Algorithms and Data Structures
Lecture II

This Lecture

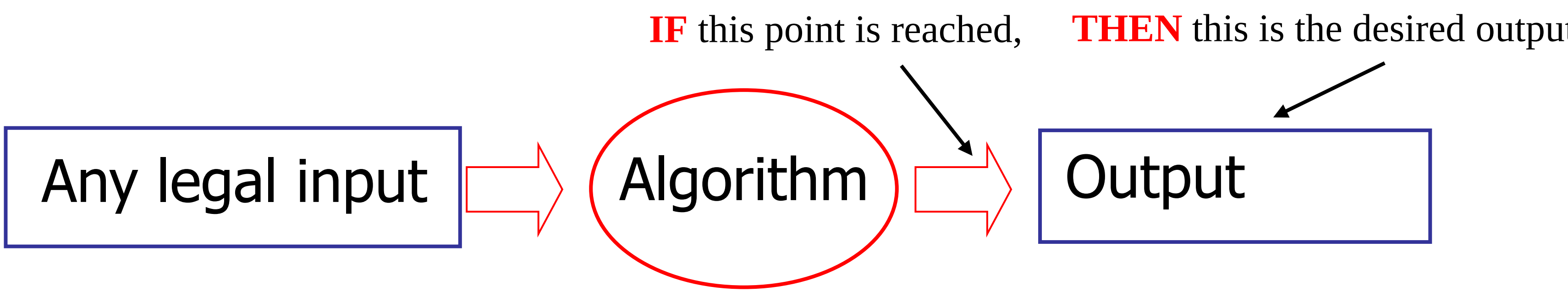
- Correctness of algorithms
- Growth of functions and asymptotic notation
- Some basic math revisited
- Divide and conquer example – the merge sort

Correctness of Algorithms

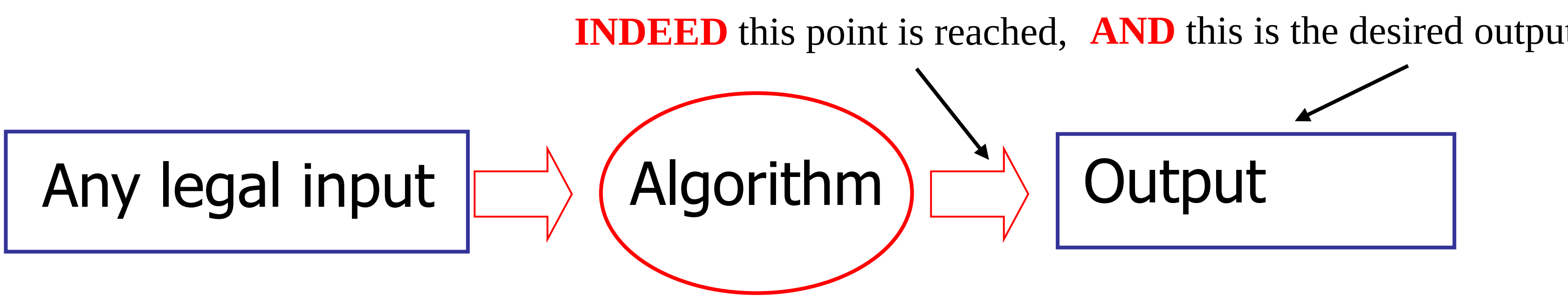
- The algorithm is *correct* if for any legal input it terminates and produces the desired output.
- Automatic proof of correctness is not possible
- But there are practical techniques and rigorous formalisms that help to reason about the correctness of algorithms

Partial and Total Correctness

- Partial correctness



- Total correctness



Assertions

- To prove partial correctness we associate a number of **assertions** (statements about the state of the execution) with specific checkpoints in the algorithm.
  - E.g., A[1], ..., A[k] form an increasing sequence
- **Preconditions** – assertions that must be valid *before* the execution of an algorithm or a subroutine
- **Postconditions** – assertions that must be valid *after* the execution of an algorithm or a subroutine

Loop Invariants

- **Invariants** – assertions that are valid any time they are reached (many times during the execution of an algorithm, e.g., in loops)
- We must show three things about loop invariants:
  - **Initialization** – it is true prior to the first iteration
  - **Maintenance** – if it is true before an iteration, it remains true before the next iteration
  - **Termination** – when loop terminates the invariant gives a useful property to show the correctness of the algorithm

Example of Loop Invariants (1)

- **Invariant**: *at the start of each **for** loop, A[1…j-1] consists of elements originally in A[1…j-1] but in sorted order*

```
for j=2 to length(A)
   do key=A[j]
      i=j-1
      while i>0 and A[i]>key
        do A[i+1]=A[i]
           i--
      A[i+1]:=key
```

Example of Loop Invariants (2)

- **Invariant**: *at the start of each **for** loop, A[1…j-1] consists of elements originally in A[1…j-1] but in sorted order*

```
for j=2 to length(A)
    do key=A[j]
       i=j-1
       while i>0 and A[i]>key
         do A[i+1]=A[i]
            i--
       A[i+1]:=key
```

- **Initialization**: *j = 2,* the invariant trivially holds because *A*[1] is a sorted array ☺

Example of Loop Invariants (3)

- **Invariant**: *at the start of each **for** loop, A[1…j-1] consists of elements originally in A[1…j-1] but in sorted order*

```
for j=2 to length(A)
   do key=A[j]
      i=j-1
      while i>0 and A[i]>key
        do A[i+1]=A[i]
           i--
      A[i+1]:=key
```

- **Maintenance**: the inner **while** loop moves elements $A[j-1]$, $A[j-2]$, …, $A[j-k]$ one position right without changing their order. Then the former $A[j]$ element is inserted into k-th position so that $A[k-1] \le A[k] \le A[k+1]$.

$A[1...j-1]$ sorted + $A[j] \rightarrow A[1...j]$ sorted

Example of Loop Invariants (4)

- **Invariant**: *at the start of each **for** loop, A[1…j-1] consists of elements originally in A[1…j-1] but in sorted order*

```
for j=2 to length(A)
   do key=A[j]
      i=j-1
      while i>0 and A[i]>key
        do A[i+1]=A[i]
           i--
      A[i+1]:=key
```

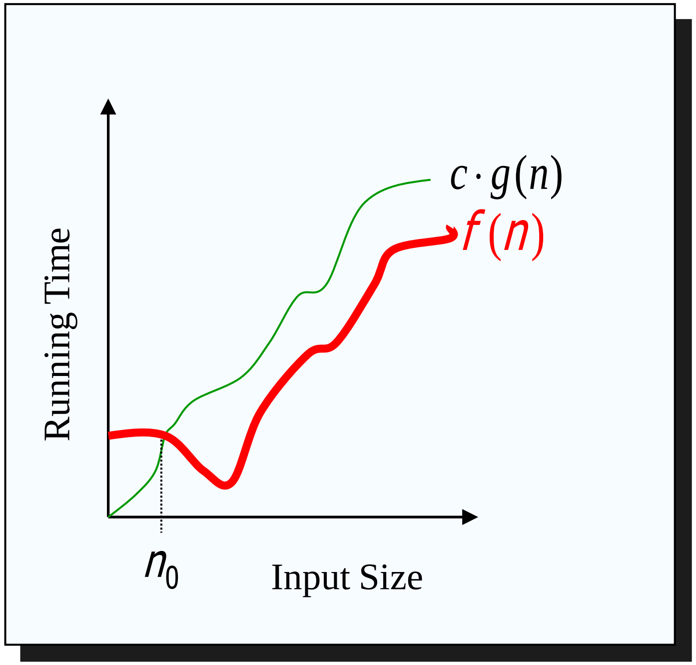
- **Termination**: the loop terminates, when *j=n+1*. Then the invariant states: *"A[1…n] consists of elements originally in A[1…n] but in sorted order"* ☺

Asymptotic Analysis

- Goal: to simplify analysis of running time by getting rid of ”details”, which may be affected by specific implementation and hardware
  - like “rounding”: 1,000,001 $\approx$ 1,000,000
  - $3n^2 \approx n^2$
- Capturing the essence: how the running time of an algorithm increases with the size of the input *in the limit*.
  - Asymptotically more efficient algorithms are best for all but small inputs

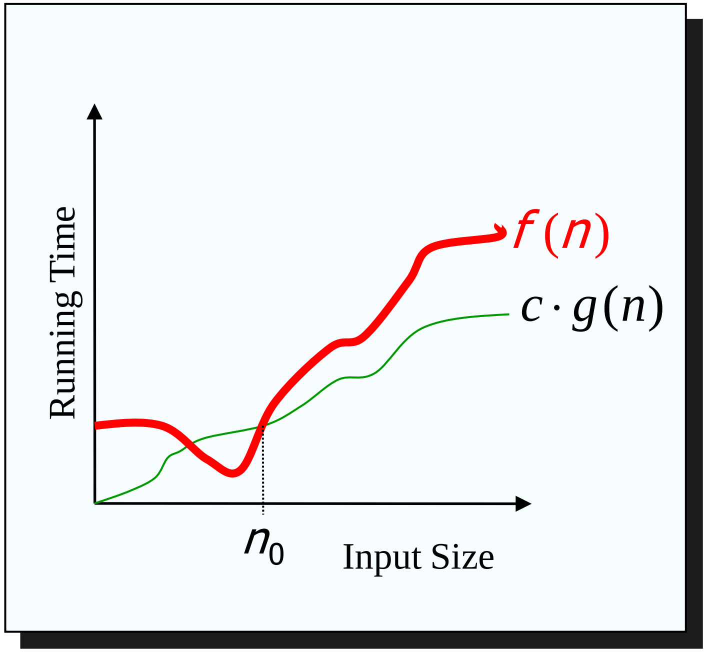
Asymptotic Notation

- The "big-Oh" O-Notation
  - asymptotic upper bound
  - $f(n) = O(g(n))$, if there exists constants c and n_0, *s.t.* $\mathbf{f(n) \le c\, g(n)}$ for $n \ge n_0$
  - $f(n)$ and $g(n)$ are functions over non-negative integers
- Used for *worst-case* analysis



Asymptotic Notation (2)

- The "big-Omega" Ω–Notation
  - asymptotic lower bound
  - $f(n) = \Omega(g(n))$ if there exists constants c and n_0, *s.t.* $\mathbf{c\ g(n) \leq f(n)}$ for $n \geq n_0$
- Used to describe *best-case* running times or lower bounds of algorithmic problems
  - E.g., lower-bound of searching in an unsorted array is $\Omega(n)$.

