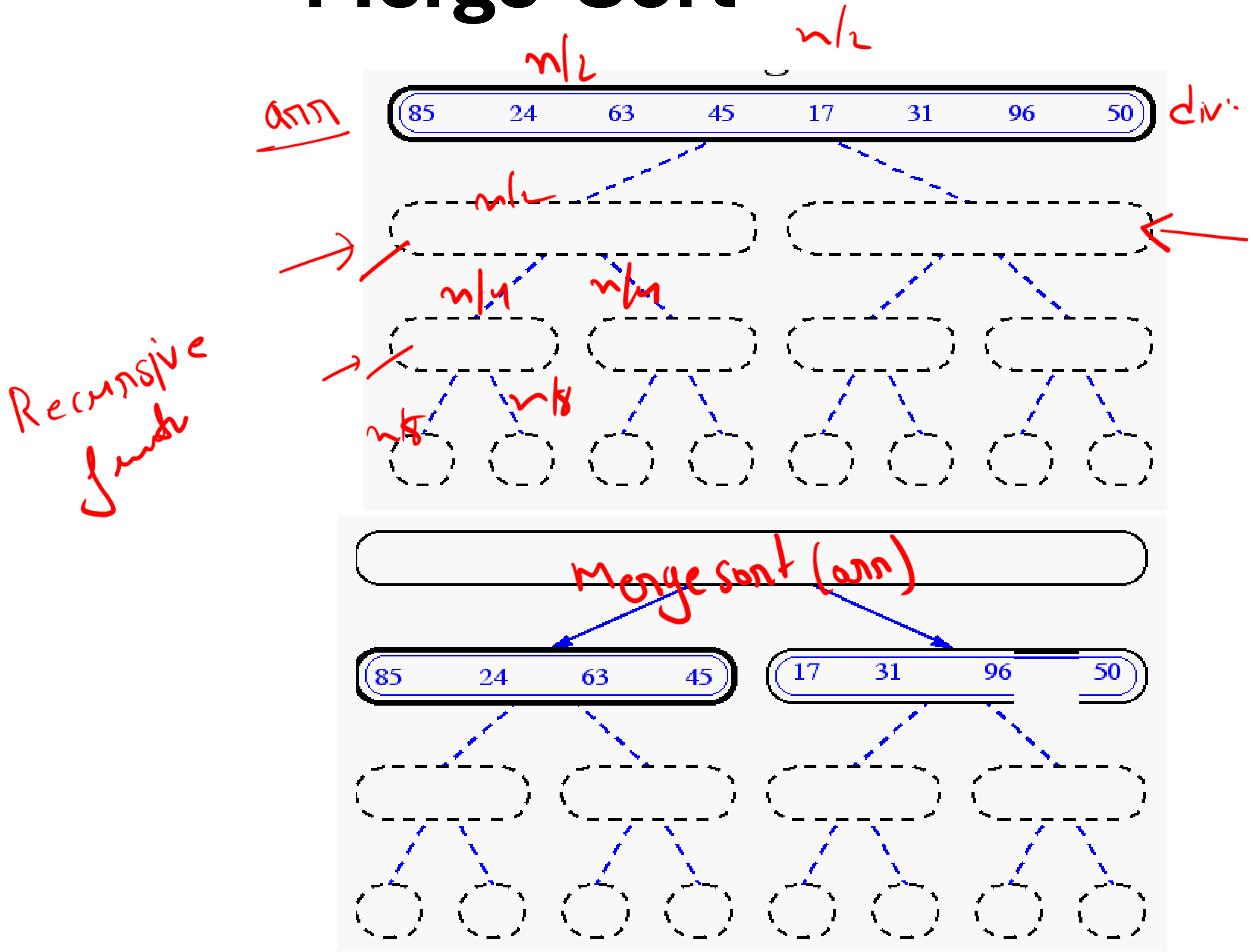
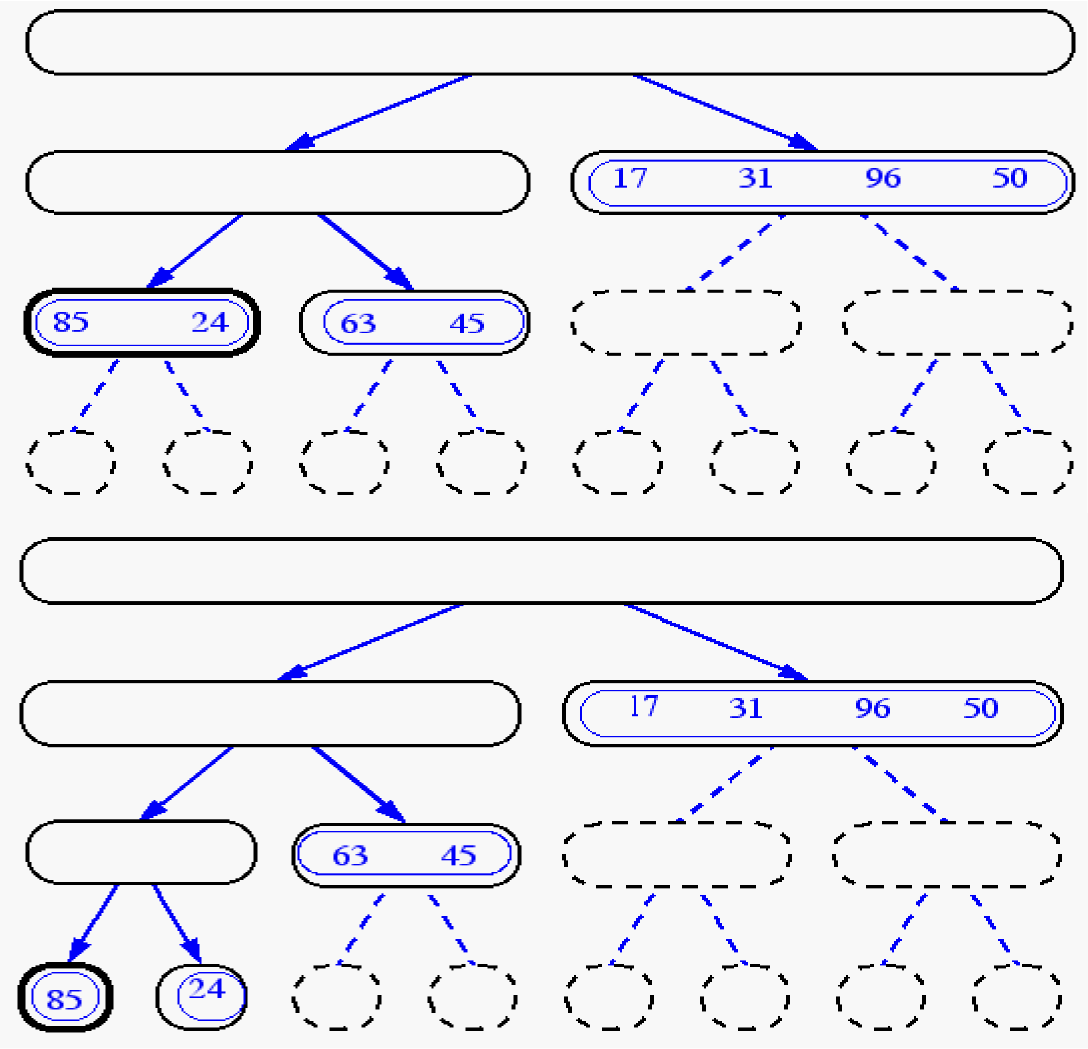
Merge-Sort

