Algorithms as a technology, introduction to algorithms

Algorithms as a Technology

Algorithms as a Technology

Suppose computers were infinitely fast and computer memory was free. Would you have any reason to study algorithms?

YES

You would still like to demonstrate that your solution method terminates and does so with the correct answer.

- Computers may be fast, but they are not infinitely fast.
- Memory may be inexpensive, but it is not free.
- Computing time is therefore a bounded resource, and so is space in memory.
- You should use these resources wisely, and algorithms that are efficient in terms of time or space will help you do so.

Algorithms Efficiency

- Different algorithms devised to solve the same problem often differ dramatically in their efficiency.
- These differences can be much more significant than differences due to hardware and software.

Algorithms Efficiency - Example

- Two algorithms for Sorting:

1. ***insertion sort:*** takes time roughly equal to $c_1 n^2$ (Where c_1 is a constant that does not depend on n).
2. ***merge sort:*** takes time roughly equal to

   c2 n lg n (Where c2 is another constant that does not depend on n).

Insertion sort typically has a smaller constant factor than merge sort, so that : $c_1 < c_2$

Insertion sort: $c_1 n.n$ merge sort: $c_2 n.\lg n$

Algorithms Efficiency - Example

Insertion sort: $c_1 n.n$

Factor of **n** in its running time

n=1000

n=1.000.000

merge sort: $c_2 n.\lg n$

Factor of **lg n** in its running time

lg n = 10

lg n = 20

Algorithms Efficiency - Example

Insertion sort: $c_1 n.n$

merge sort: $c_2 n.\lg n$

Sorting Array 10.000.000 number (80 MB)

Computer A (faster)

10 billion instruction/second !

2 n^2 instructions to sort n numbers

$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}}$$

20.000 s (# 5.5 hours)

Computer B (slower)

10 million instruction/second

50n lg n

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}}$$

1163 s (# 20 minutes)

17 times faster

Sorting Array 100.000.000 number (800 MB)

23 days

4 hours

Introduction to Algorithms

Overall Picture

- This course is **not** about:
  - Programming languages
  - Computer architecture
  - Software architecture
  - Software design and implementation principles
    - Issues concerning small and large scale programming
- We will only touch upon the theory of complexity and computability

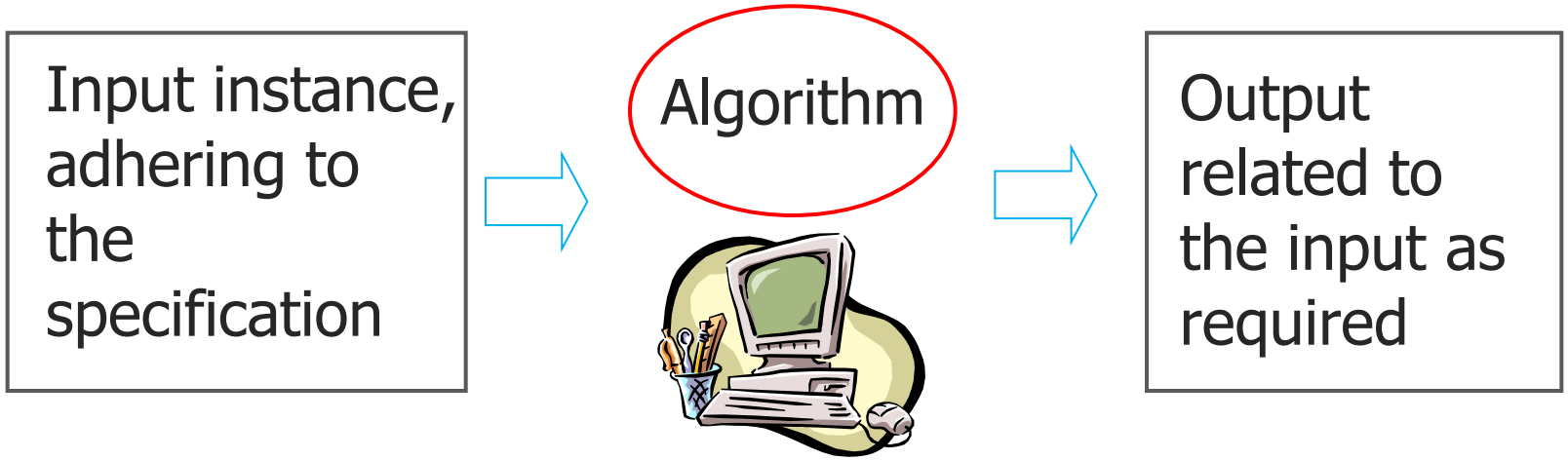
History

- Name: mathematician Mohammed al-Khowarizmi, in Latin became Algorismus
- First algorithm: Euclidean Algorithm, greatest common divisor, 400-300 B.C.
- 19th century – Charles Babbage, Ada Lovelace.
- 20th century – Alan Turing, Alonzo Church, John von Neumann

Algorithmic problem

- Infinite number of input *instances* satisfying the specification. For example:
  - A sorted, non-decreasing sequence of natural numbers. The sequence is of non-zero, finite length:
    - 1, 20, 908, 909, 100000, 1000000000.
    - 3.

Algorithmic Solution



- Algorithm describes actions on the input instance
- Infinitely many correct algorithms for the same algorithmic problem

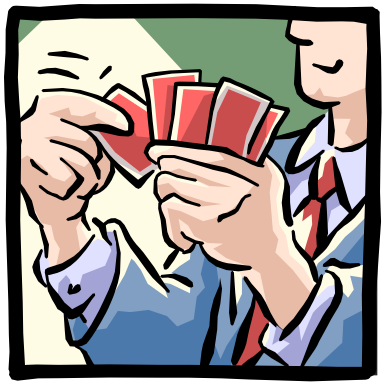
Example: Sorting

INPUT

sequence of numbers

$a_1, a_2, a_3, \ldots, a_n$

2 5 4 10 7



OUTPUT

a permutation of the sequence of numbers

$b_1, b_2, b_3, \ldots, b_n$

2 4 5 7 10

Correctness

For any given input the algorithm halts with the output:

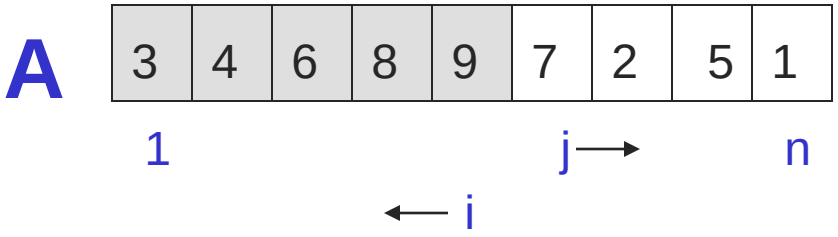
- $b_1 < b_2 < b_3 < \ldots < b_n$
- $b_1, b_2, b_3, \ldots, b_n$ is a permutation of $a_1, a_2, a_3, \ldots, a_n$

Running time

Depends on

- number of elements (n)
- how (partially) sorted they are
- algorithm

Insertion Sort



Strategy

- Start "empty handed"
- Insert a card in the right position of the already sorted hand
- Continue until all cards are inserted/sorted

```
for j=2 to length(A)
    do key=A[j]
    "insert A[j] into the
    sorted sequence A[1..j-1]"
       i=j-1
       while i>0 and A[i]>key
          do A[i+1]=A[i]
             i--
       A[i+1]:=key
```

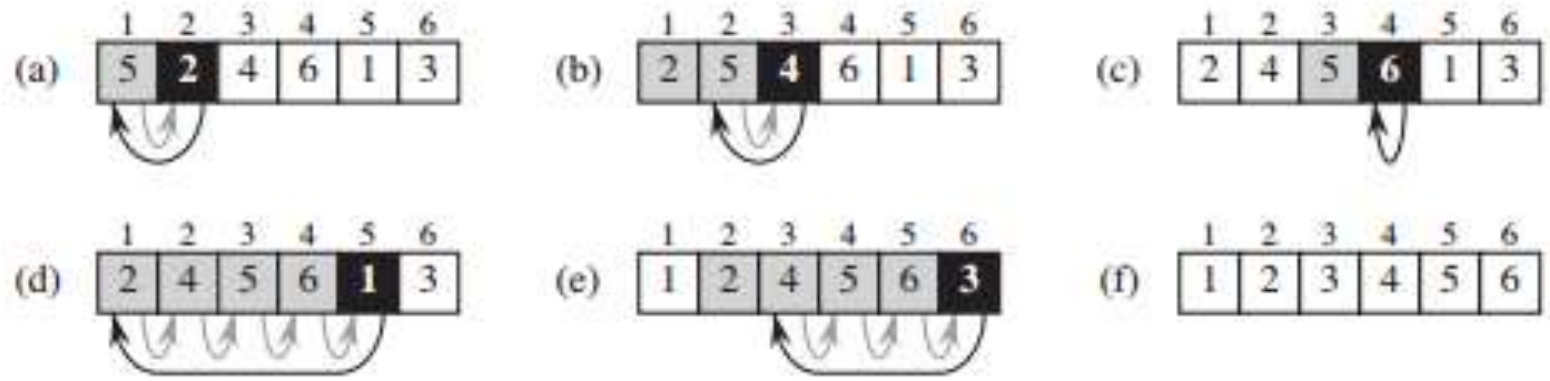
Insertion Sort - Example

Example

A

5	2	4	6	1	3

```
for j=2 to length(A)
   do key=A[j]
   "insert A[j] into the
   sorted sequence A[1..j-1]"
     i=j-1
     while i>0 and A[i]>key
        do A[i+1]=A[i]
           i--
     A[i+1]:=key
```



Analysis of Algorithms

- Efficiency:
  - Running time
  - Space used
- Efficiency as a function of input size:
  - Number of data elements (numbers, points)
  - A number of bits in an input number

The RAM model

- Very important to choose the level of detail.
- The RAM model:
  - Instructions are executed one after another, with no concurrent operations.
  - Instructions (each taking constant time):
    - Arithmetic (add, subtract, multiply, etc.)
    - Data movement (assign)
    - Control (branch, subroutine call, return)
  - Data types – integers and floats

Analysis of Insertion Sort

- Time to compute the **running time** as a function of the **input size**

	cost (time)	**times**
for j=2 **to** *length*(A)	c_1	n
do key=A[j]	c_2	n-1
"insert A[j] into the sorted sequence A[1..j-1]"	0	n-1
i=j-1	c_3	n-1
while i>0 **and** A[i]>key	c_4	$\sum_{j=2}^{n} t_j$
do A[i+1]=A[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i--	c_6	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1]:=key	c_7	n-1

t_j: number of times the while loop is executed for that value of j.

Ci: a constant denotes the execution time of the ith line.

input size: number of items in the input.

running time: the number of primitive operations or "steps" executed.