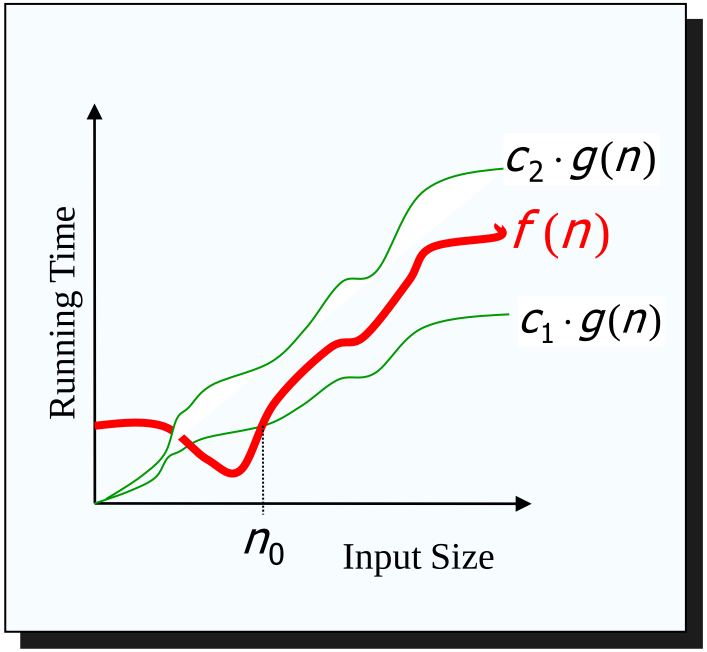


Asymptotic Notation (3)

- Simple Rule: Drop lower order terms and constant factors.
  - $50\, n \log n$ is $O(n \log n)$
  - $7n - 3$ is $O(n)$
  - $8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$
- Note: Even though ($50\, n \log n$) is $O(n^5)$, it is expected that such an approximation be of as small an order as possible

Asymptotic Notation (4)

- The “big-Theta” Θ–Notation
  - asymptoticly tight bound
  - $f(n) = \Theta(g(n))$ if there exists constants c_1, c_2, and n_0, *s.t.* $\mathbf{c_1\, g(n) \le f(n) \le c_2\, g(n)}$ for $n \ge n_0$
- $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- $O(f(n))$ is often misused instead of $\Theta(f(n))$



Asymptotic Notation (5)

- Two more asymptotic notations
  - "Little-Oh" notation $f(n)=o(g(n))$ non-tight analogue of Big-Oh
    - For every c, there should exist n_0, s.t. $\mathbf{f(n)} < \mathbf{c}\ \mathbf{g(n)}$ for $n \geq n_0$
    - Used for **comparisons** of running times. If $f(n)=o(g(n))$, it is said that *g(n) dominates f(n).*
  - "Little-omega" notation $f(n)=\omega(g(n))$ non-tight analogue of Big-Omega

Asymptotic Notation (6)

- Analogy with real numbers
  - $f(n) = O(g(n)) \quad \cong \quad f \leq g$
  - $f(n) = \Omega(g(n)) \quad \cong \quad f \geq g$
  - $f(n) = \Theta(g(n)) \quad \cong \quad f = g$
  - $f(n) = o(g(n)) \quad \cong \quad f < g$
  - $f(n) = \omega(g(n)) \quad \cong \quad f > g$
- Abuse of notation: $f(n) = O(g(n))$ actually means $f(n) \in O(g(n))$

A Quick Math Review

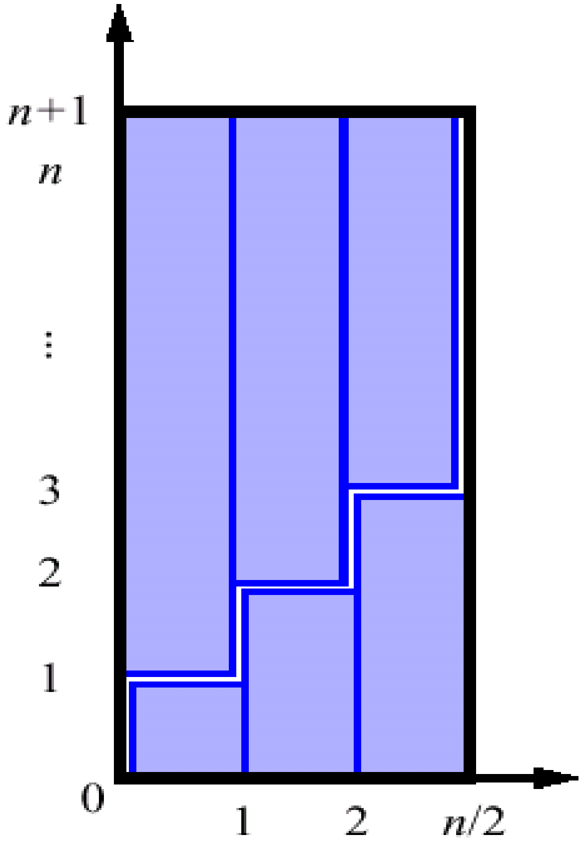
- Geometric progression
  - given an integer n_0 and a real number $0 < a \neq 1$

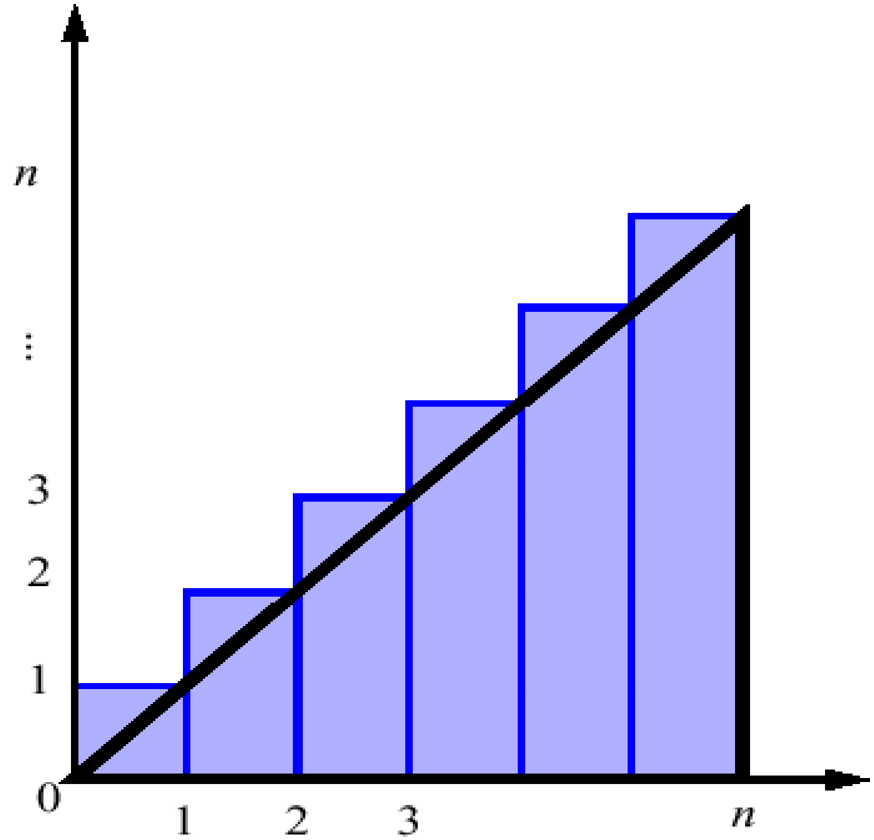
$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \ldots + a^n = \frac{1 - a^{n+1}}{1 - a}$$

  - geometric progressions exhibit exponential growth
- Arithmetic progression

$$\sum_{i=0}^{n} i = 1 + 2 + 3 + \ldots + n = \frac{n^2 + n}{2}$$

A Quick Math Review (2)





Summations

- The running time of insertion sort is determined by a nested loop

```
for j←2 to length(A)
   key←A[j]
i←j-1
   while i>0 and A[i]>key
      A[i+1]←A[i]
      i←i-1
   A[i+1]←key
```

- Nested loops correspond to summations

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

Proof by Induction

- We want to show that property P is true for all integers $n \geq n_0$
- **Basis**: prove that P is true for n_0
- **Inductive step**: prove that if P is true for all k such that $n_0 \leq k \leq n - 1$ then P is also true for n
- Example $S(n) = \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ for $n \geq 1$
- Basis $S(1) = \sum_{i=0}^{1} i = \frac{1(1+1)}{2}$

Proof by Induction (2)

- Inductive Step

$$S(k)=\sum_{i=0}^{k} i=\frac{k(k+1)}{2} \text{ for } 1\leq \mathrm{k} \leq n-1$$

$$S(n)=\sum_{i=0}^{n} i=\sum_{i=0}^{n-1} i+n=S(n-1)+n=$$

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

$$=\frac{n(n+1)}{2}$$

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

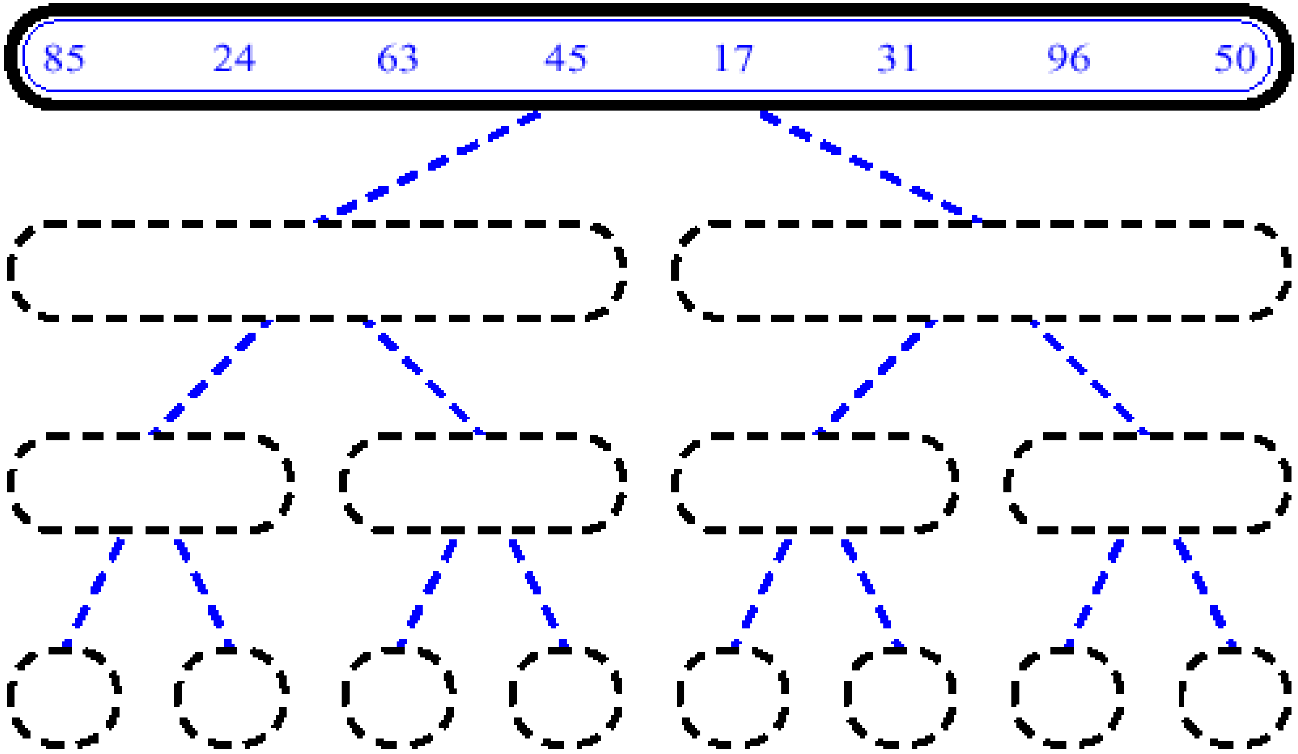
- **Divide**: If S has at least two elements (nothing needs to be done if S has zero or one elements), remove all the elements from S and put them into two sequences, S_1 and S_2 , each containing about half of the elements of S. (i.e. S_1 contains the first $\lceil n/2 \rceil$ elements and S_2 contains 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