- Algorithm:
  - **Divide**: If **S** has at least two elements (nothing needs to be done if S has zero or one element), 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** Recursive sort sequences S_1 and S_2.
  - **Combine:** Put back the elements into S by merging the sorted sequences S_1 and S_2 into a unique sorted sequence.

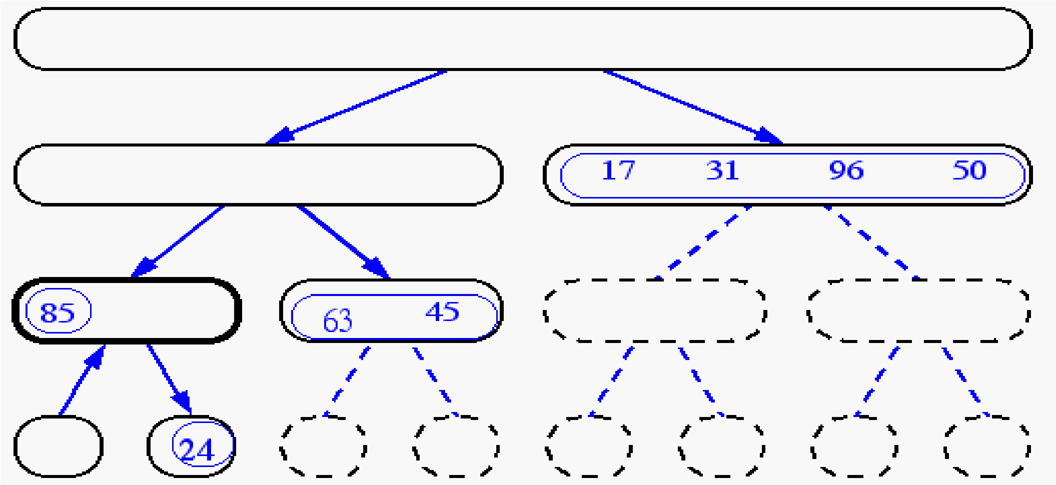
Merge-Sort

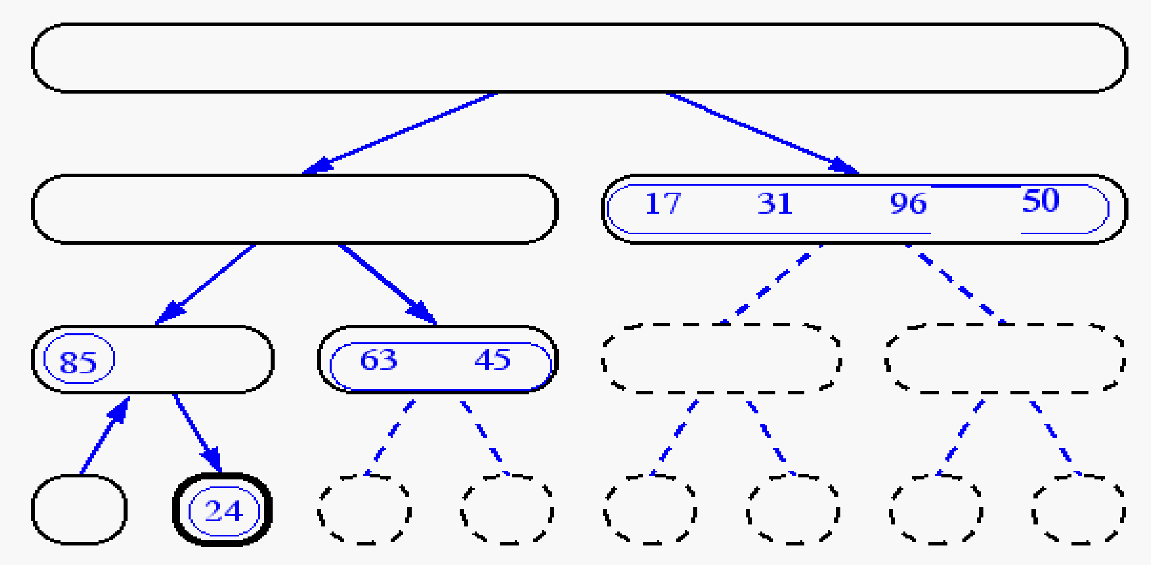

n/2
n/2
arr
85 24 63 45 17 31 96 50
div.
n/2
n/4
n/4
n/8
n/8
Recursive func
Merge sort (arr)
85 24 63 45
17 31 96 50

Merge-Sort(cont.)