Analysis of Insertion Sort – Best Case

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$
$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

T (n) : the running time of Insertion Sort

Best case: elements already sorted → $t_j=1$,

For each j = 2,3 … n, we then find that A[i] <= key in line 5 when i has its initial value of j - 1. Thus tj = 1 for j = 2,3 … n, and the best-case running time is:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

T(n) = a n + b → *linear* time

Analysis of Insertion Sort – Worst Case

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$
$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

T (n) : the running time of Insertion Sort

Worst case: elements in reverse sorted order → $t_j = j$,

We must compare each element A[j] with each element in the entire sorted subarray A [1..j-1] → $t_j = j$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

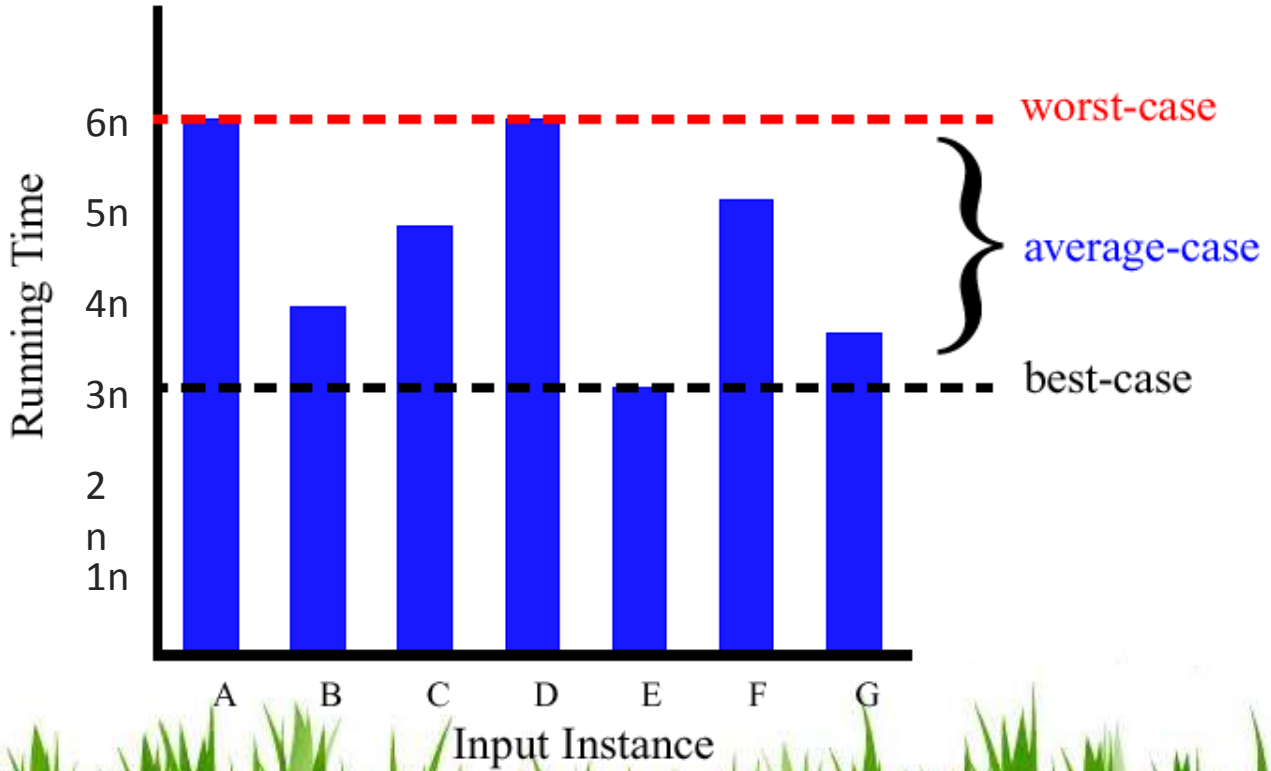
T(n) = a n^2 + b n + c → *quadratic* function

Best/Worst/Average Case

- **Best case**: elements already sorted $\rightarrow t_j=1$, running time = $f(n)$, i.e., *linear* time.
- **Worst case**: elements are sorted in inverse order
  $\rightarrow t_j=j$, running time = $f(n^2)$, i.e., *quadratic* time
- **Average case**: $t_j=j/2$, running time = $f(n^2)$, i.e., *quadratic* time

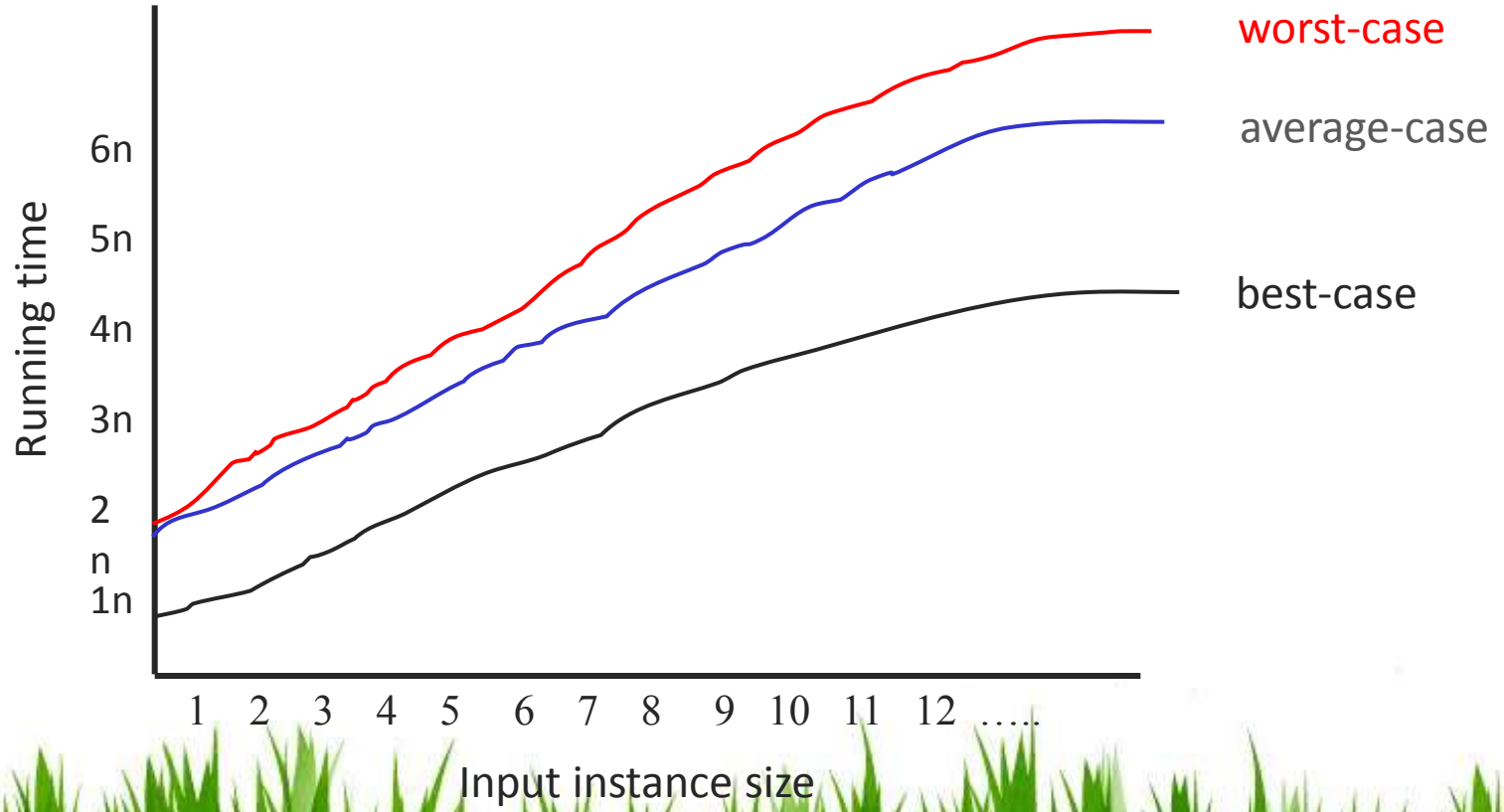
Best/Worst/Average Case (2)

– For a specific size of input *n*, investigate running times for different input instances:



Best/Worst/Average Case (3)

– For inputs of all sizes:



Best/Worst/Average Case (4)

- **Worst case** is usually used:
  - It is an upper-bound and in certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance
  - For some algorithms **worst case** occurs fairly often
  - The **average case** is often as bad as the **worst case**
  - Finding the **average case** can be very difficult

That's it?

- Is **insertion sort** the best approach to sorting?
- Alternative strategy based on divide and conquer
- MergeSort
  - sorting the numbers <4, 1, 3, 9> is split into
  - sorting <4, 1> and <3, 9> and
  - merging the results
  - Running time *f(n log n)*

Divide and Conquer

Merge Sort

Divide and Conquer

- *Divide and conquer* method for algorithm design:
  - **Divide**: If the input size is too large to deal with in a straightforward manner, divide the problem into two or more disjoint subproblems
  - **Conquer**: Use divide and conquer recursively to solve the subproblems
  - **Combine**: Take the solutions to the subproblems and "merge" these solutions into a solution for the original problem

MergeSort: Algorithm

- **Divide**: If *S* has at least two elements (nothing needs to be done if *S* has zero or one elements), remove all the elements from *S* and put them into two sequences, S_1 and S_2 , each containing about half of the elements of S. (i.e. S_1 contains the first $\lceil n/2 \rceil$ elements and S_2 contains the remaining $\lfloor n/2 \rfloor$ elements.
- **Conquer**: Sort sequences S_1 and S_2 using MergeSort.
- **Combine**: Put back the elements into *S* by merging the sorted sequences S_1 and S_2 into one sorted sequence

Merge Sort: Algorithm

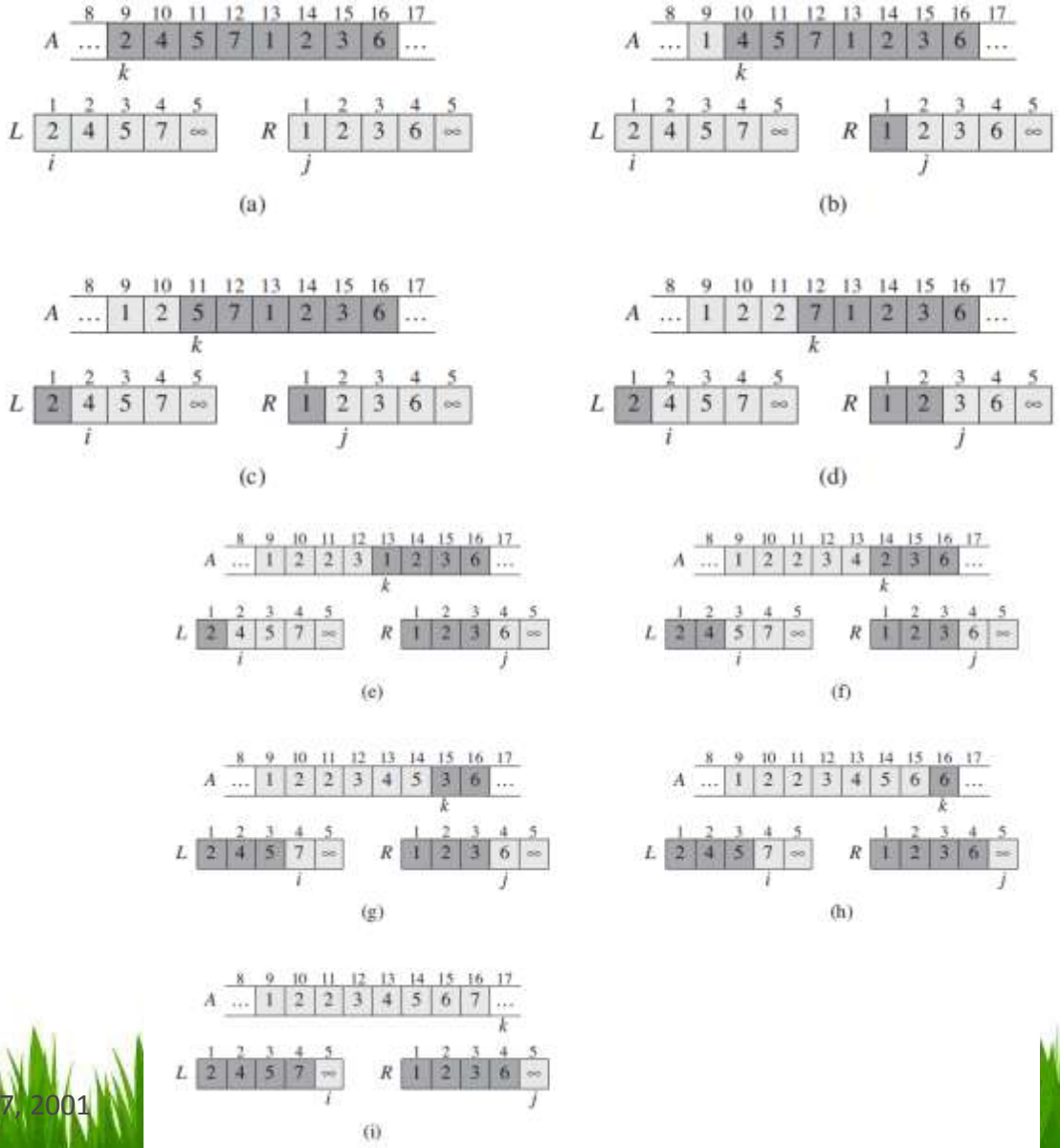
```
Merge-Sort(A, p, r)
    if p < r then
        q←(p+r)/2
        Merge-Sort(A, p, q)
        Merge-Sort(A, q+1, r)
        Merge(A, p, q, r)
```

```
Merge(A, p, q, r)
   Take the smallest of the two top most elements of
sequences A[p..q] and A[q+1..r] (they are in sorted
order) and put into the resulting sequence. Repeat
this, until both sequences are empty. Copy the
resulting sequence into A[p..r].
A: An array,
p, q, r: indices into the array where p<=q<r.
```