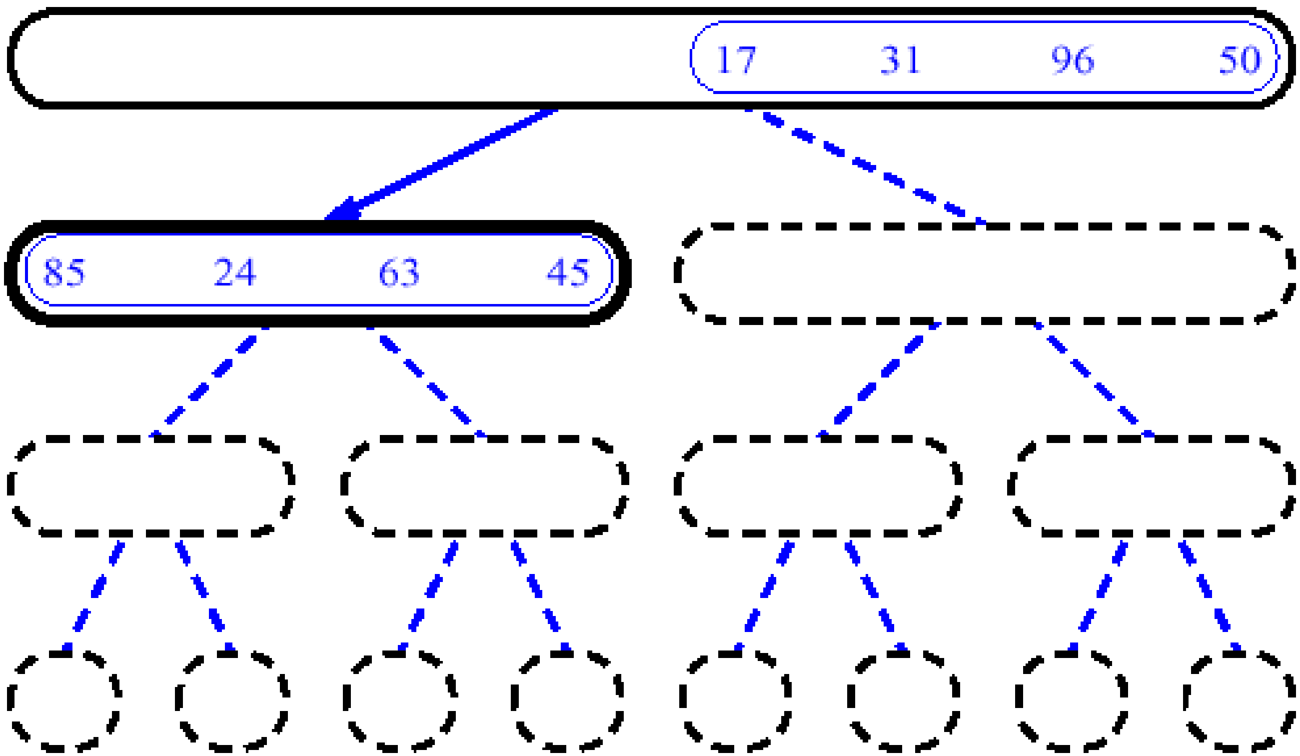
```
Merge-Sort(A, p, r)
    if p < r then
        q←(p+r)/2
        Merge-Sort(A, p, q)
        Merge-Sort(A, q+1, r)
        Merge(A, p, q, r)
```

```
Merge(A, p, q, r)
    Take the smallest of the two topmost elements of
sequences A[p..q] and A[q+1..r] and put into the
resulting sequence. Repeat this, until both sequences
are empty. Copy the resulting sequence into A[p..r].
```