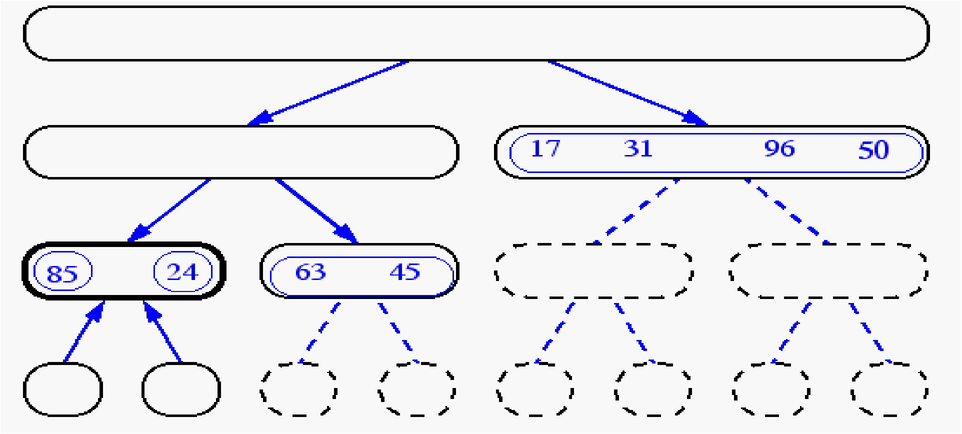


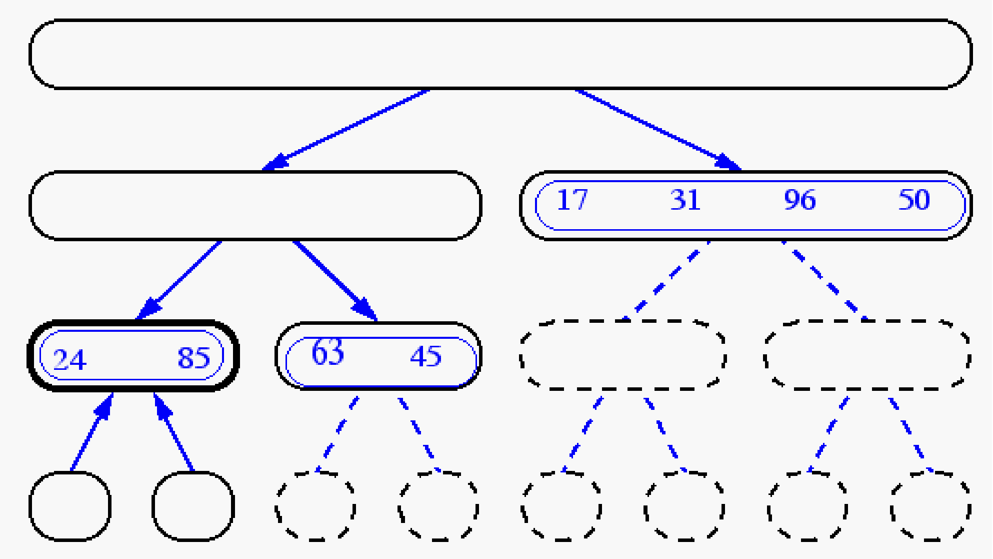
Merge-Sort (cont.)



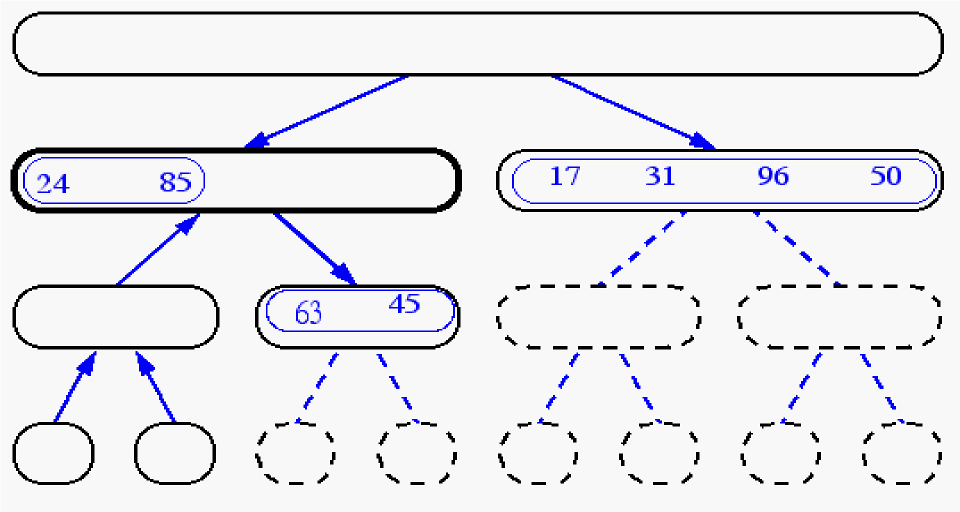


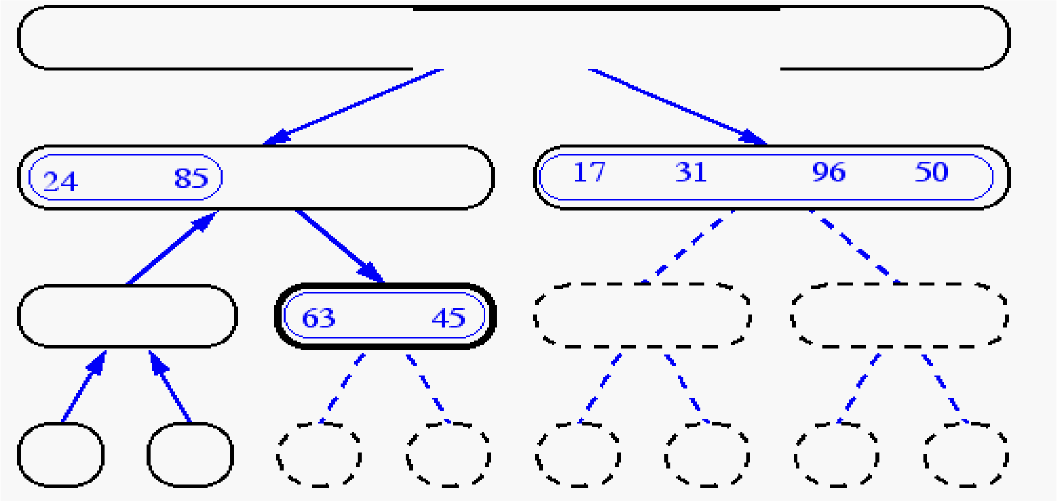
Merge-Sort (cont.)