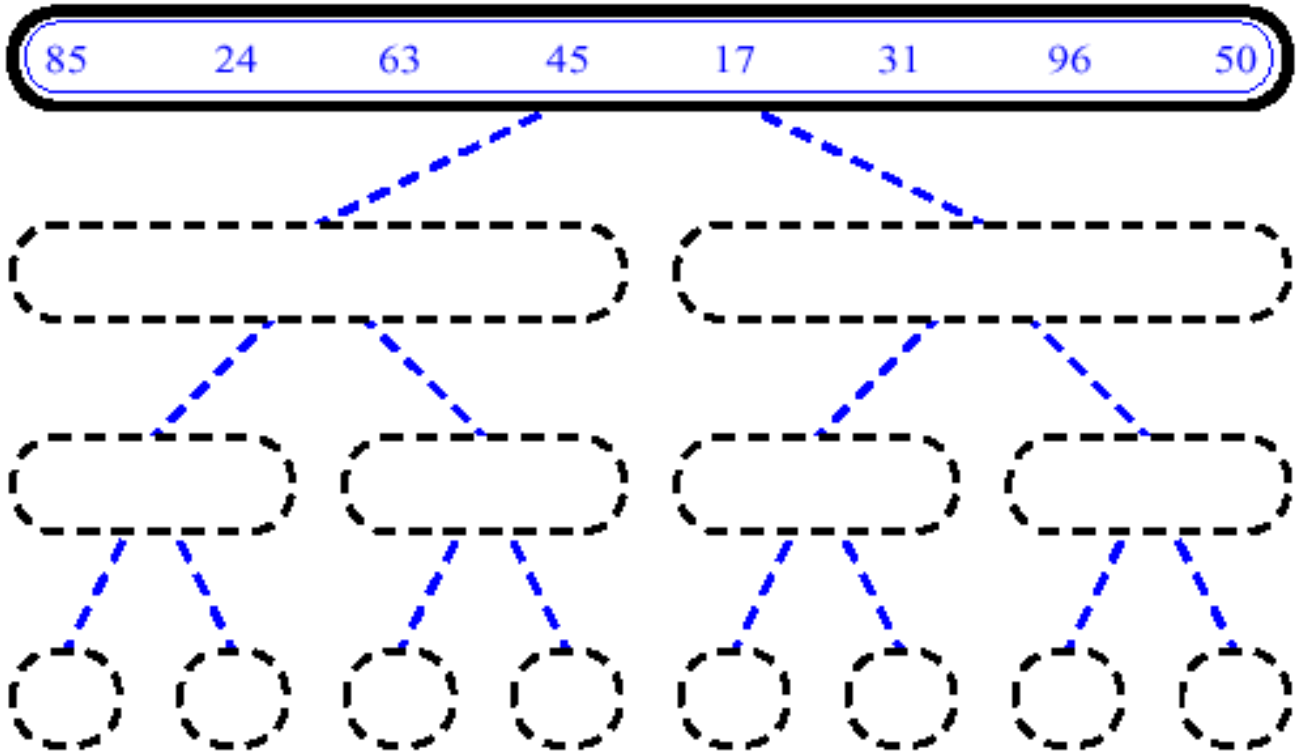
Merge Sort: Algorithm

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

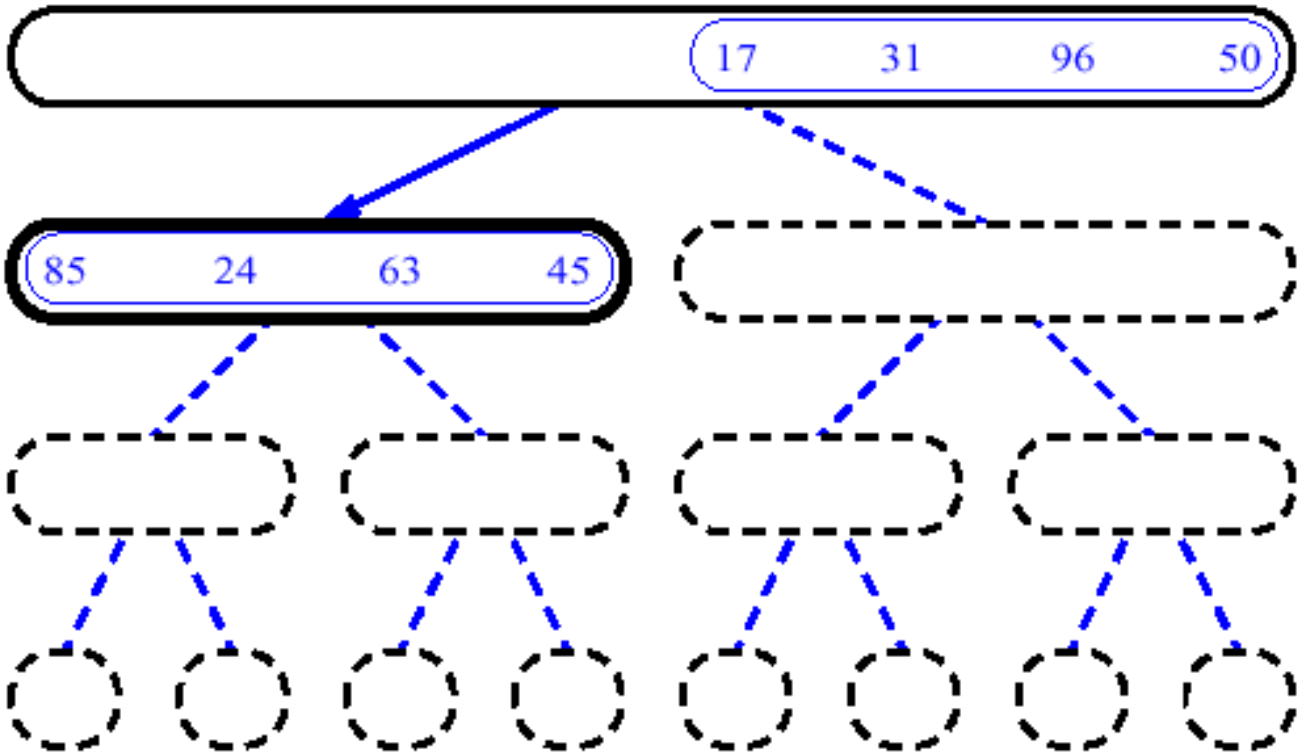
Merge Sort: Algorithm



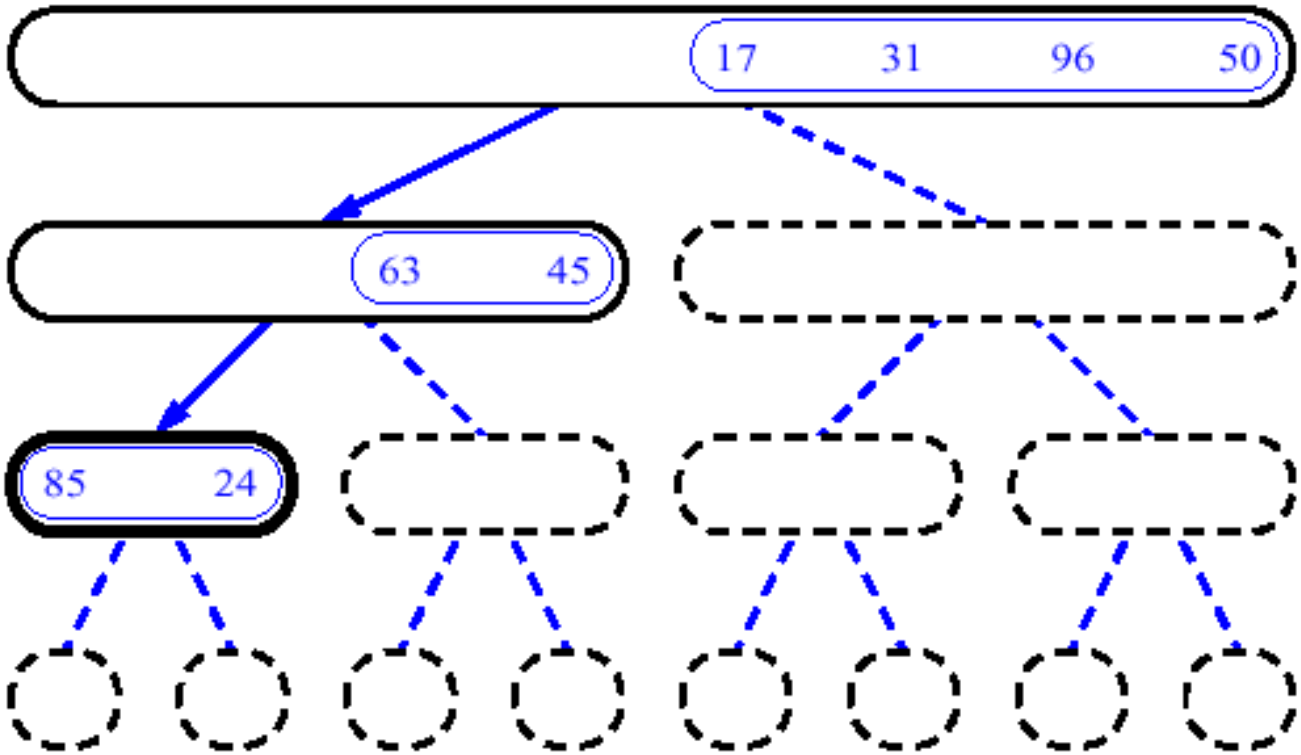
MergeSort (Example) - 1



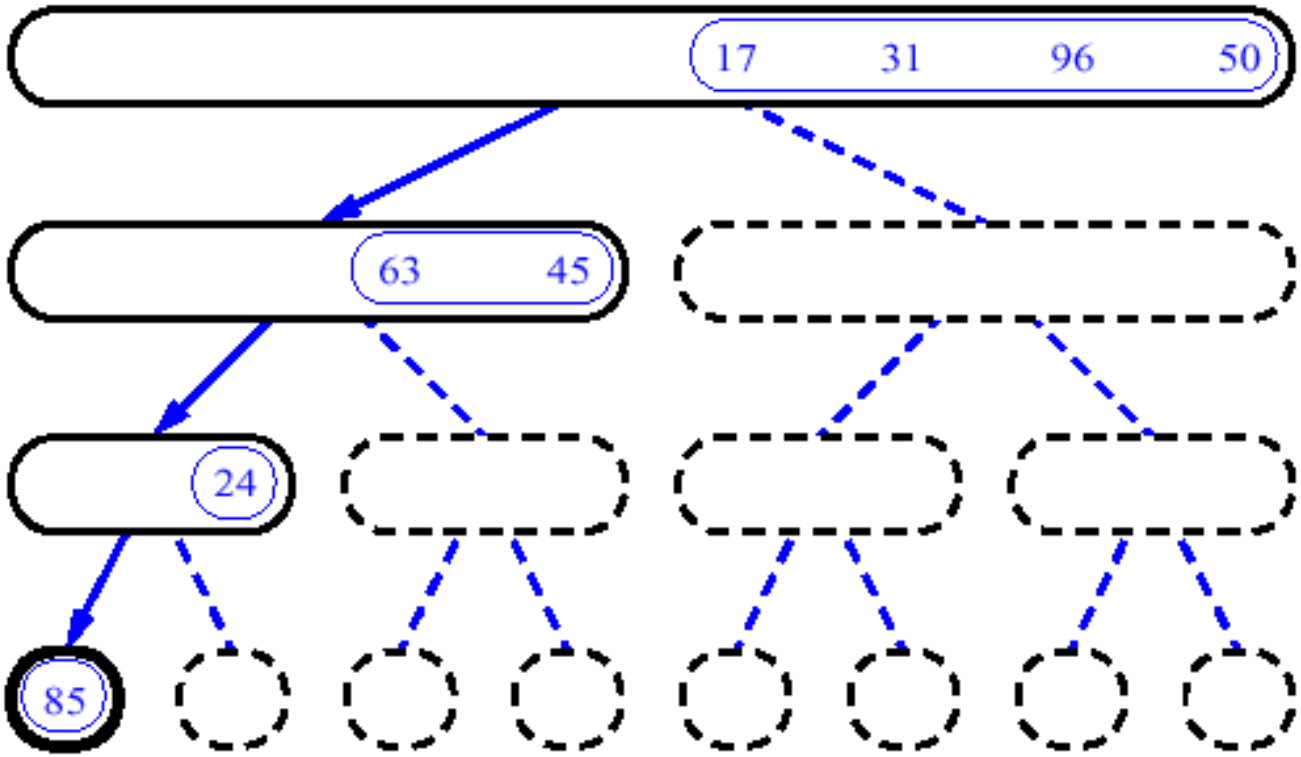
MergeSort (Example) - 2



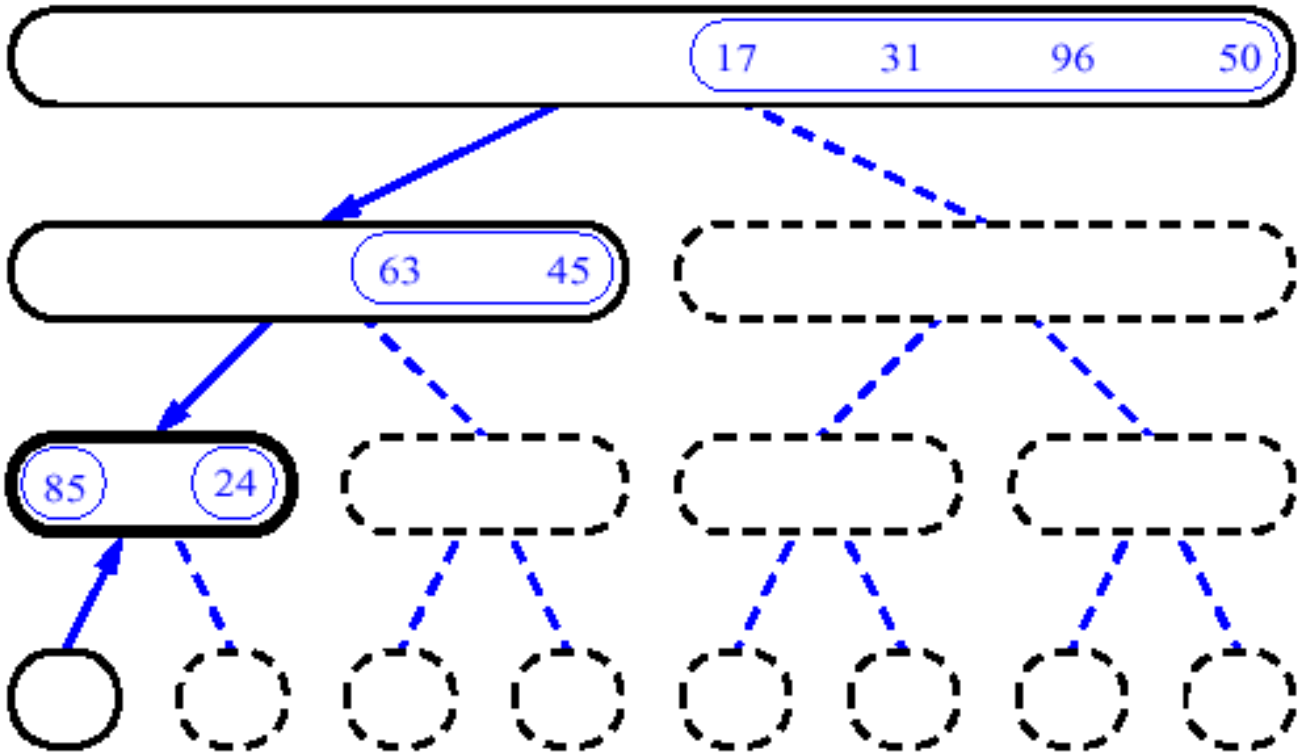
MergeSort (Example) - 3



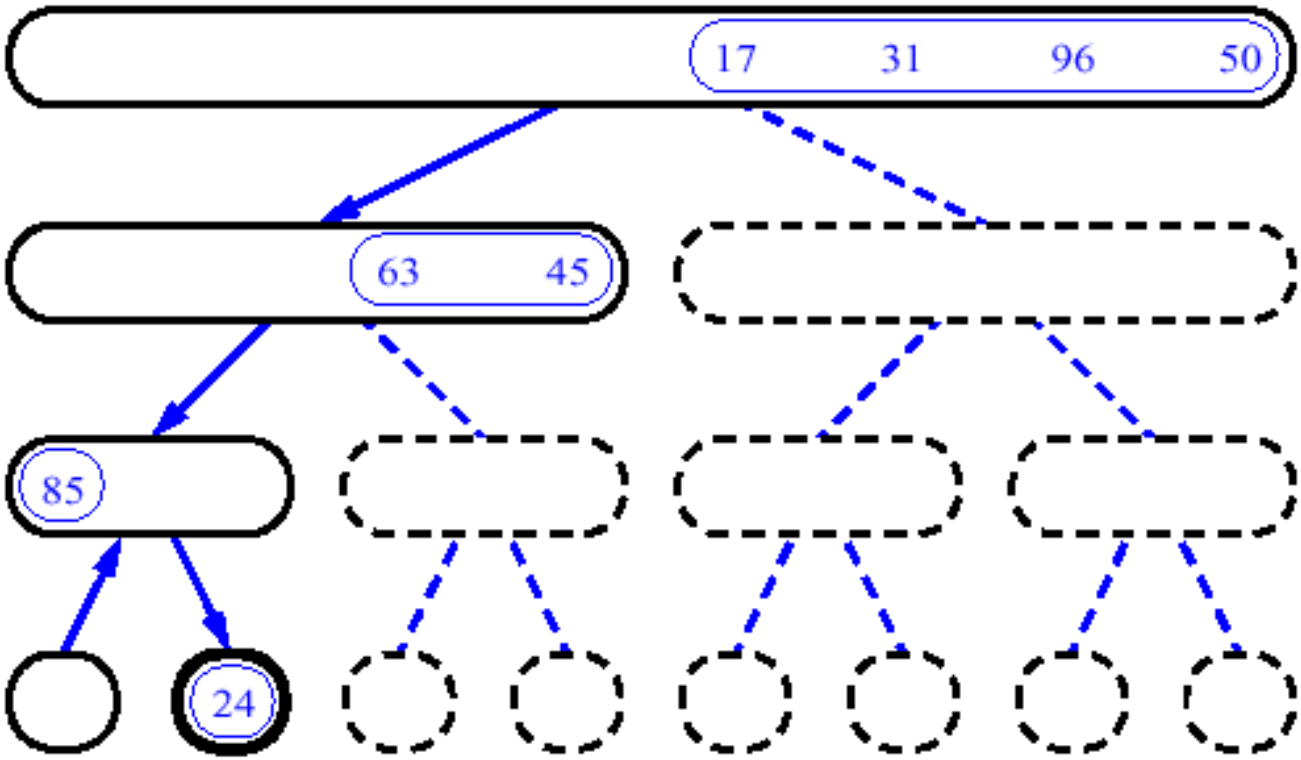
MergeSort (Example) - 4



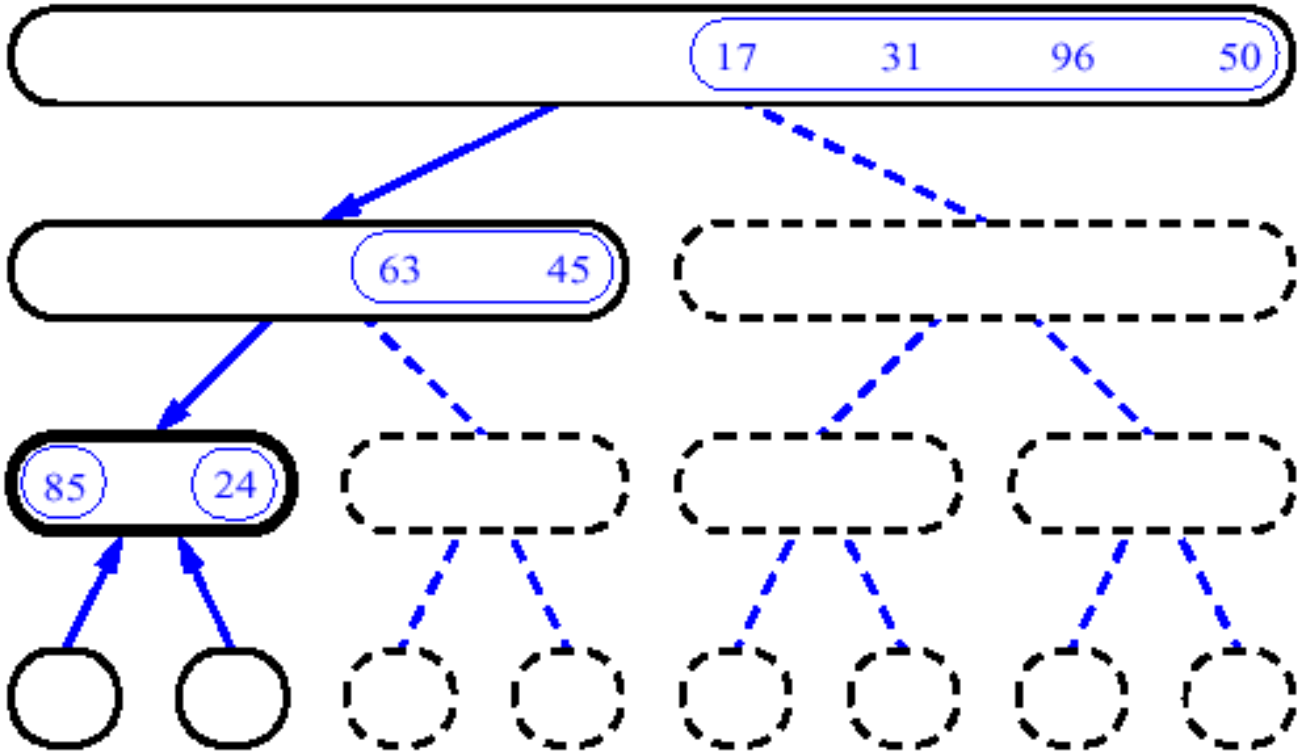
MergeSort (Example) - 5