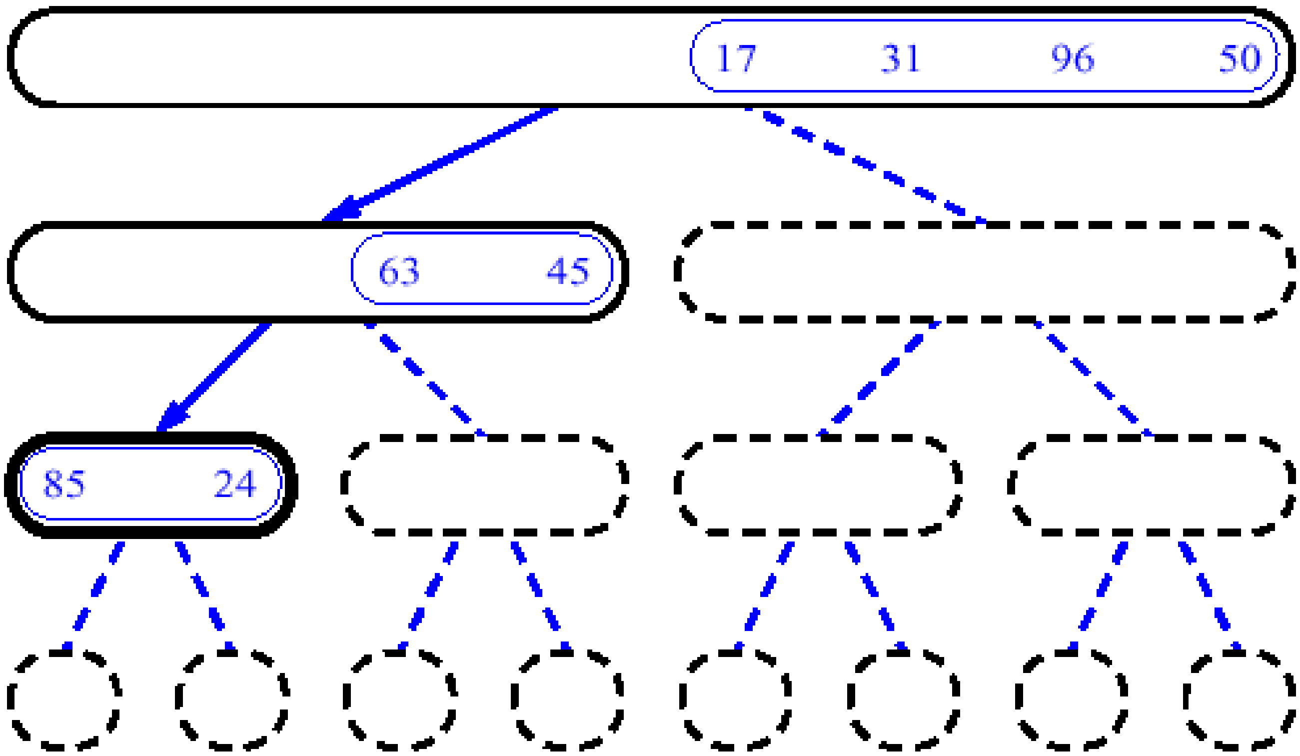
MergeSort (Example) - 1



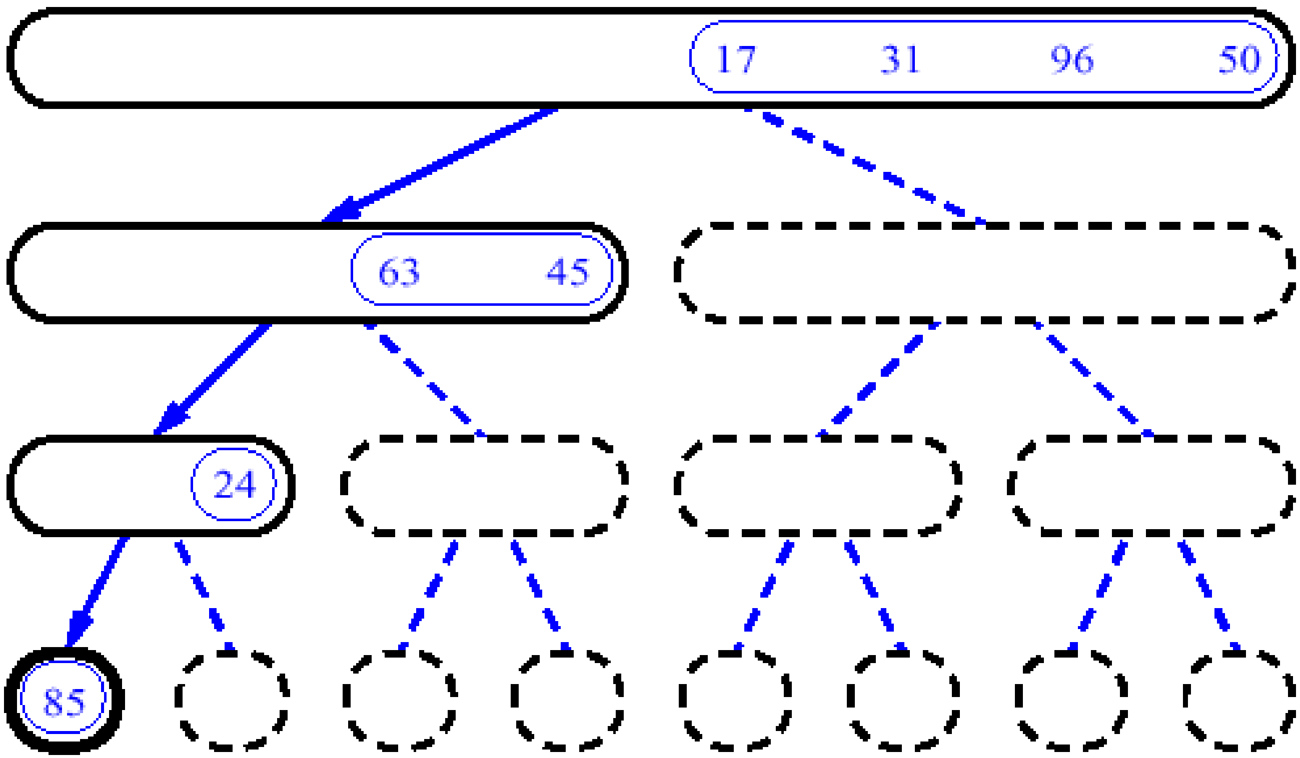
MergeSort (Example) - 2



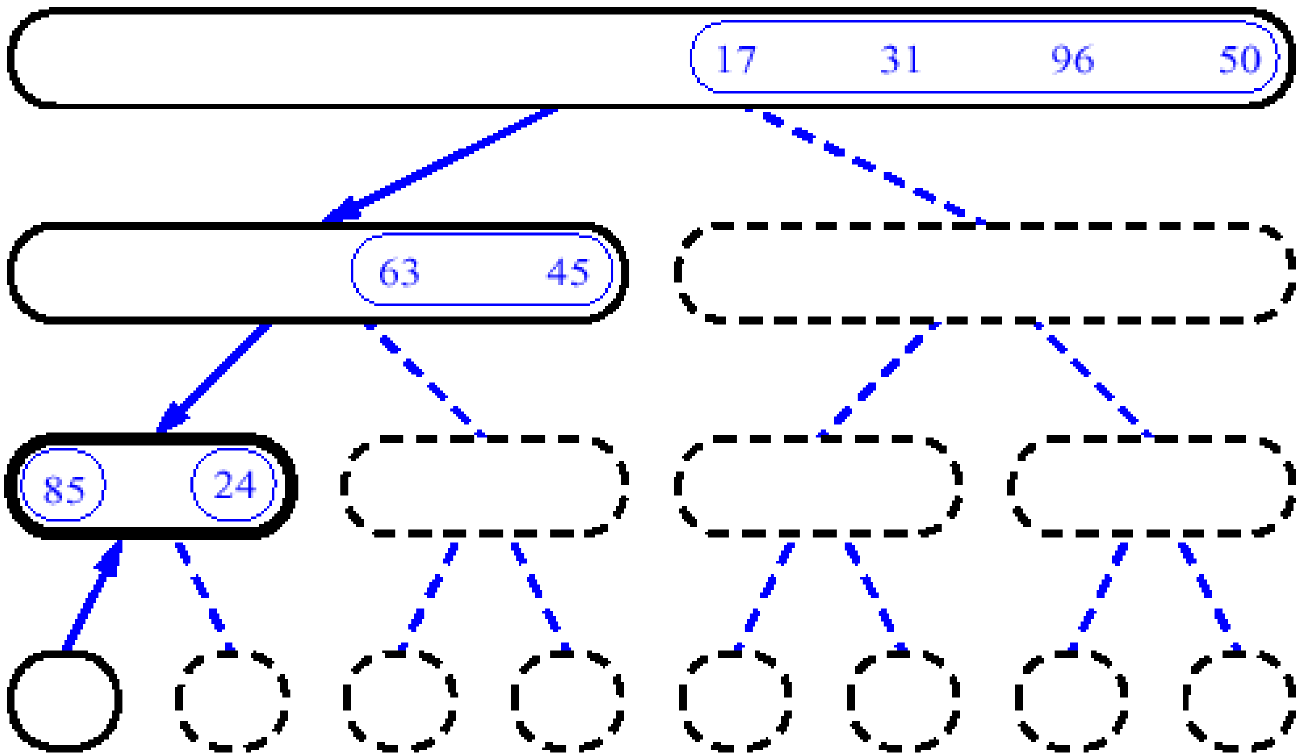
MergeSort (Example) - 3



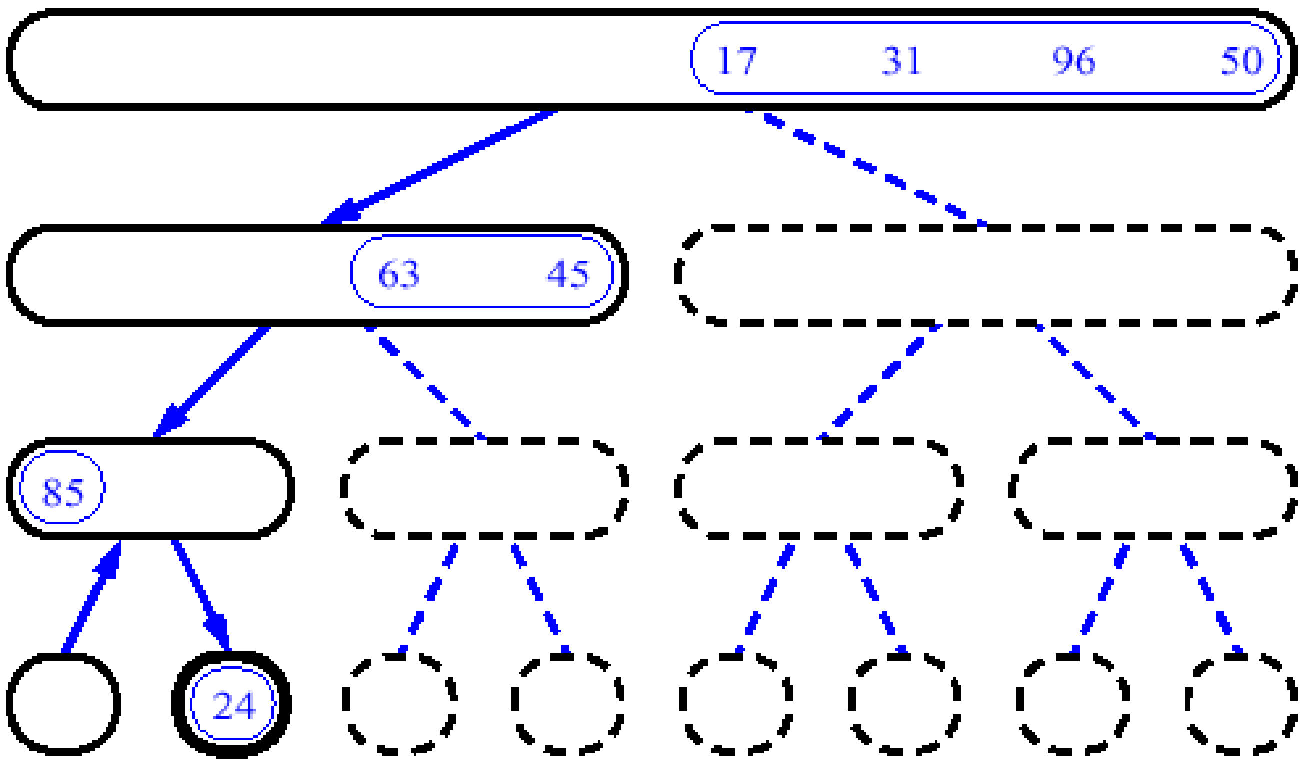
MergeSort (Example) - 4



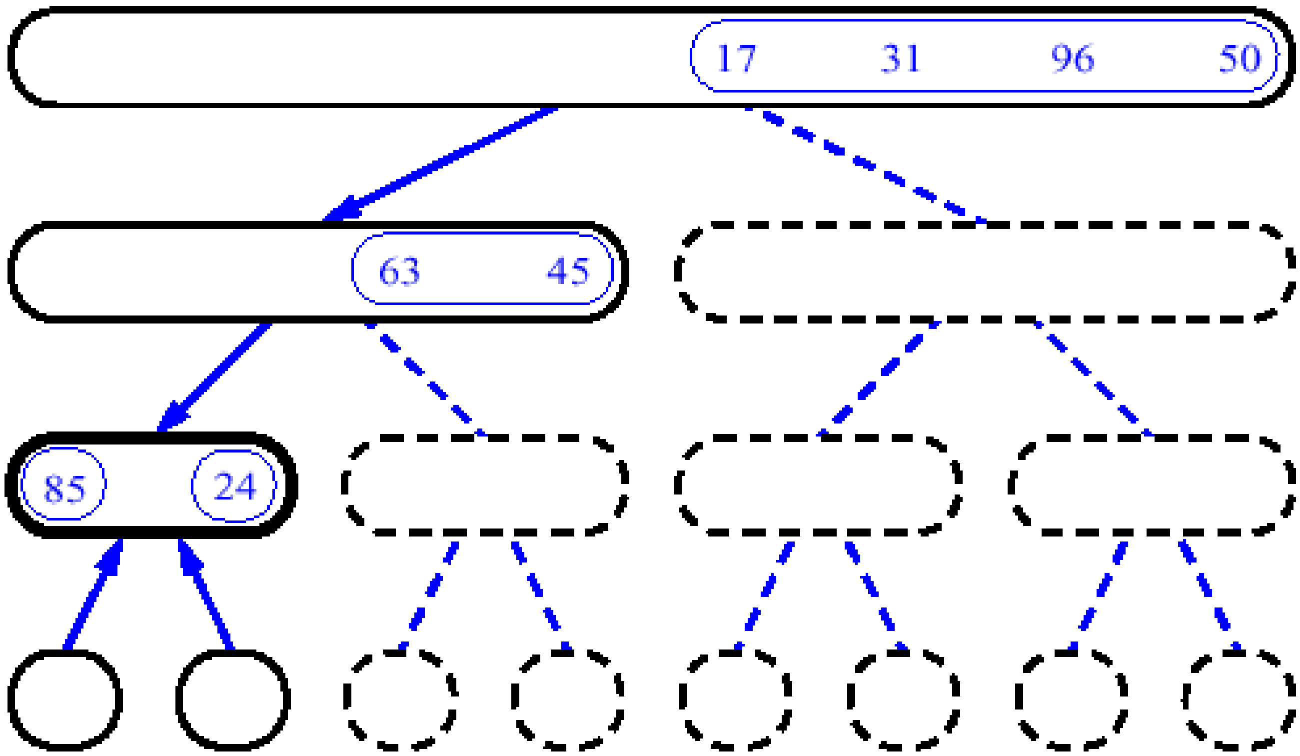