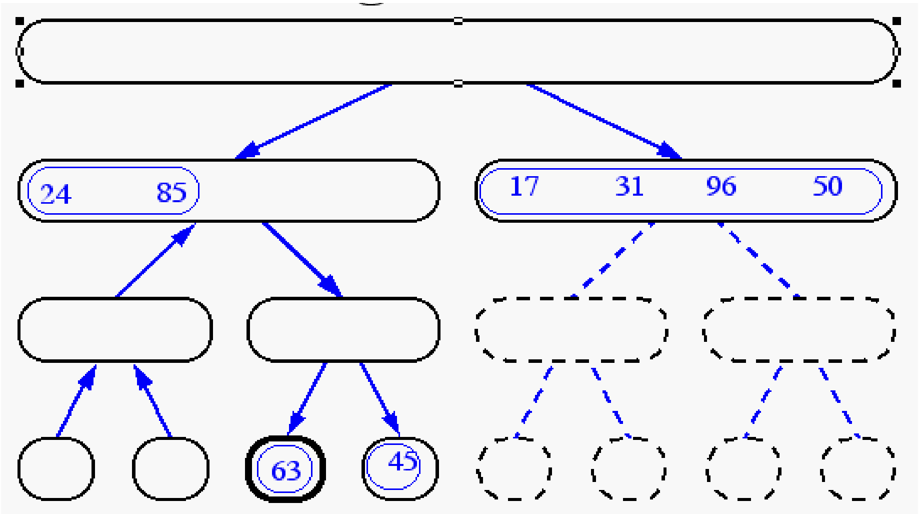


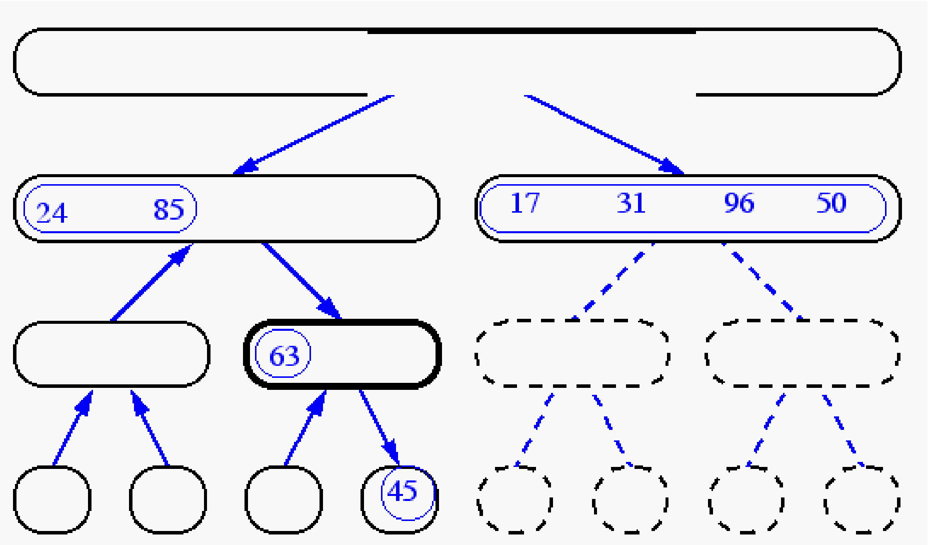
Merge-Sort (cont.)



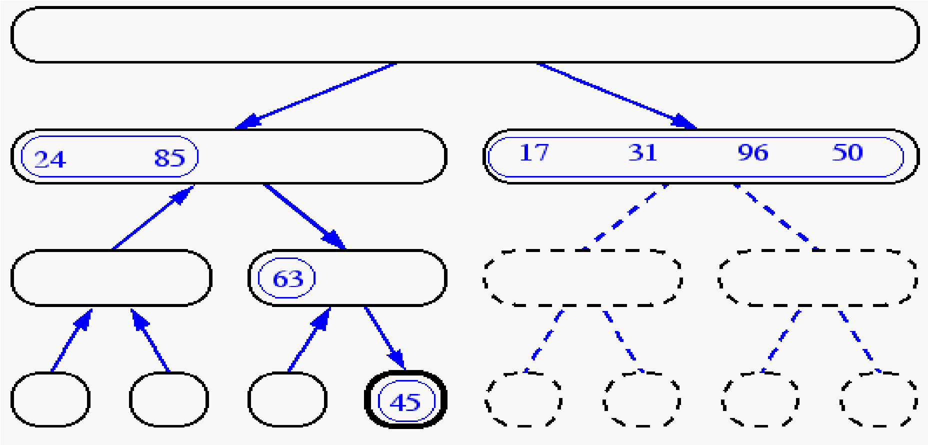


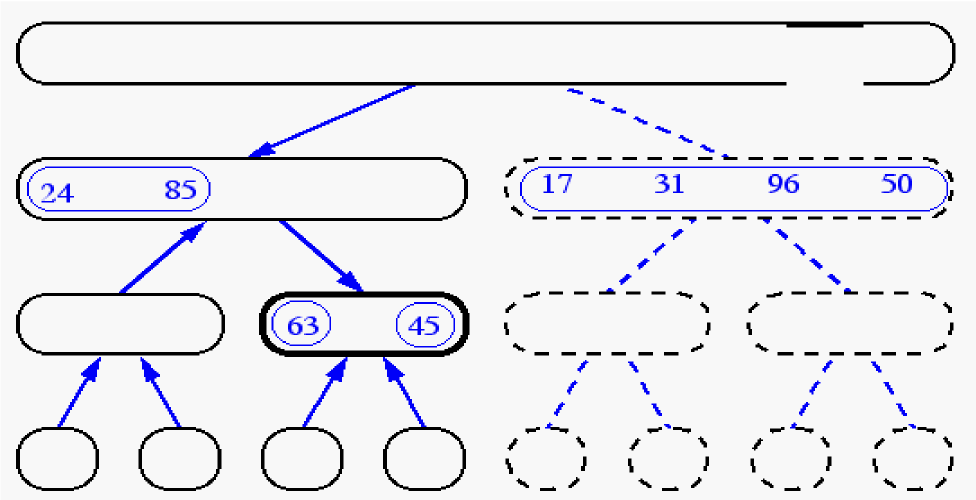
Merge-Sort (cont.)