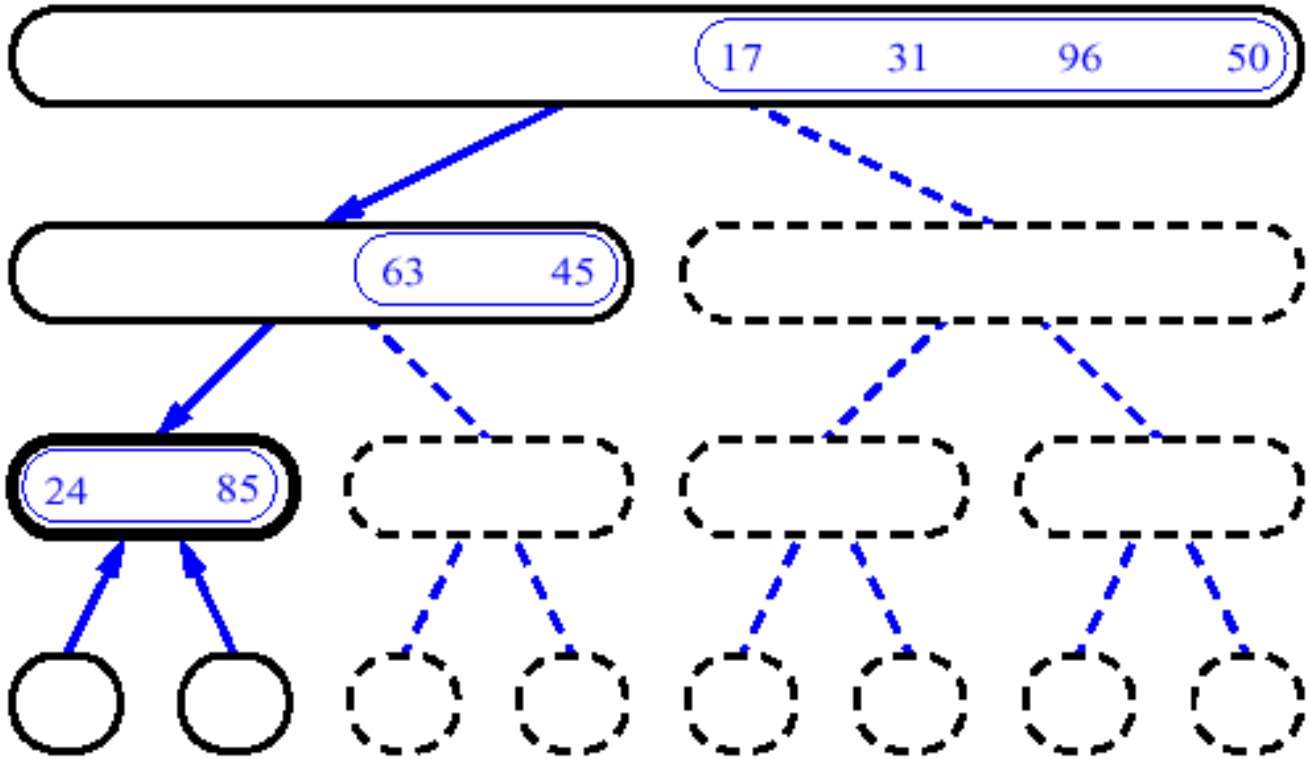
MergeSort (Example) - 6



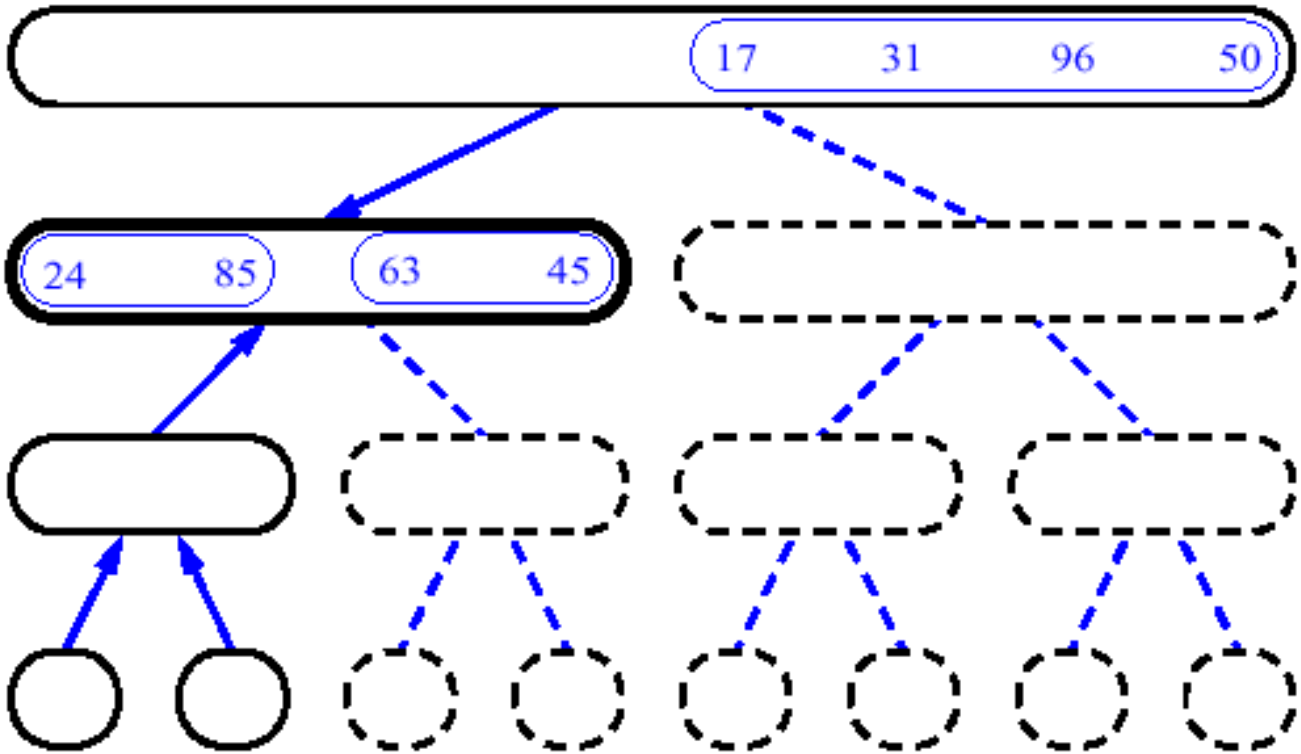
MergeSort (Example) - 7



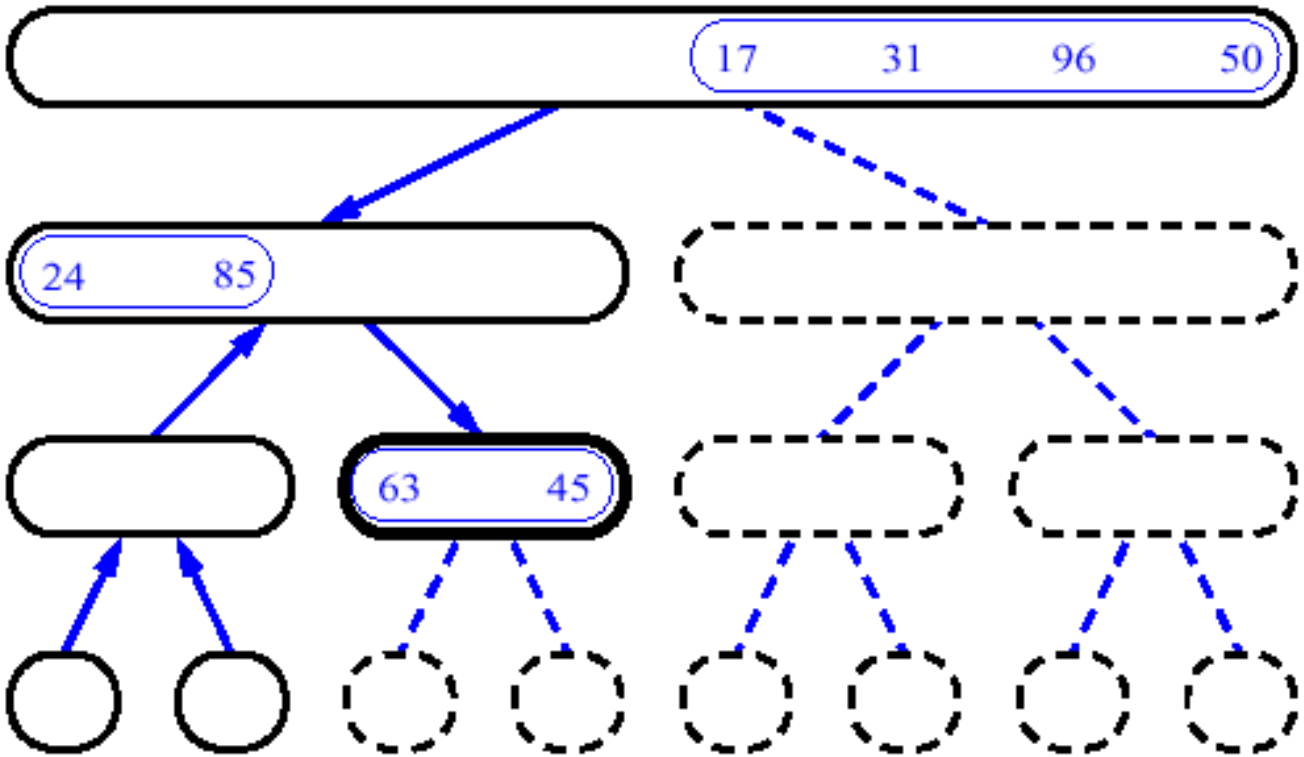
MergeSort (Example) - 8



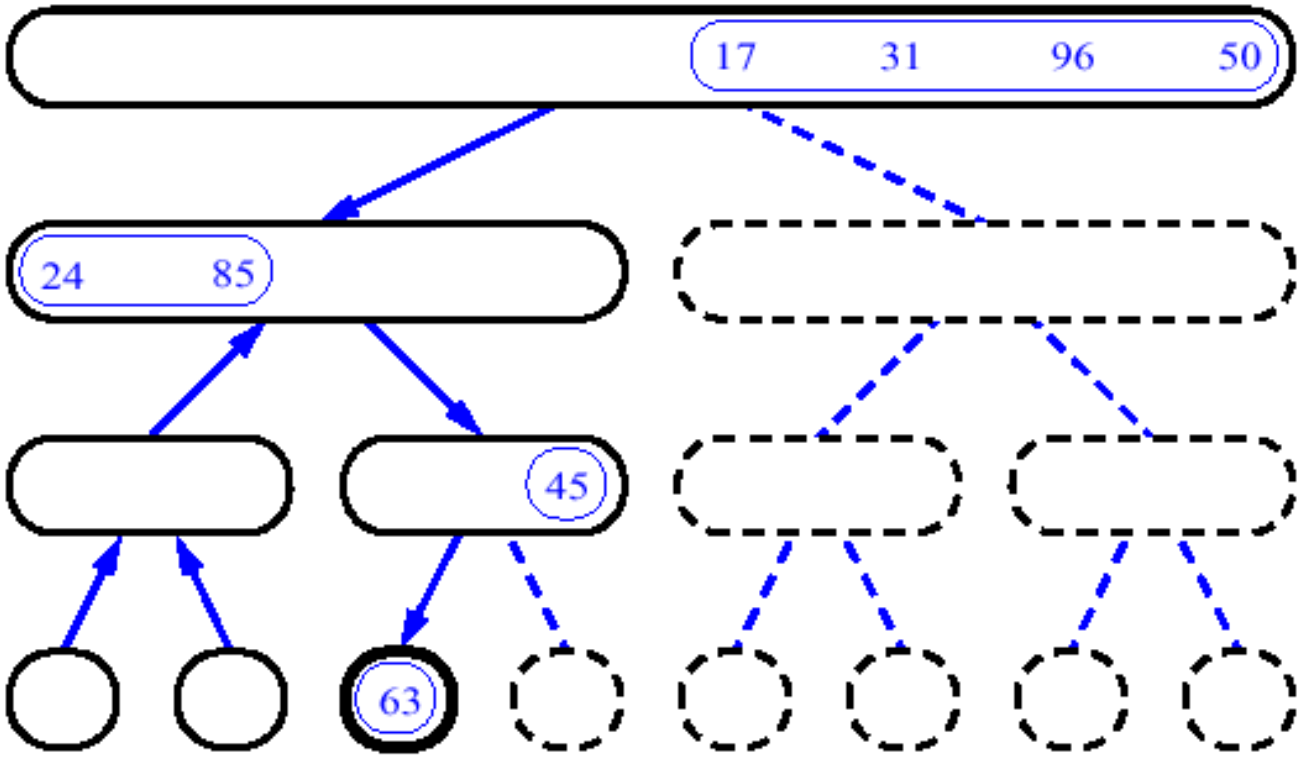
MergeSort (Example) - 9



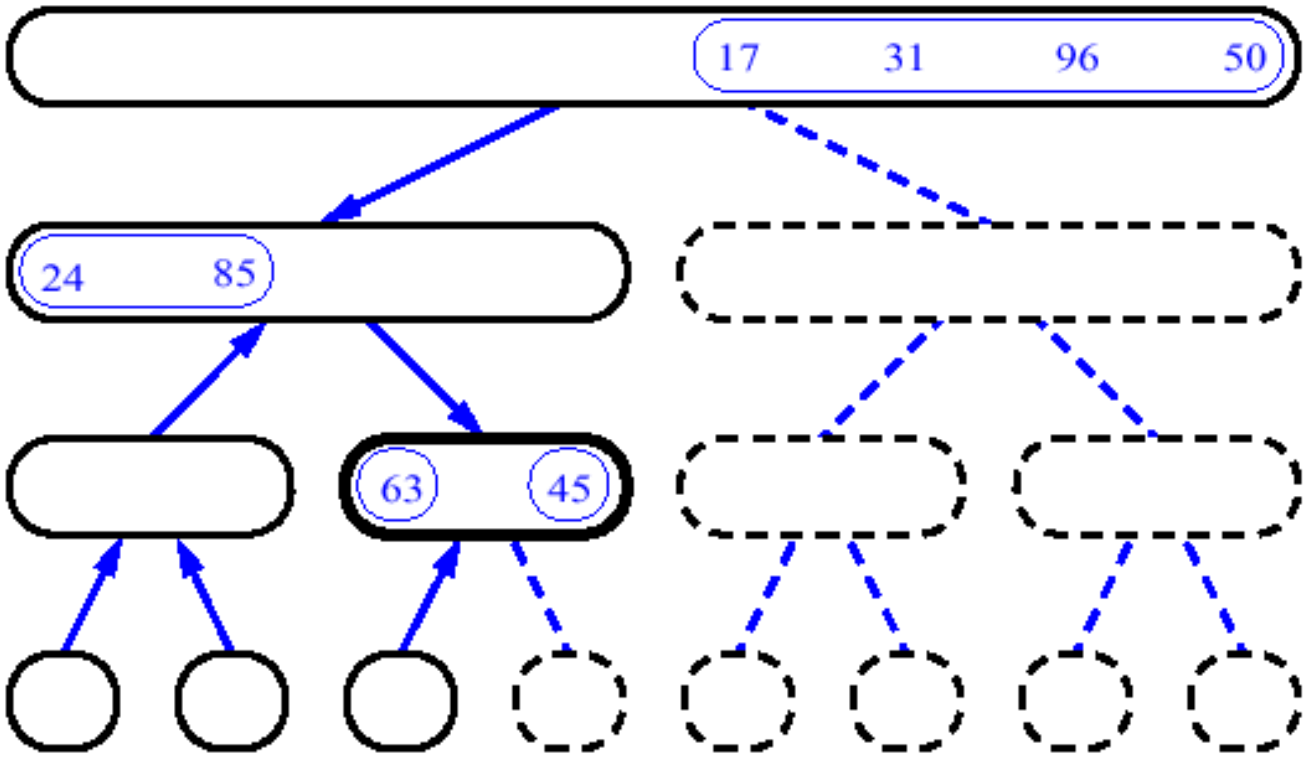
MergeSort (Example) - 10



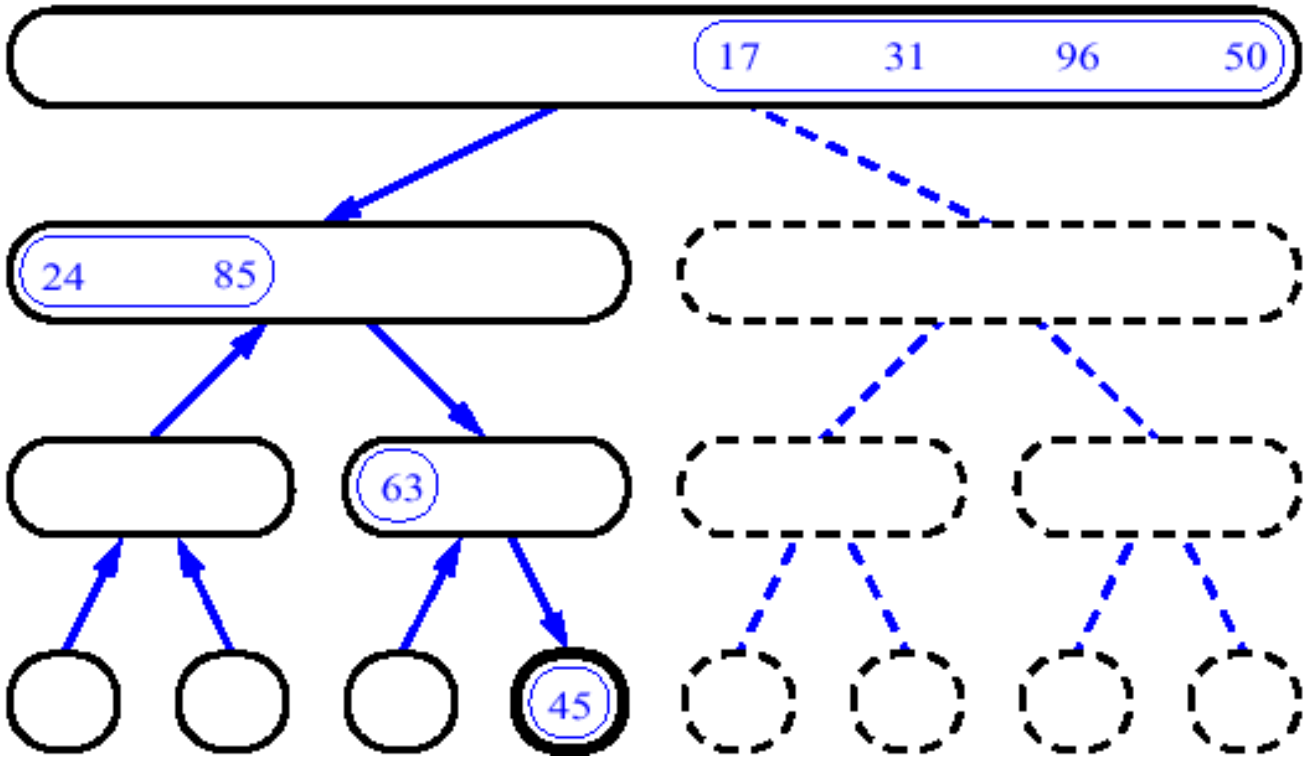
MergeSort (Example) - 11



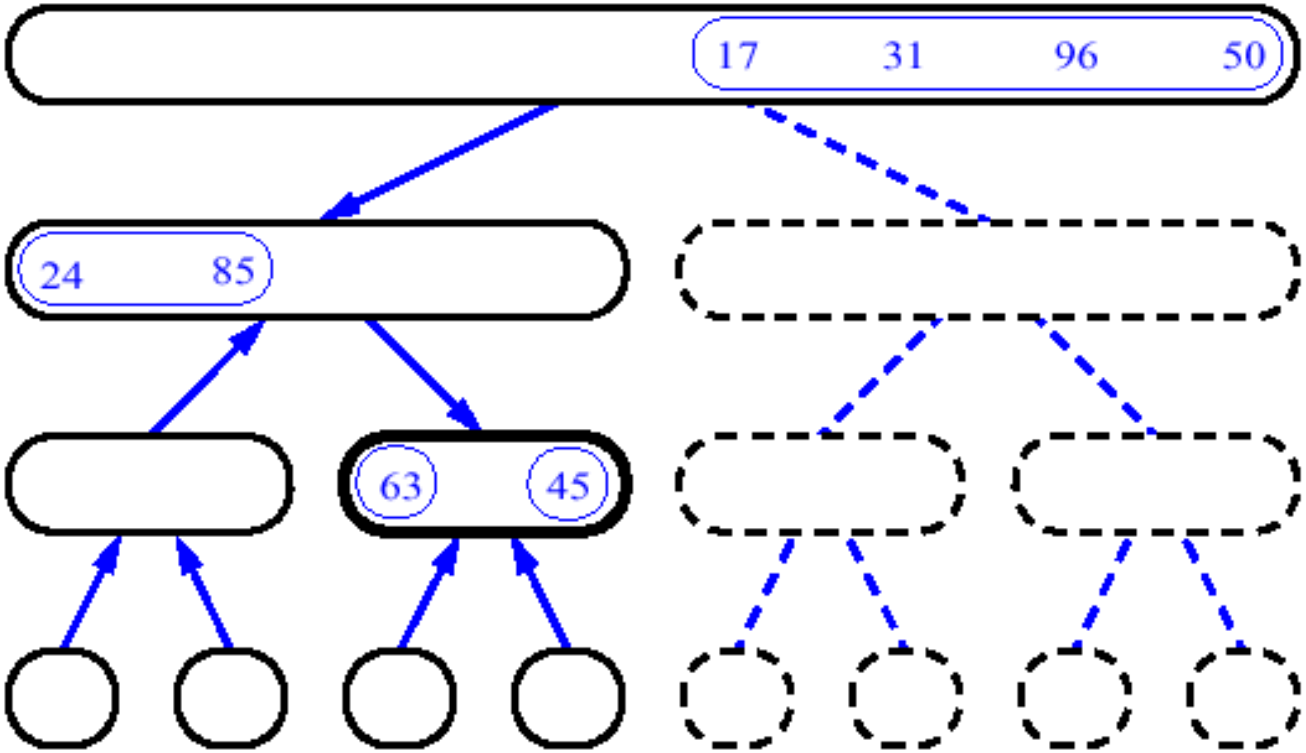
MergeSort (Example) - 12



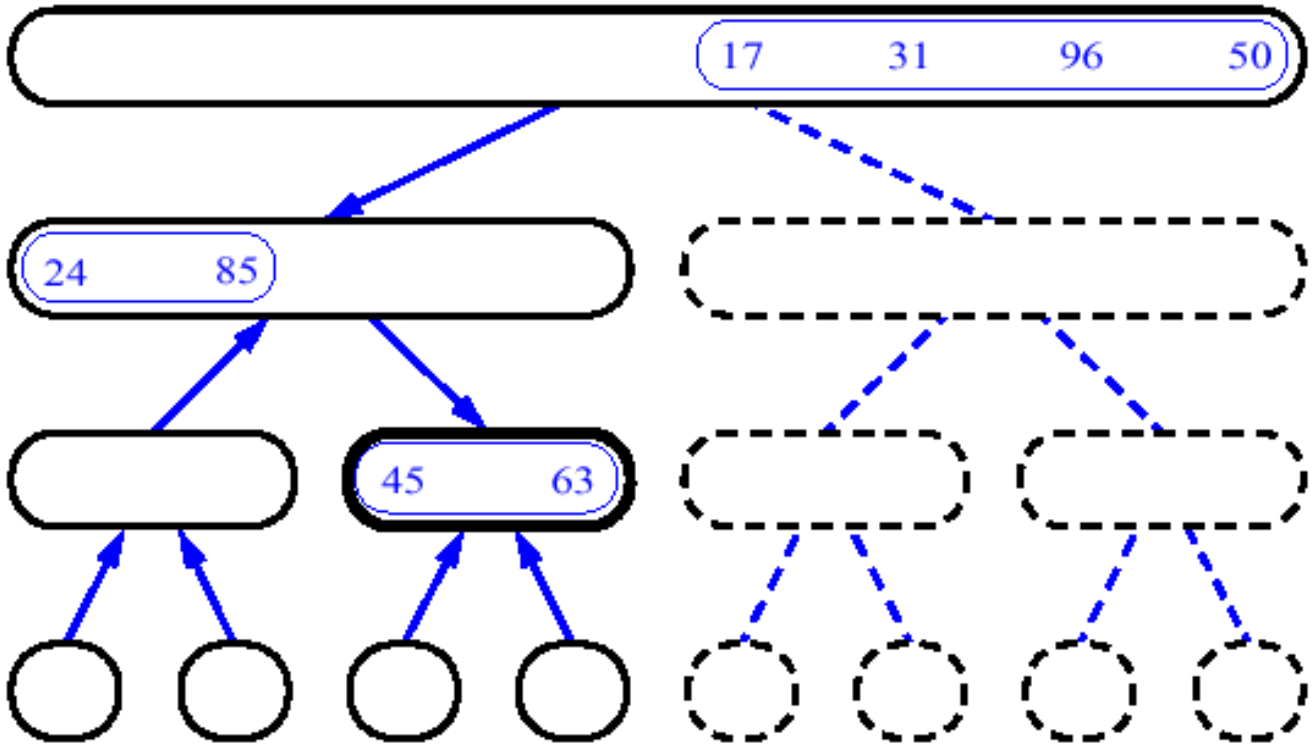