MergeSort (Example) - 5



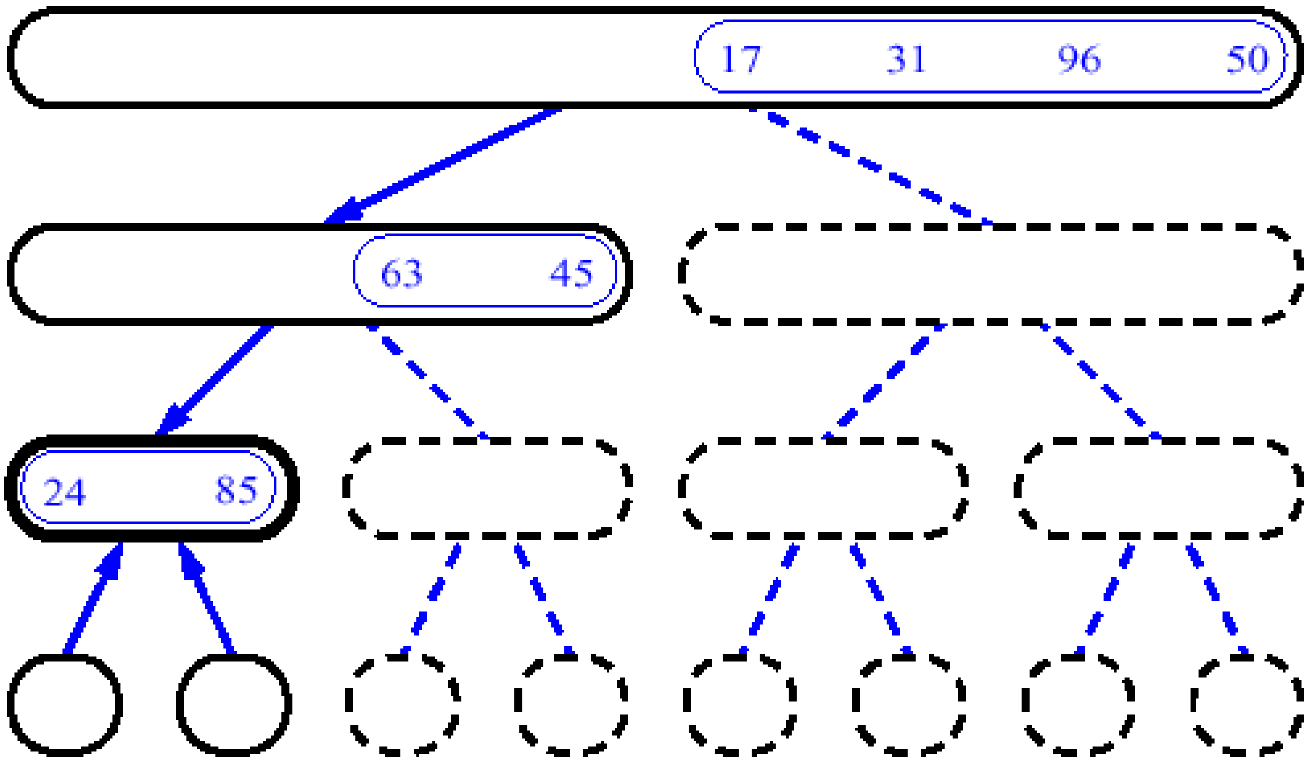
MergeSort (Example) - 6



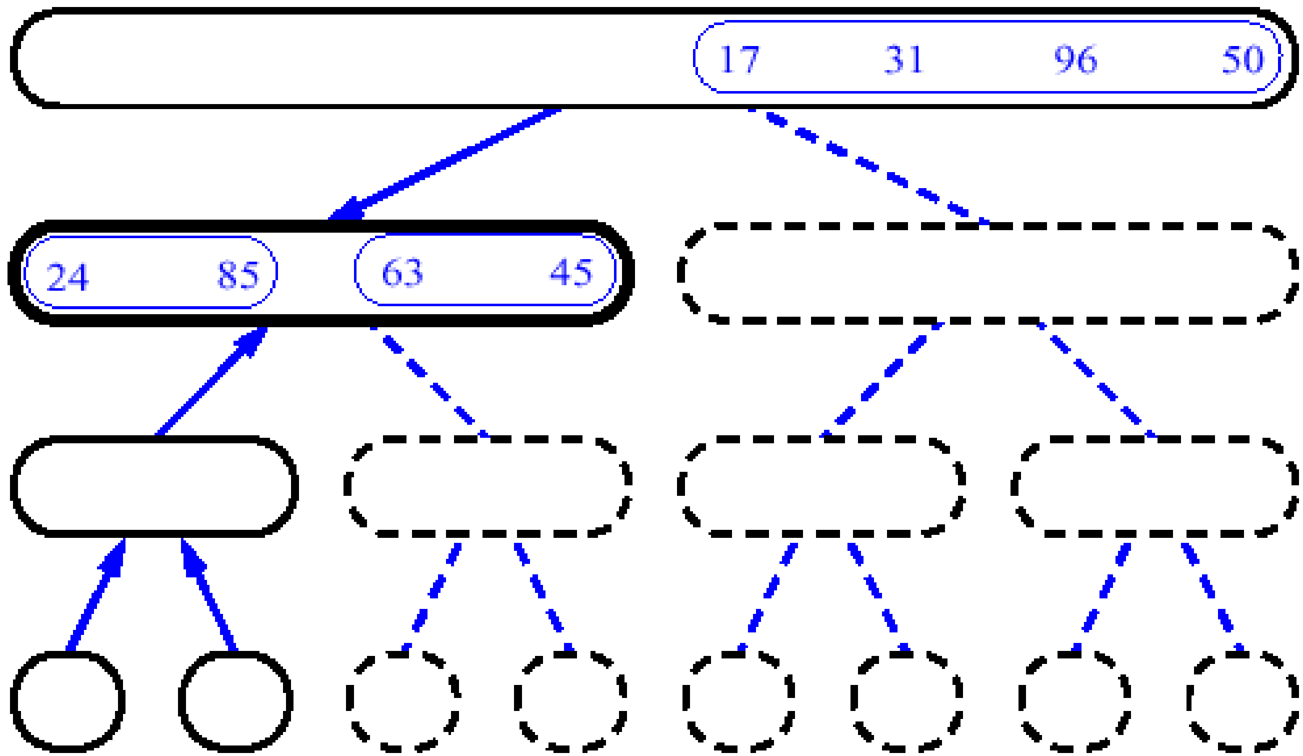
MergeSort (Example) - 7



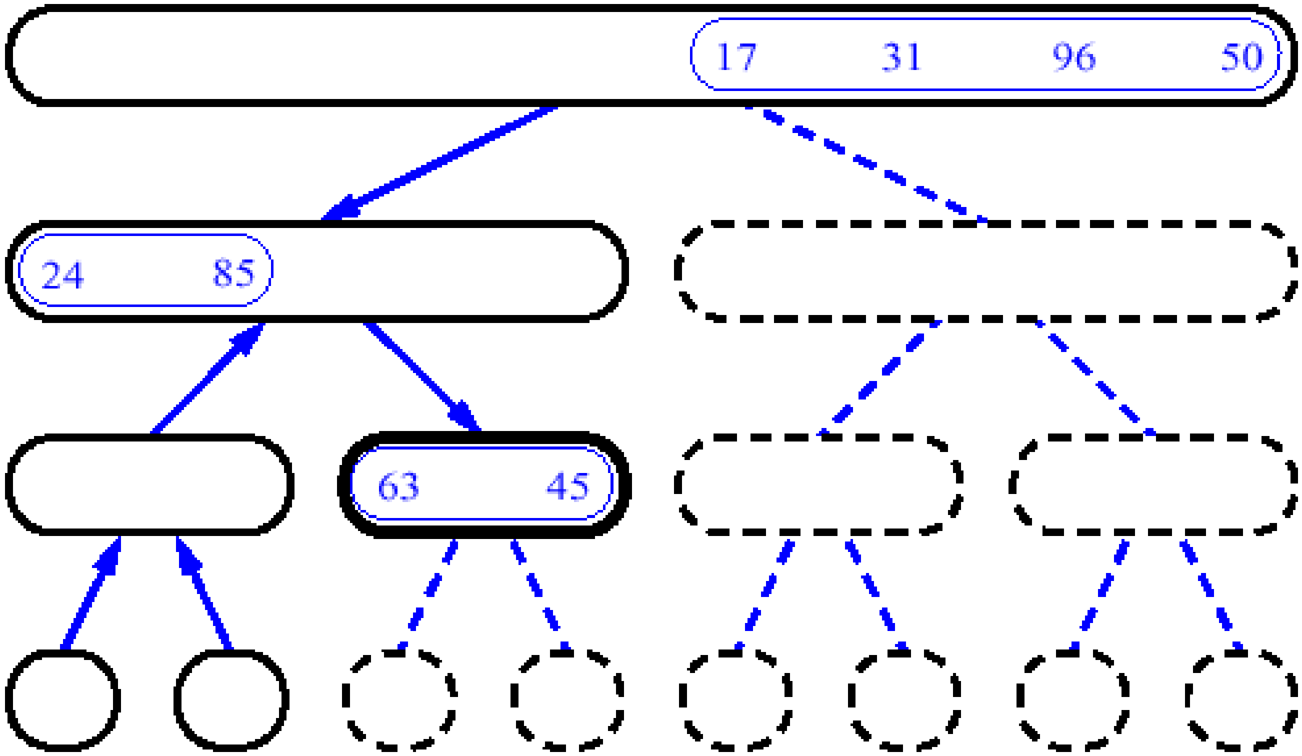
MergeSort (Example) - 8



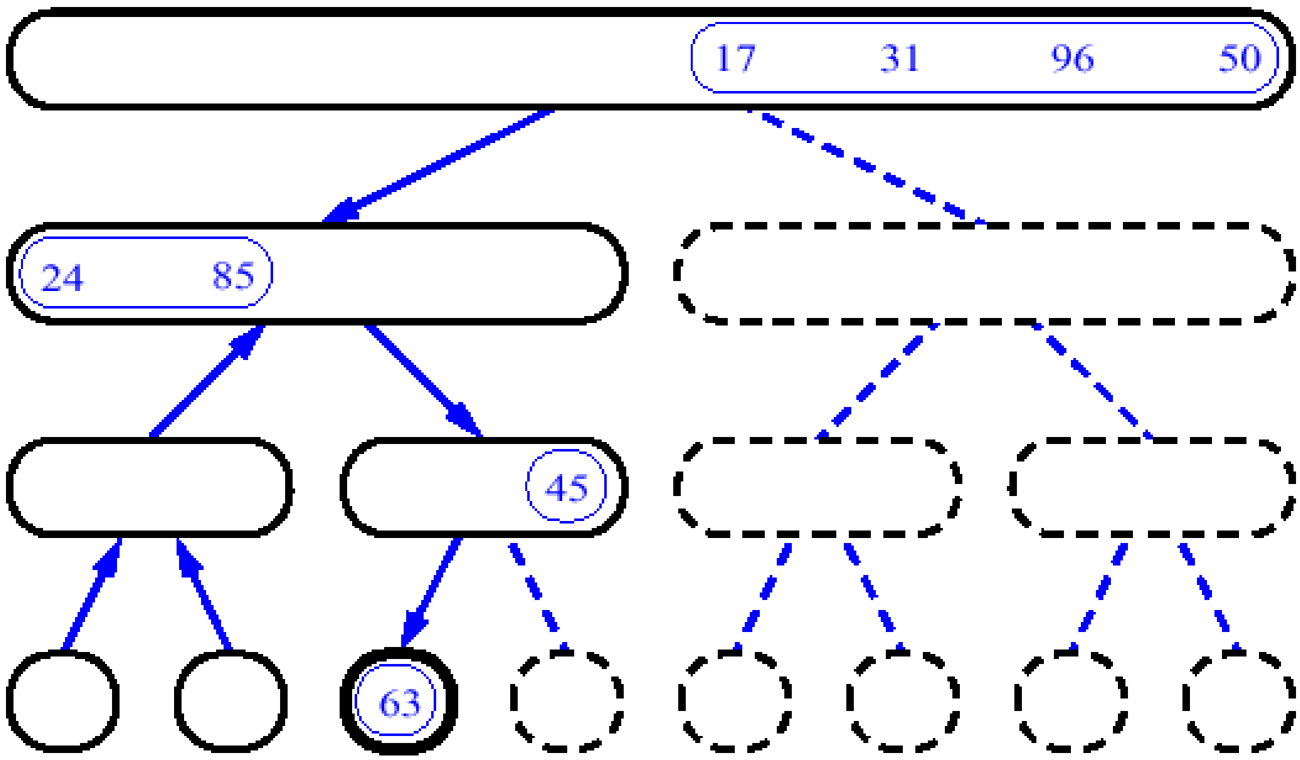
MergeSort (Example) - 9