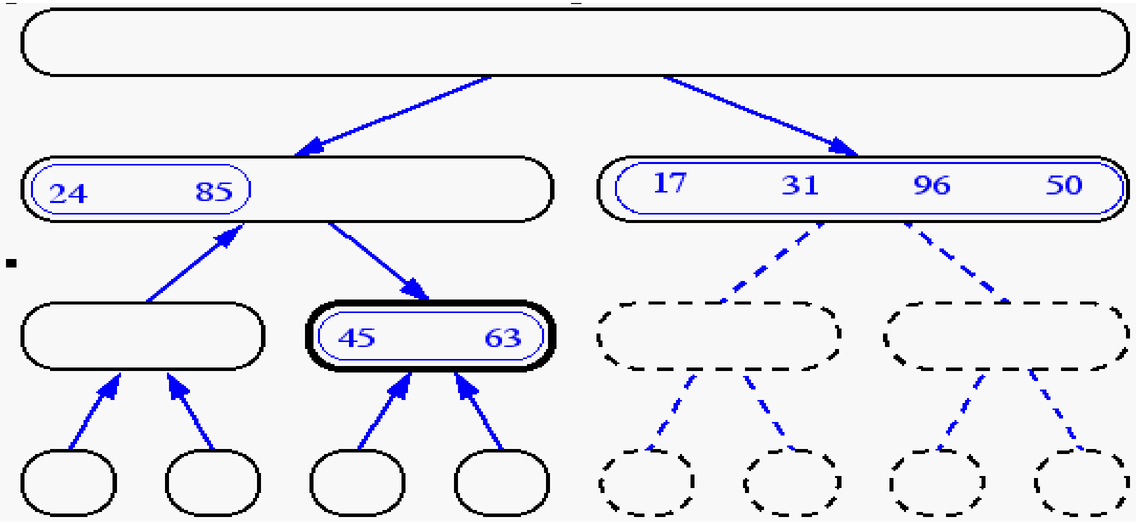


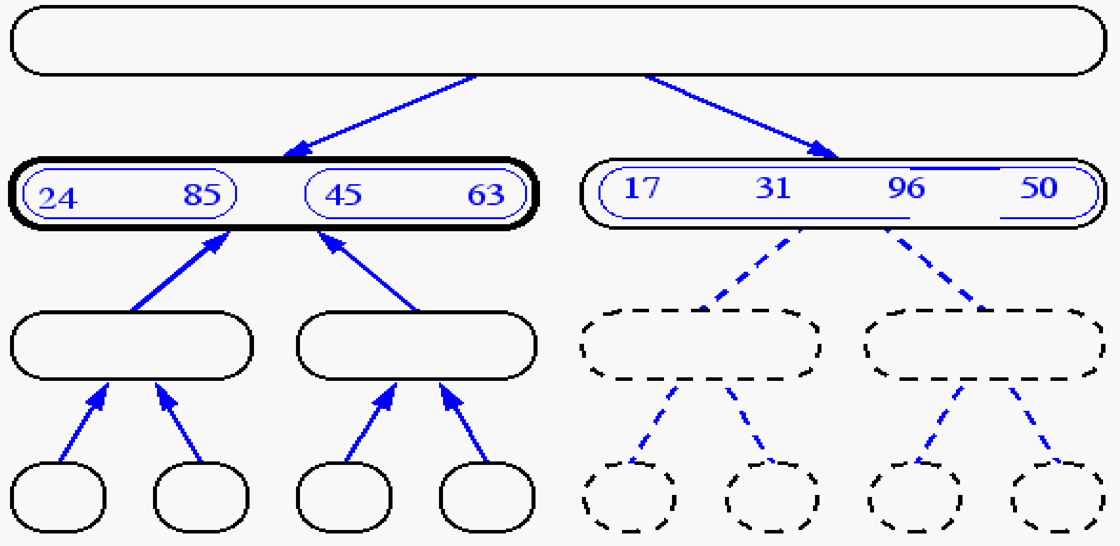
Merge-Sort (cont.)