MergeSort (Example) - 13



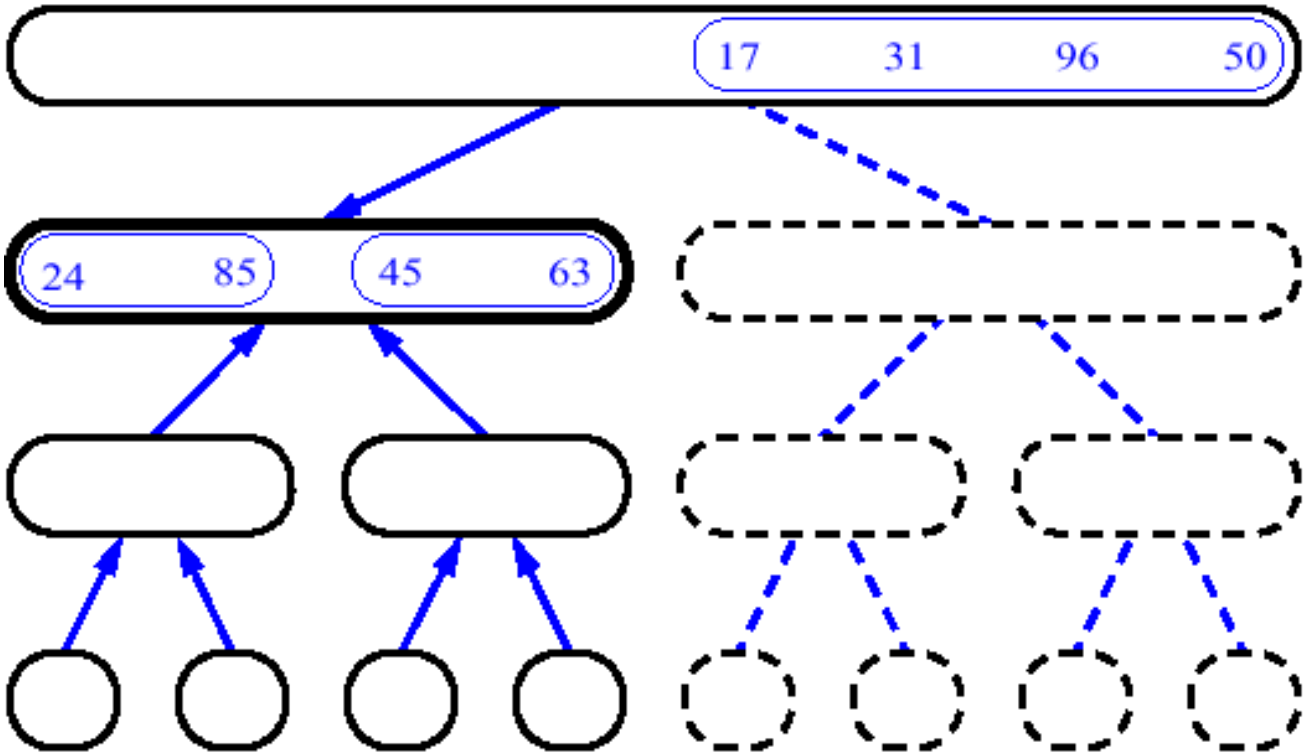
MergeSort (Example) - 14



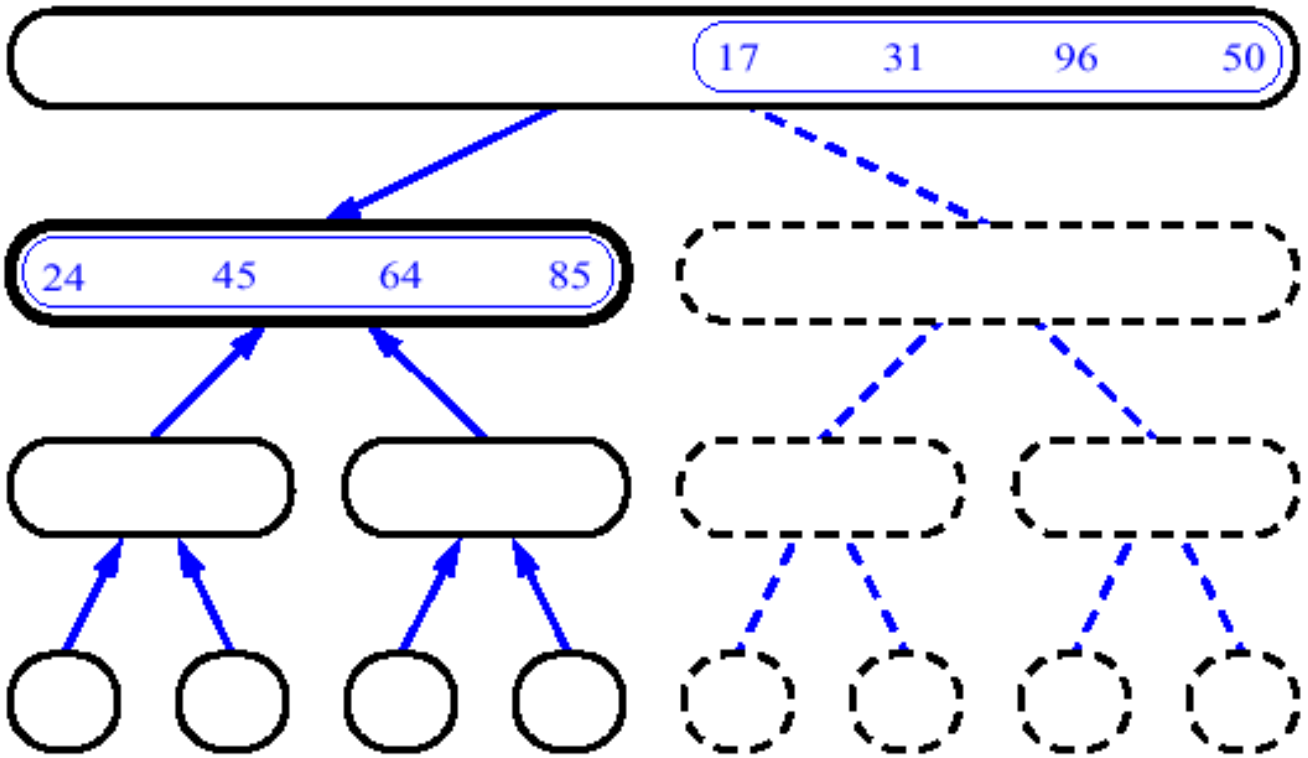
MergeSort (Example) - 15



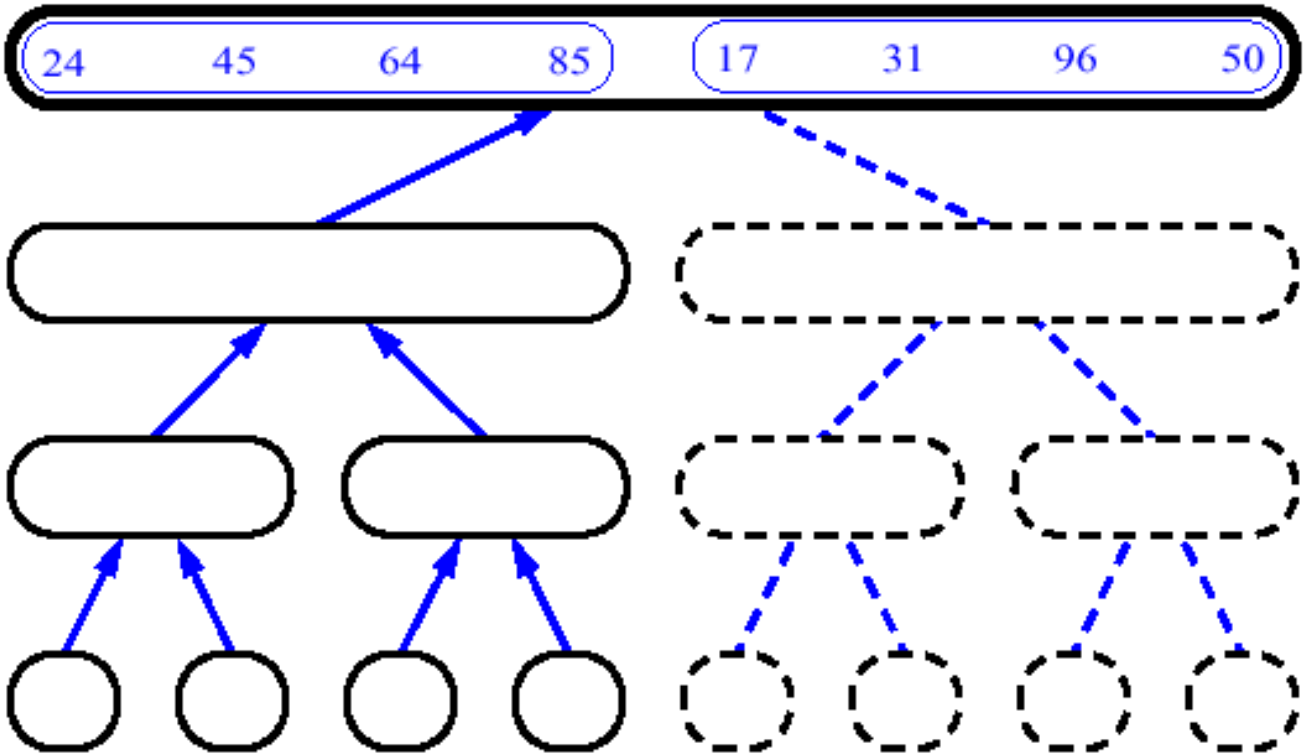
MergeSort (Example) - 16



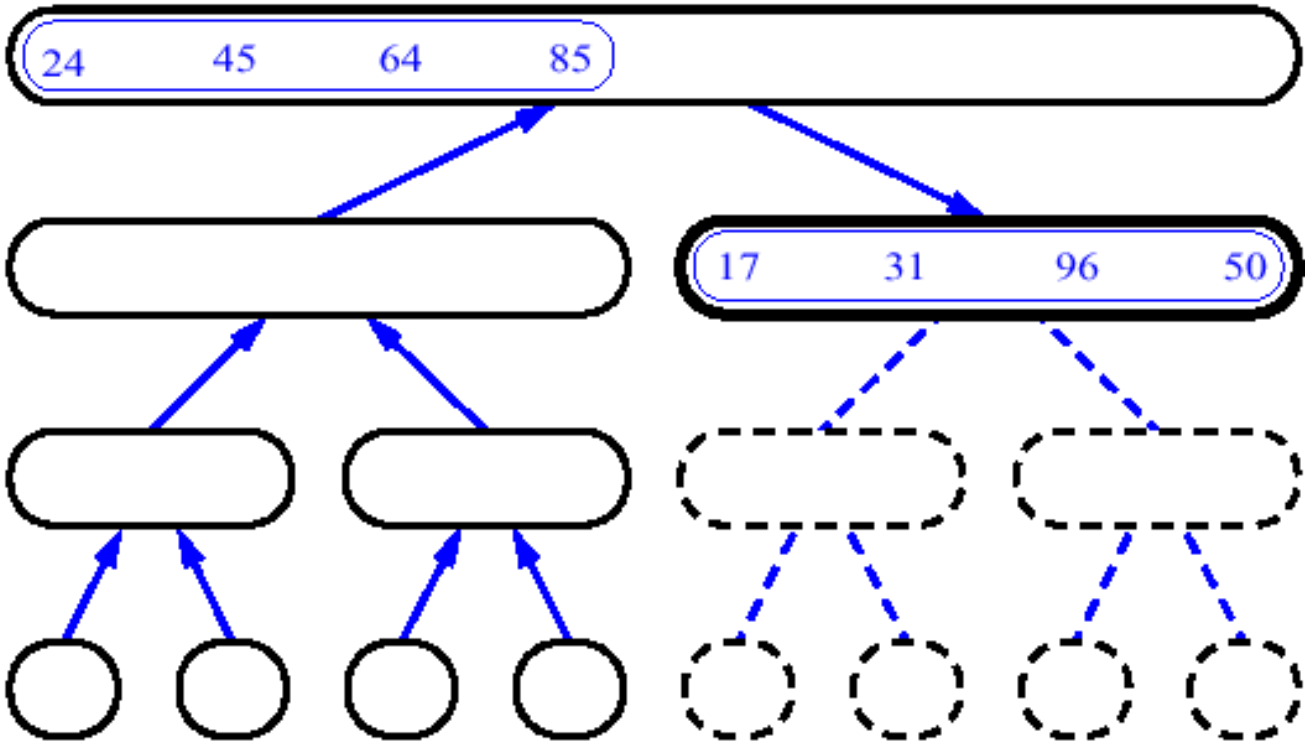
MergeSort (Example) - 17



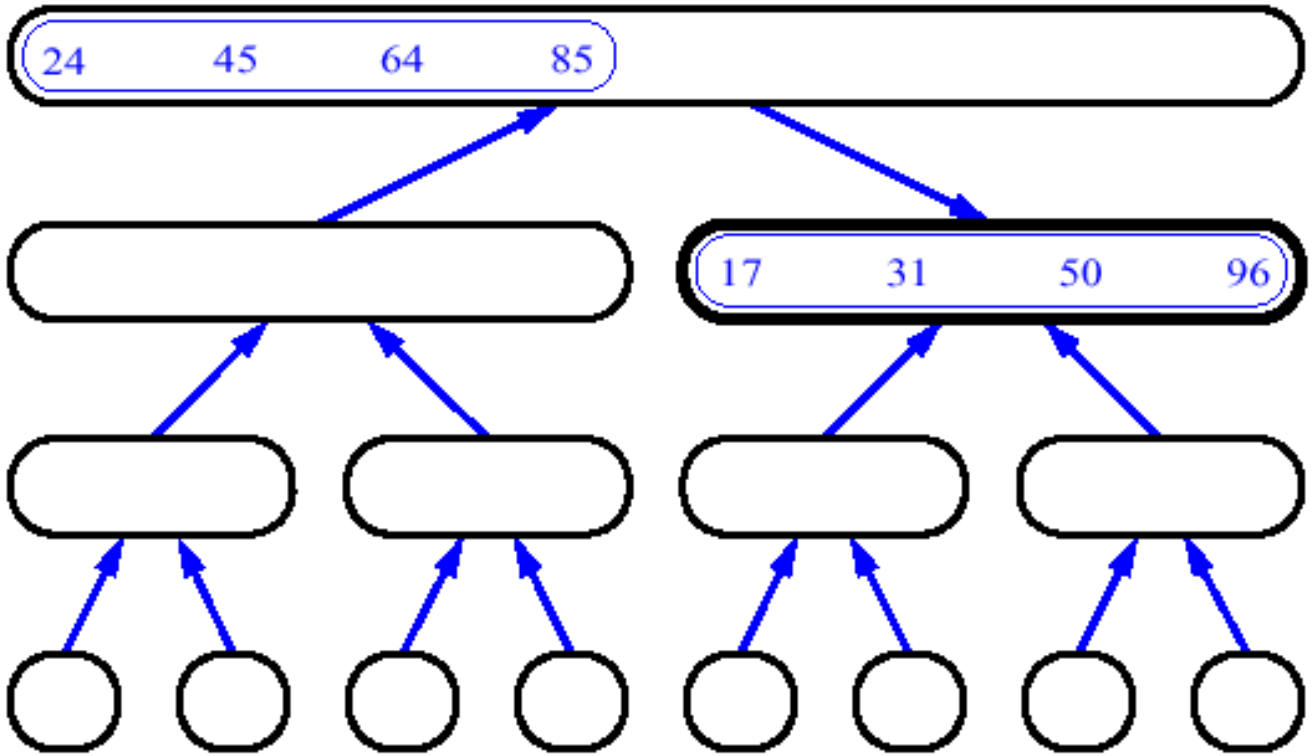
MergeSort (Example) - 18



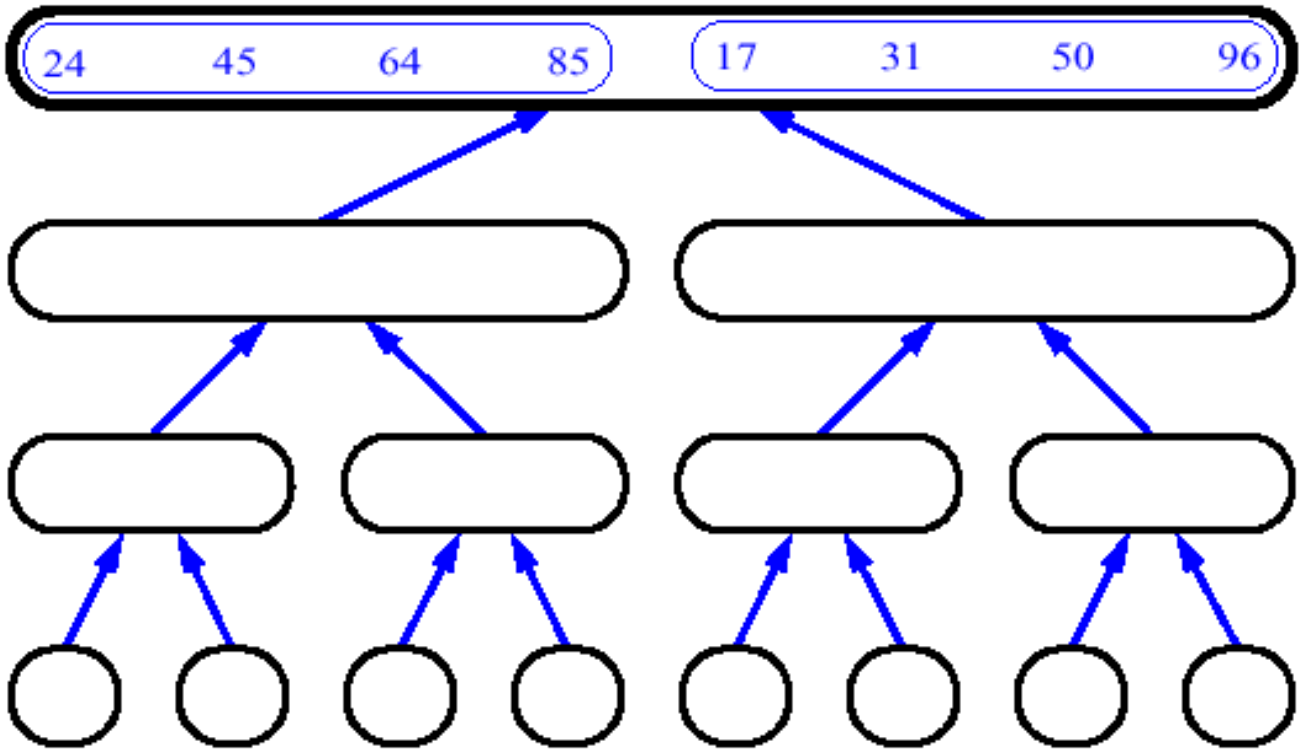
MergeSort (Example) - 19



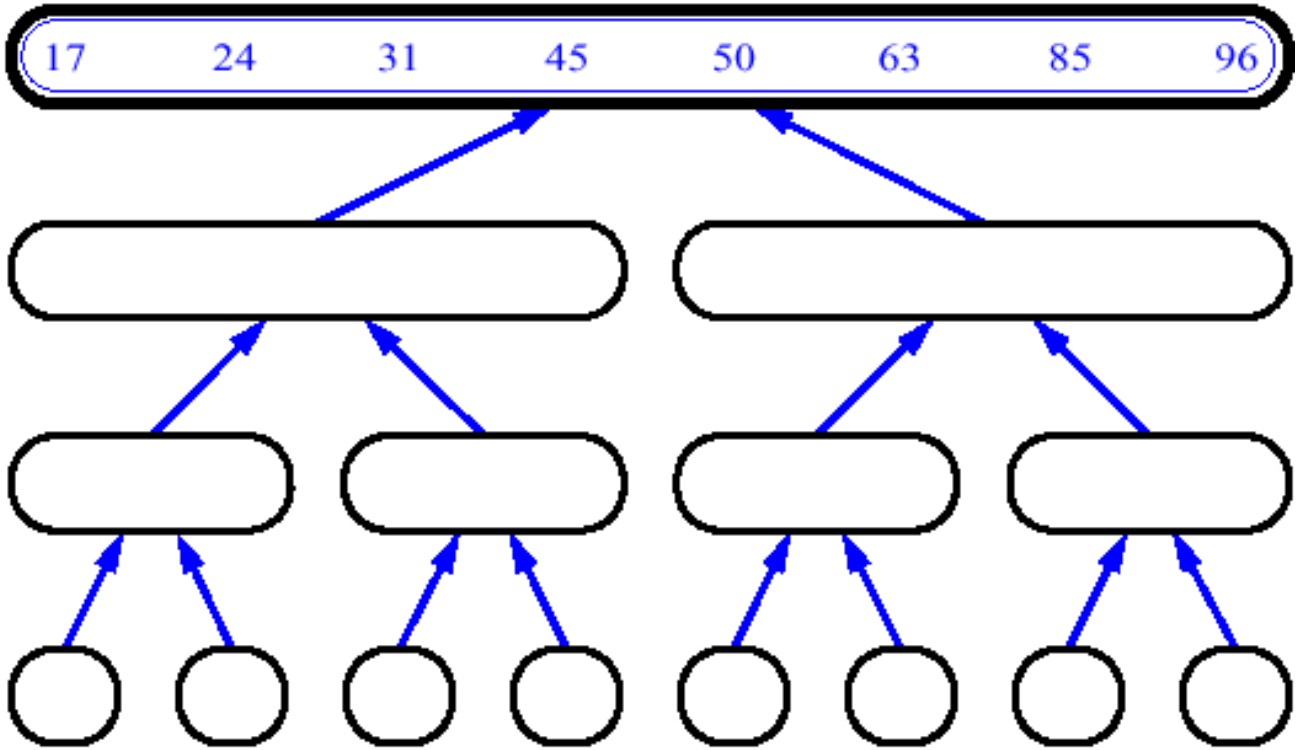
MergeSort (Example) - 20