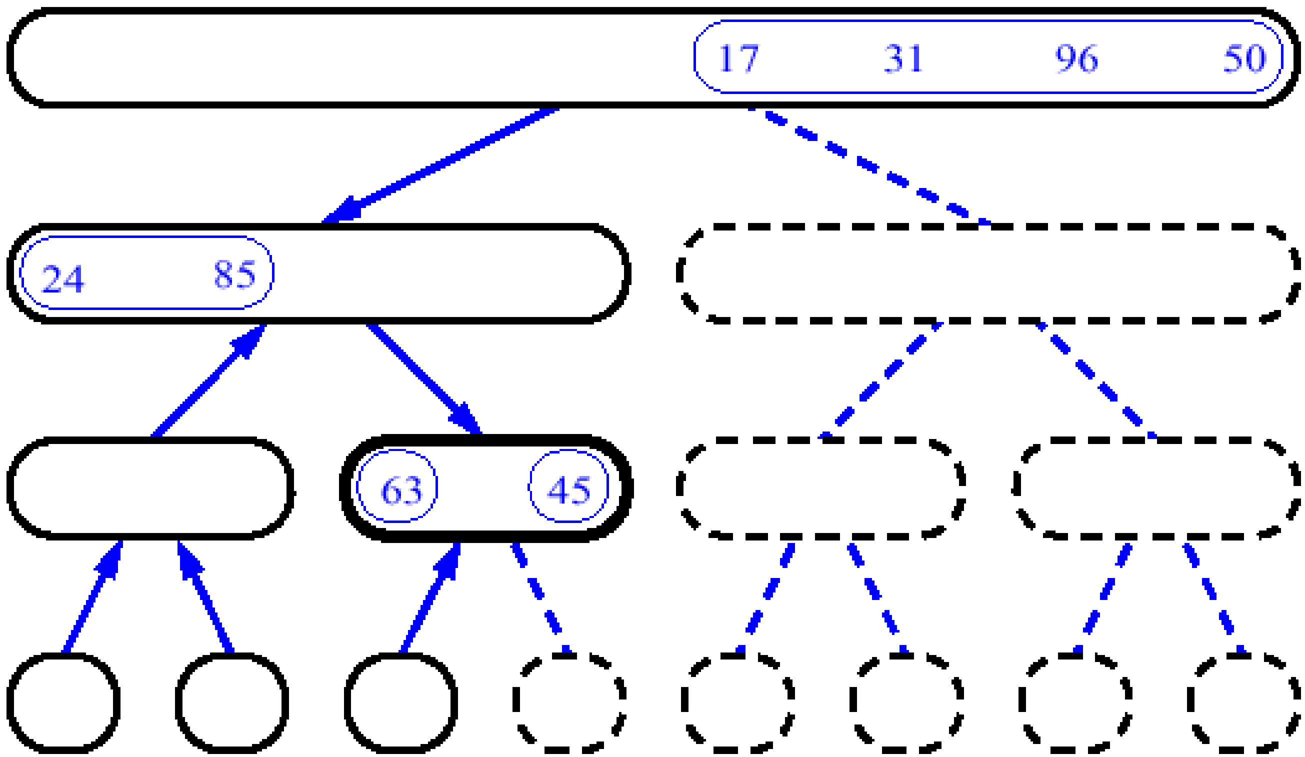
MergeSort (Example) - 10



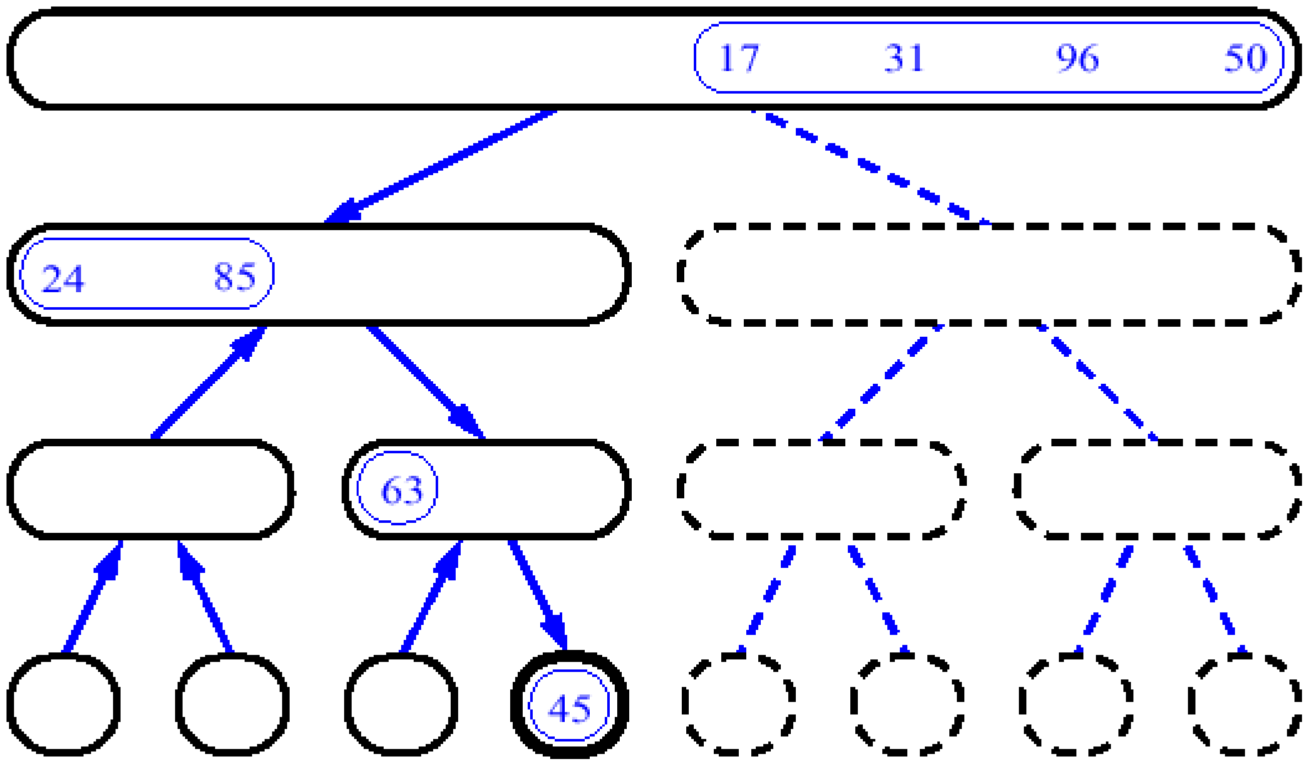
MergeSort (Example) - 11



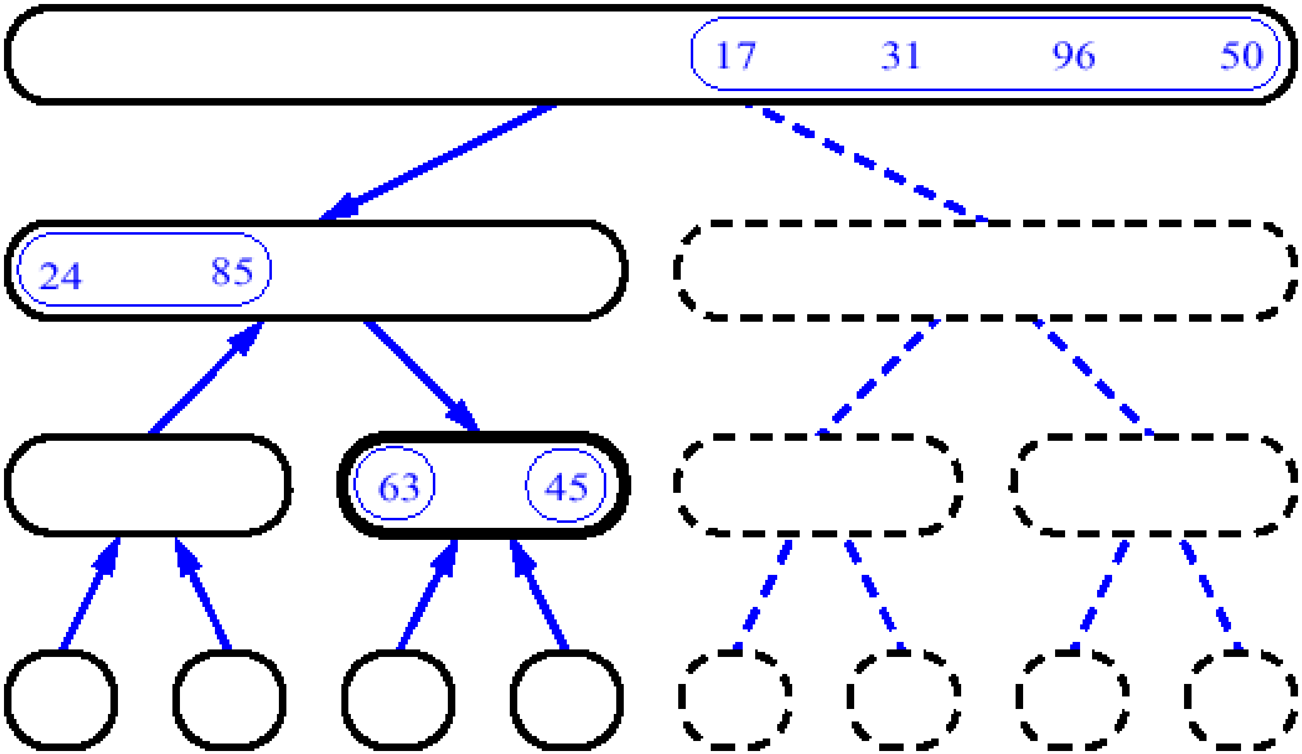
MergeSort (Example) - 12



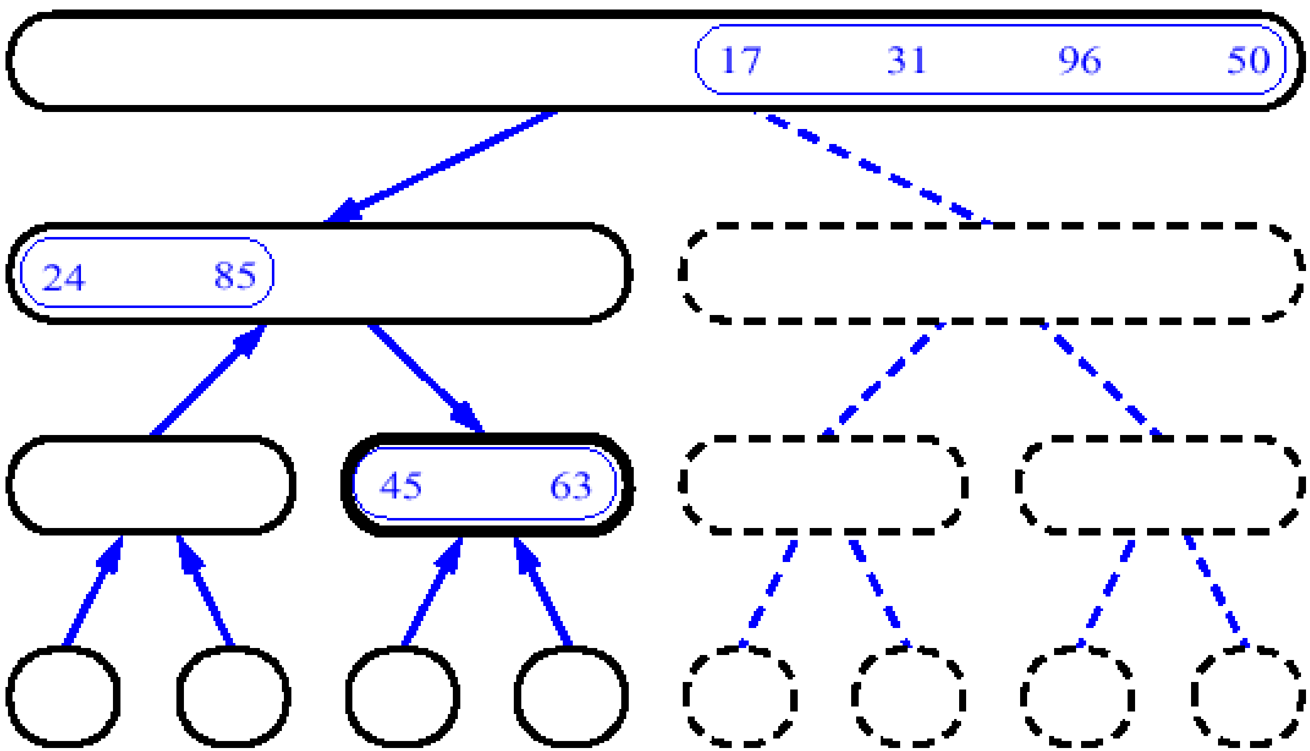
MergeSort (Example) - 13



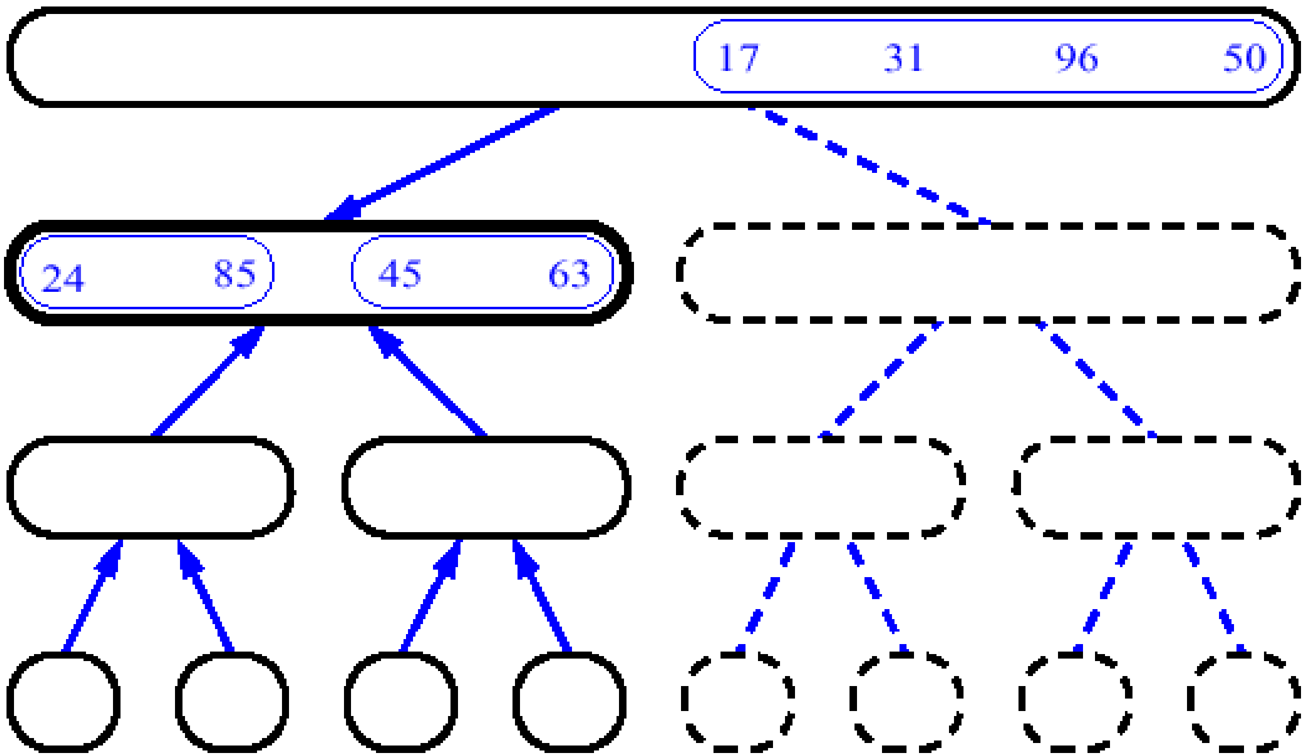
MergeSort (Example) - 14