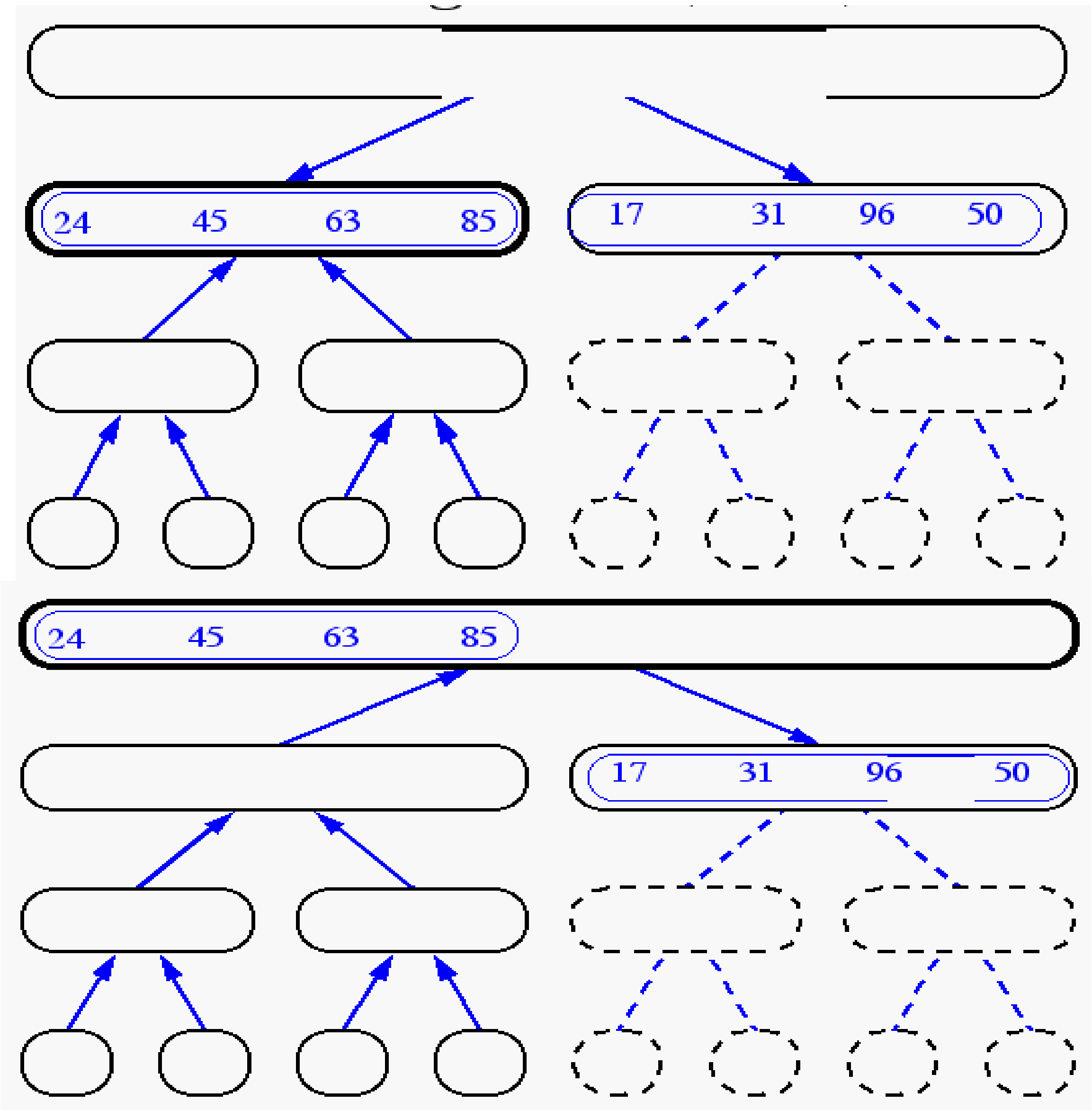


Merge-Sort(cont.)

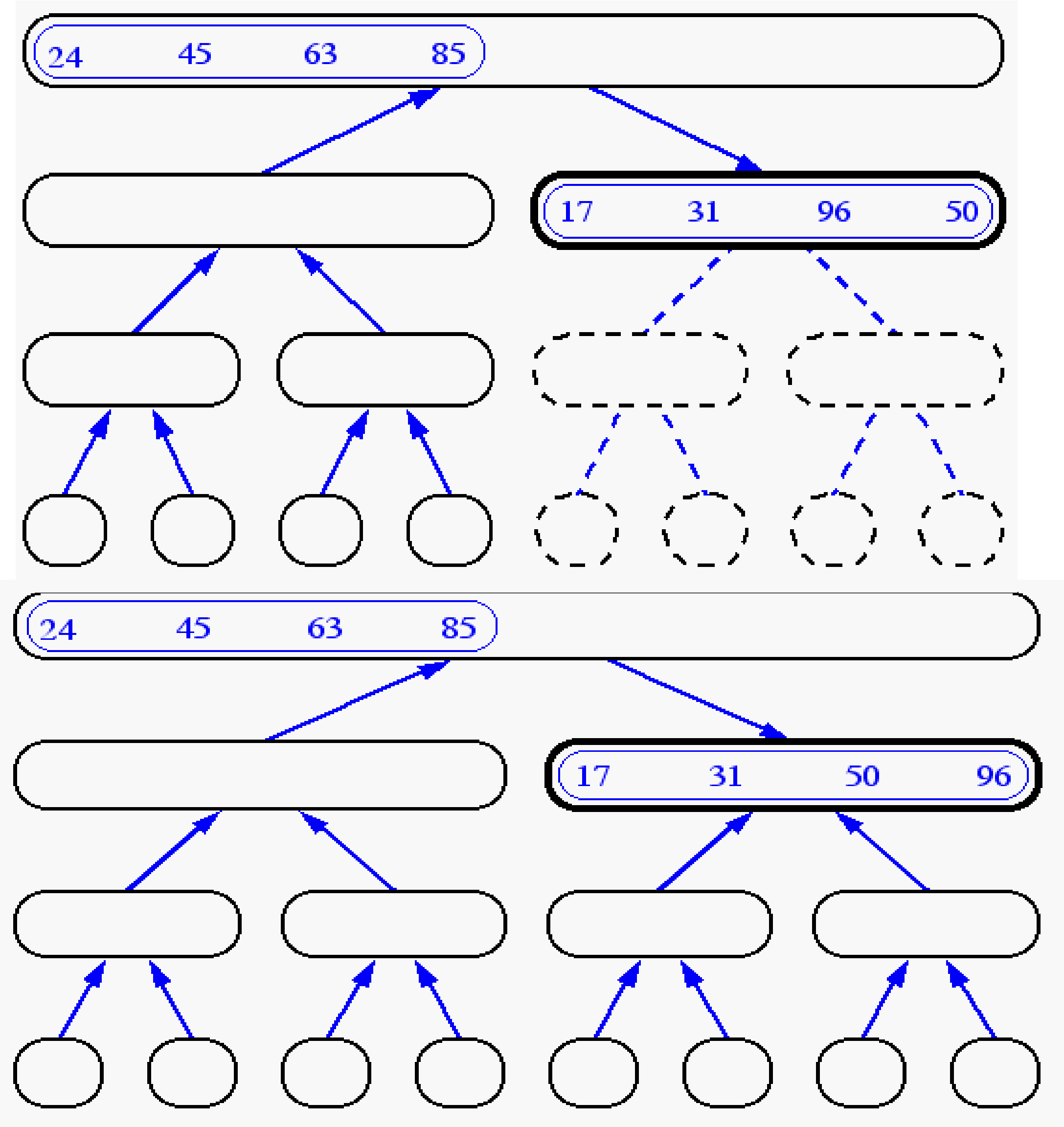




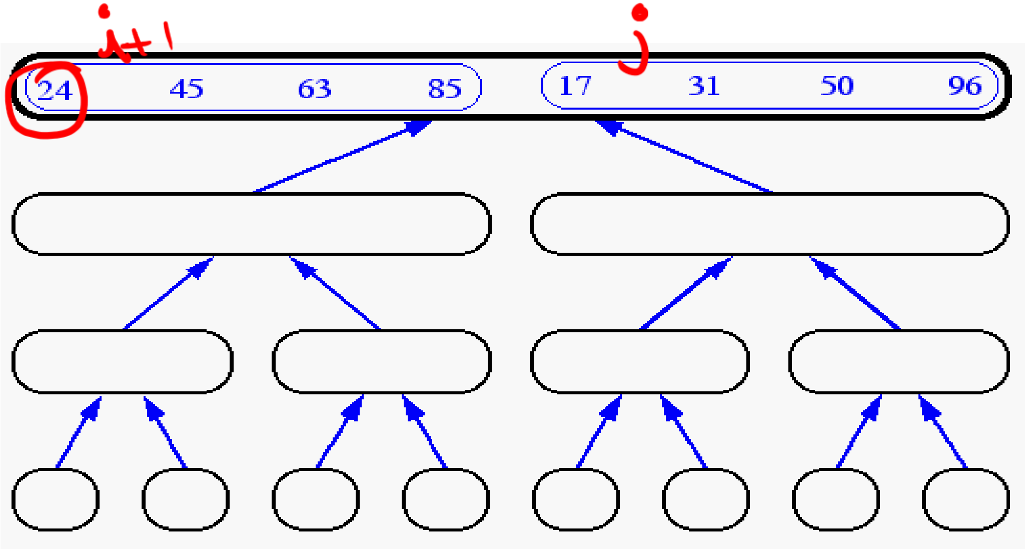
Merge-Sort (cont.)