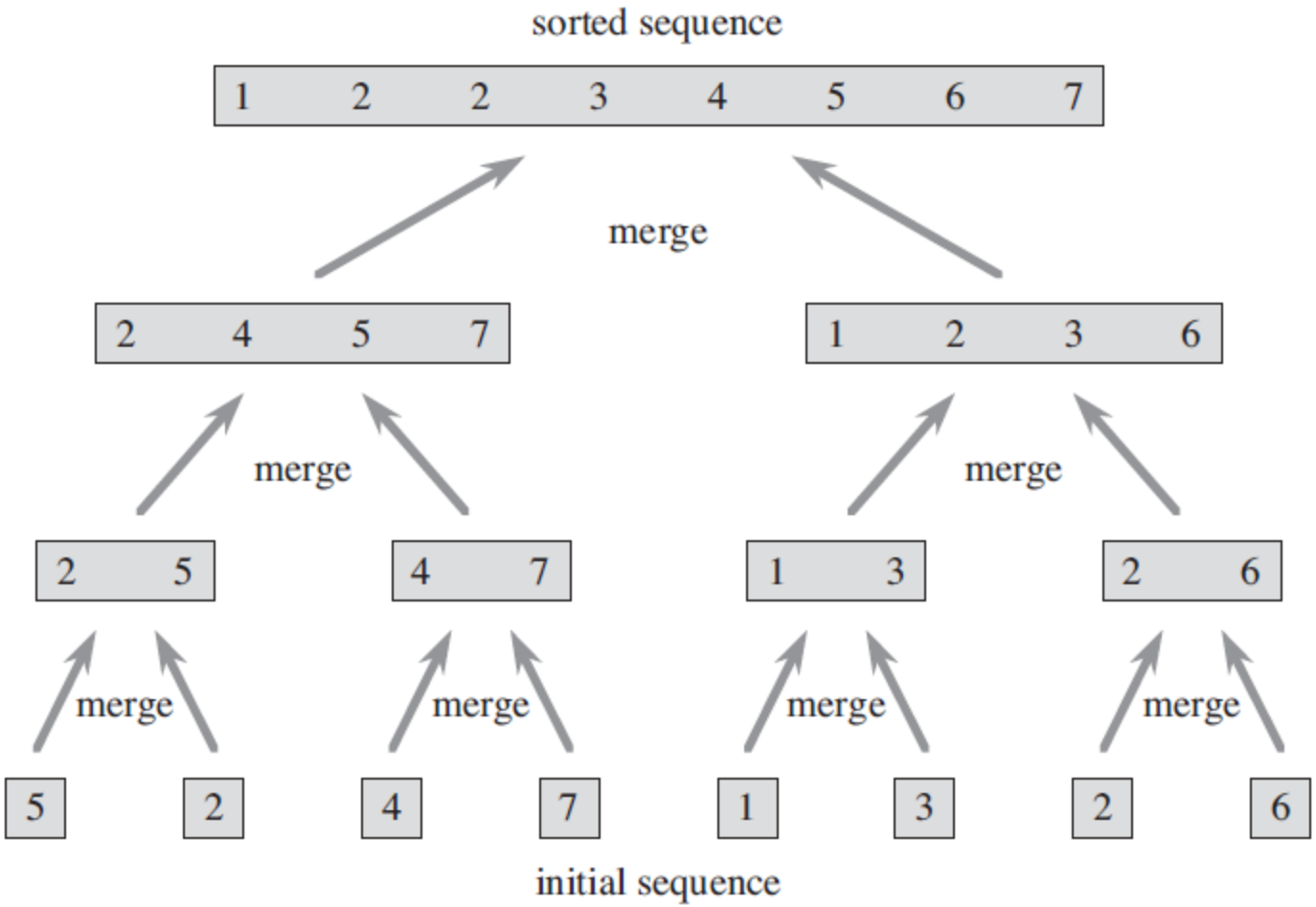
MergeSort (Example) - 21



MergeSort (Example) - 22



MergeSort (Example - 2)



Recurrences

- **Recursive calls** in algorithms can be described using recurrences
- A **recurrence** is an equation or inequality that describes a function in terms of its value on smaller inputs
- Example: Merge Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Binary search

- Idea: Divide and conquer, one of the key design techniques

```
left=1
right=length(A)
do
   j=(left+right)/2
   if A[j]==q then return j
   else if A[j]>q then right=j-1
   else left=j+1
while left<=right
return NIL
```

Example 2: Searching

INPUT

- sequence of numbers (database)
- a single number (query)

$a_1, a_2, a_3, \ldots, a_n;\ q$

2 5 4 10 7; 5

2 5 4 10 7; 9

OUTPUT

- an index of the found number or *NIL*

j

2

NIL

Searching (2)

```
j=1
while j<=length(A) and A[j]!=q
    do j++
if j<=length(A) then return j
else return NIL
```

- Worst-case running time: *f(n)*, average-case: *f(n/2)*
- We can't do better. This is a *lower bound* for the problem of searching in an arbitrary sequence.

Example 3: Searching

INPUT

- sorted non-descending sequence of numbers (database)
- a single number (query)

$a_1, a_2, a_3, \ldots, a_n;\ q$

2 4 5 7 10; 5

2 4 5 7 10; 9

OUTPUT

- an index of the found number or *NIL*

j

2

NIL

Binary search – analysis

- How many times the loop is executed:
  - With each execution its length is cult in half
  - How many times do you have to cut *n* in half to get 1?
  - *lg n*