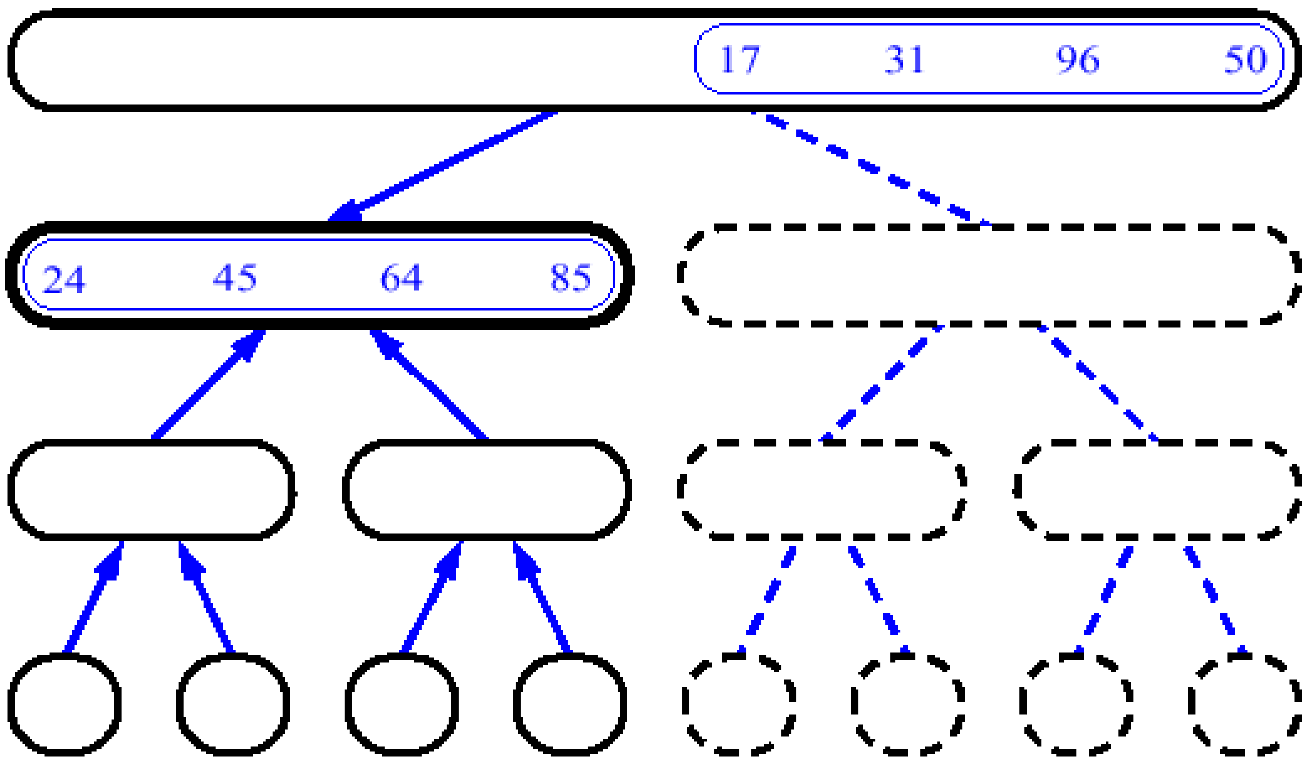
MergeSort (Example) - 15



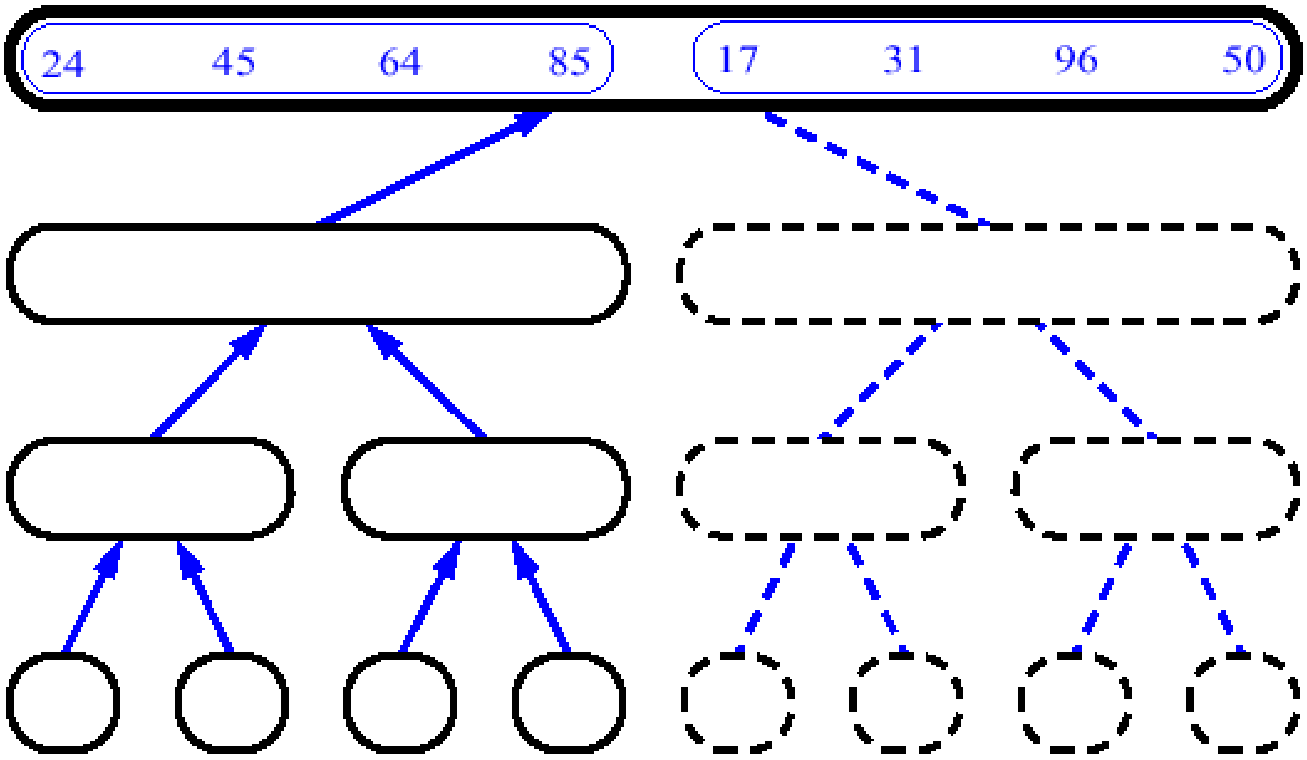
MergeSort (Example) - 16



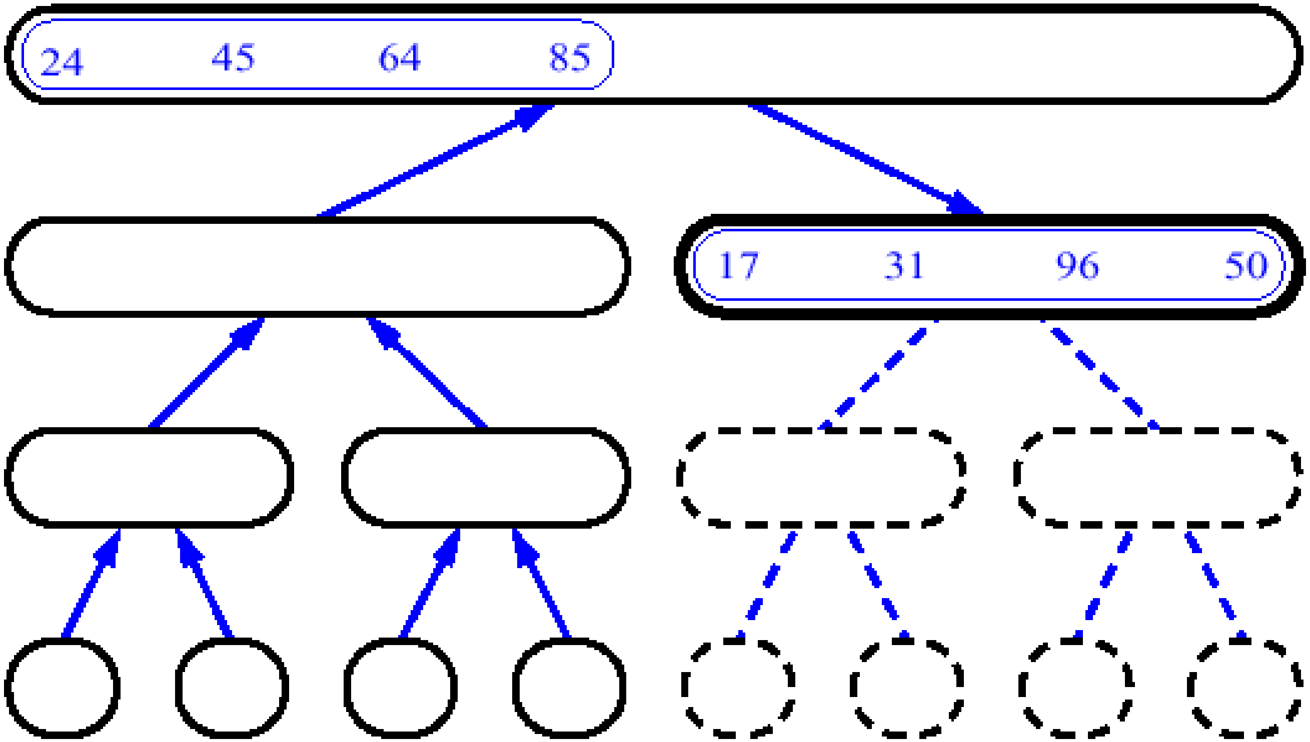
MergeSort (Example) - 17



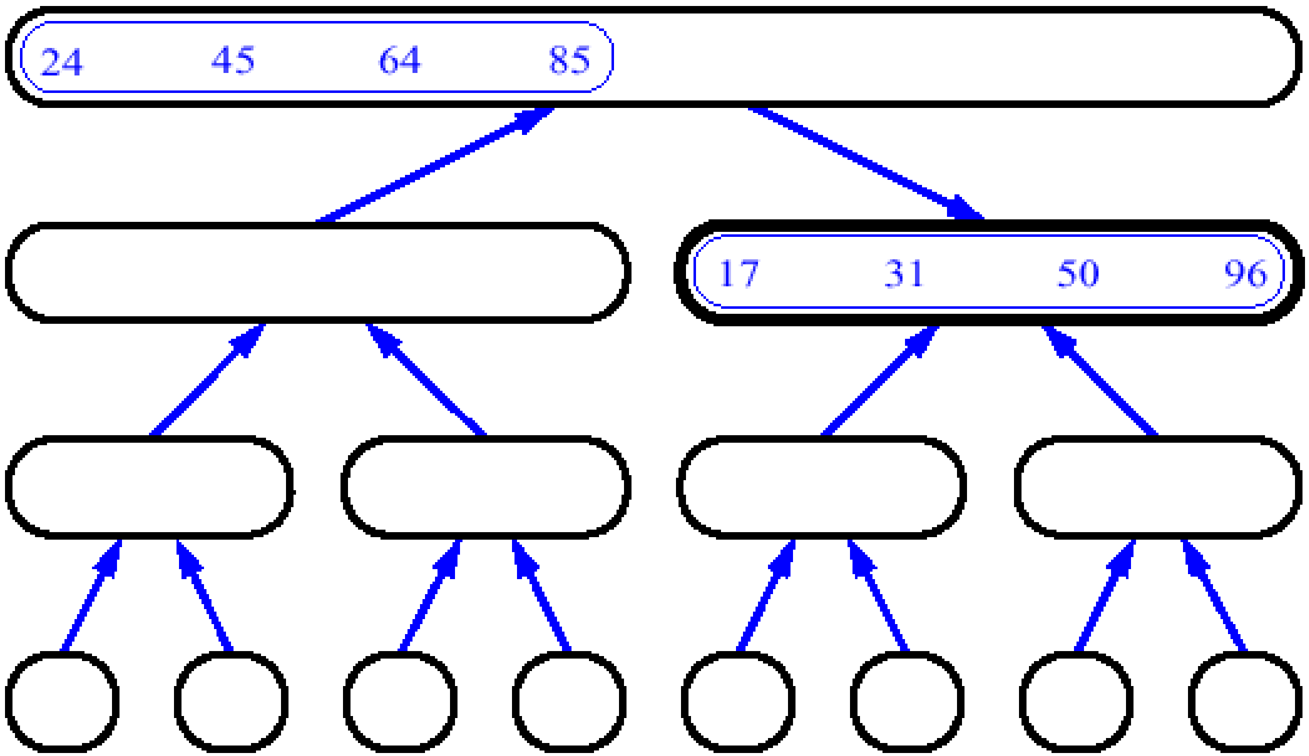
MergeSort (Example) - 18



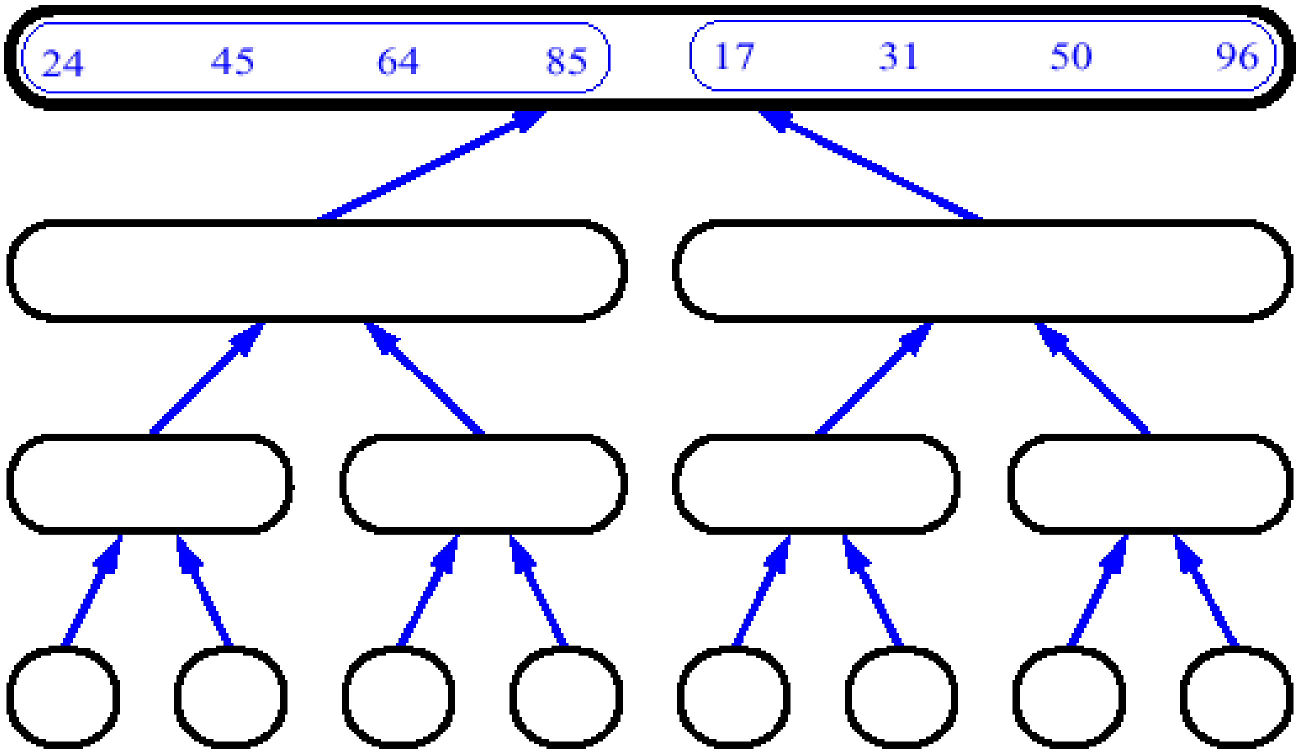