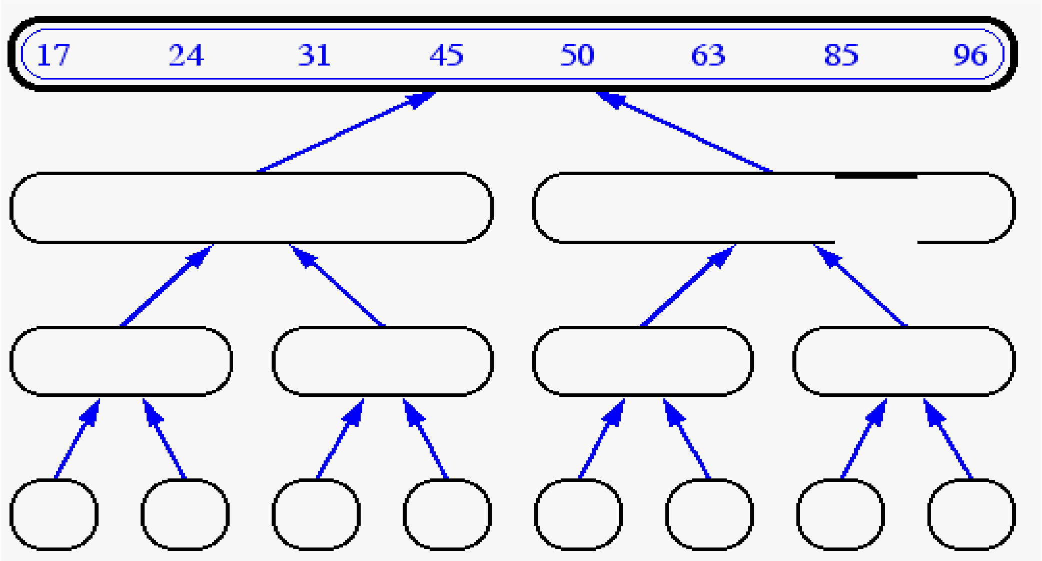


Merge-Sort (cont.)



Merge-Sort (cont.)





i

S_1 = 24 45 63 83

j = S_2 = 17 31 50 56

i = 1 , j = 1

while (S_1 on S_2 is not empty)