MergeSort (Example) - 19



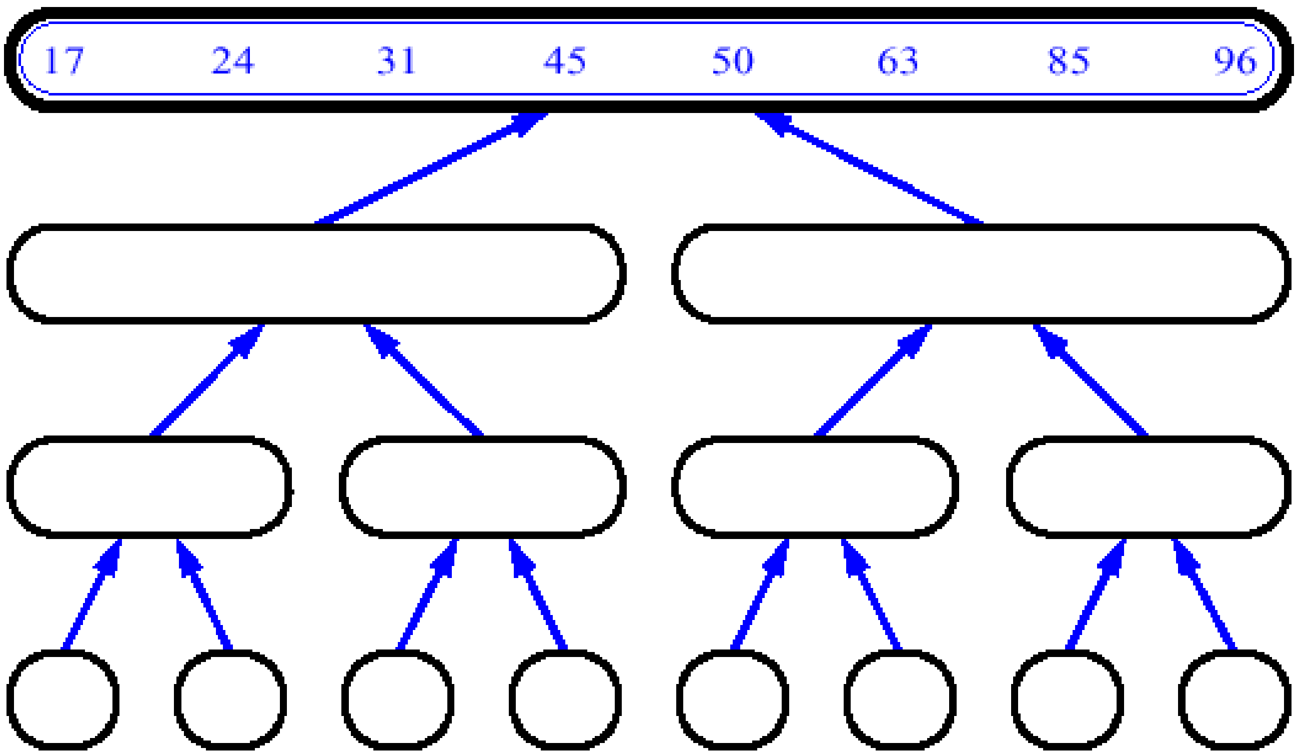
MergeSort (Example) - 20



MergeSort (Example) - 21



MergeSort (Example) - 22



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

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

Next lecture

- Solving recurrences