{

$\underline{S_1[i]} \overset{cond}{=} S_2[j]$

if $S_1[i] < S_2[j]$

add $S[i] \cdot S$

}

els $S[j] < S_1[i]$

$j = j+1$

combine

Merging Two Sequences

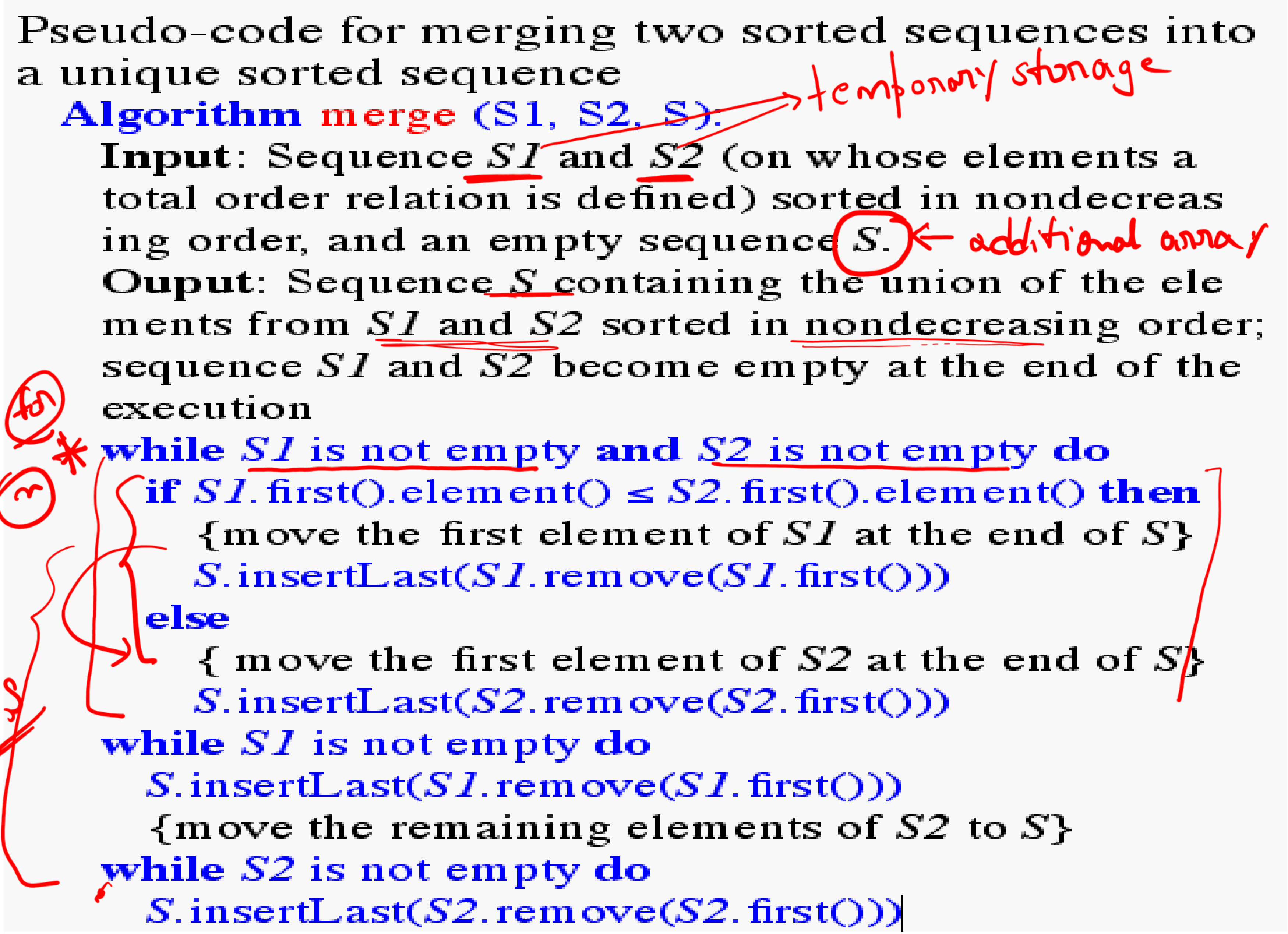
$S_1 = 2, 3$

$S_2 = 1, 7, 9$

$S_1 = \square$

$S_2 = 7, 9$

$S = 1, 2, 3, 7, 9$



Pseudo-code for merging two sorted sequences into a unique sorted sequence

```
Algorithm merge (S1, S2, S):
   Input: Sequence S1 and S2 (on whose elements a
   total order relation is defined) sorted in nondecreas
   ing order, and an empty sequence S.
   Ouput: Sequence S containing the union of the ele
   ments from S1 and S2 sorted in nondecreasing order;
   sequence S1 and S2 become empty at the end of the
   execution
   while S1 is not empty and S2 is not empty do
      if S1.first().element() ≤ S2.first().element() then
         {move the first element of S1 at the end of S}
         S.insertLast(S1.remove(S1.first()))
      else
         { move the first element of S2 at the end of S}
         S.insertLast(S2.remove(S2.first()))
   while S1 is not empty do
      S.insertLast(S1.remove(S1.first()))
      {move the remaining elements of S2 to S}
   while S2 is not empty do
      S.insertLast(S2.remove(S2.first()))
```

temporary storage

additional array

for

*

$S_1 = 1, 3, 7, 10$

$S_2 = 8, 10, 12, 13, 16, 20$

$S =$ array

$S_1[1]$ compare $S_2[1]$

→

$S_1 =$ empty

$S_2 = 10, 12, 13, 16, 20$

$S =$

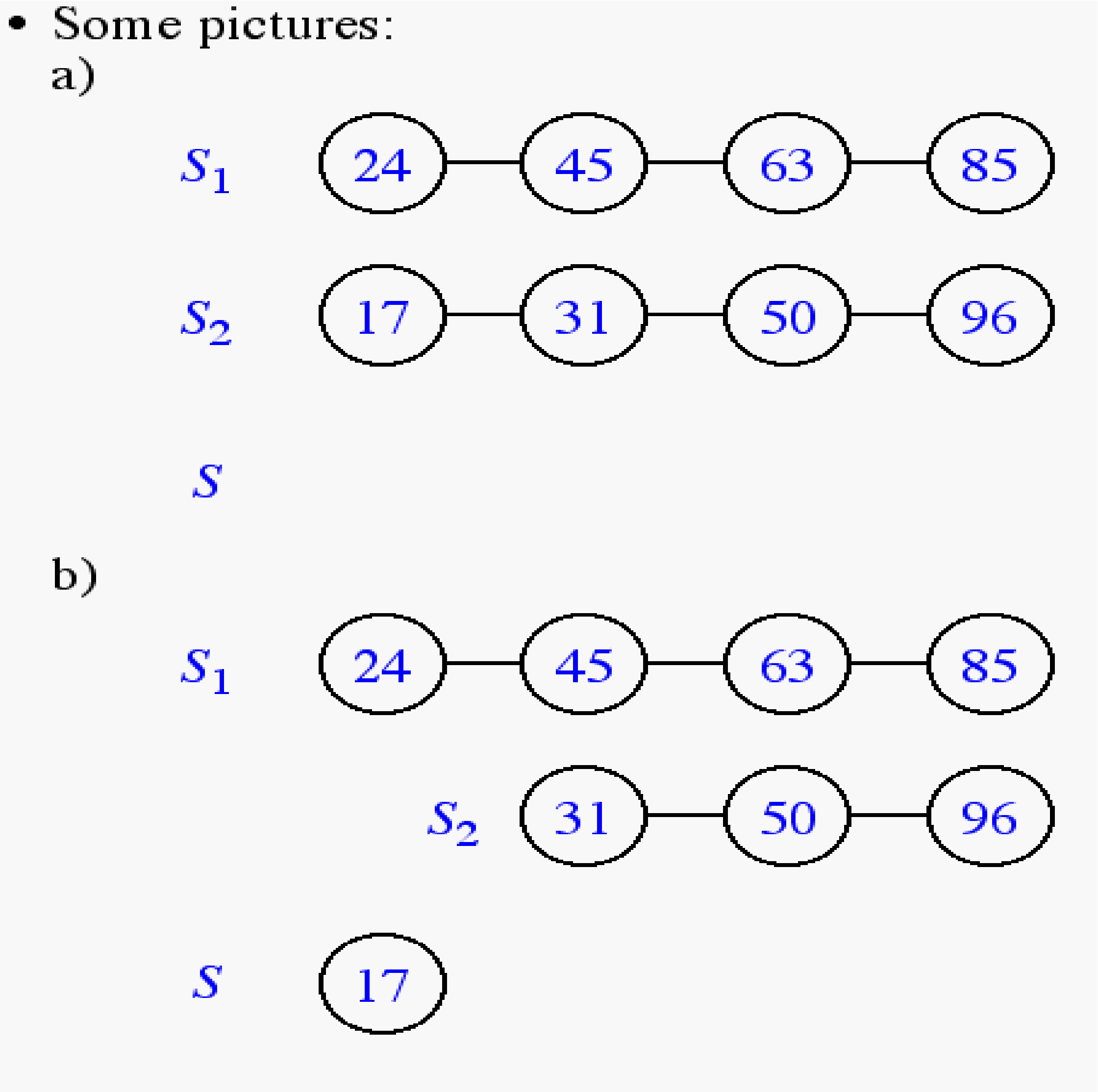
1	3	7	8	10	10	12	13	16	20

$S_1 = 10, 12, 13$

$S_2 =$ empty

S

Merging Two Sequences (cont.)

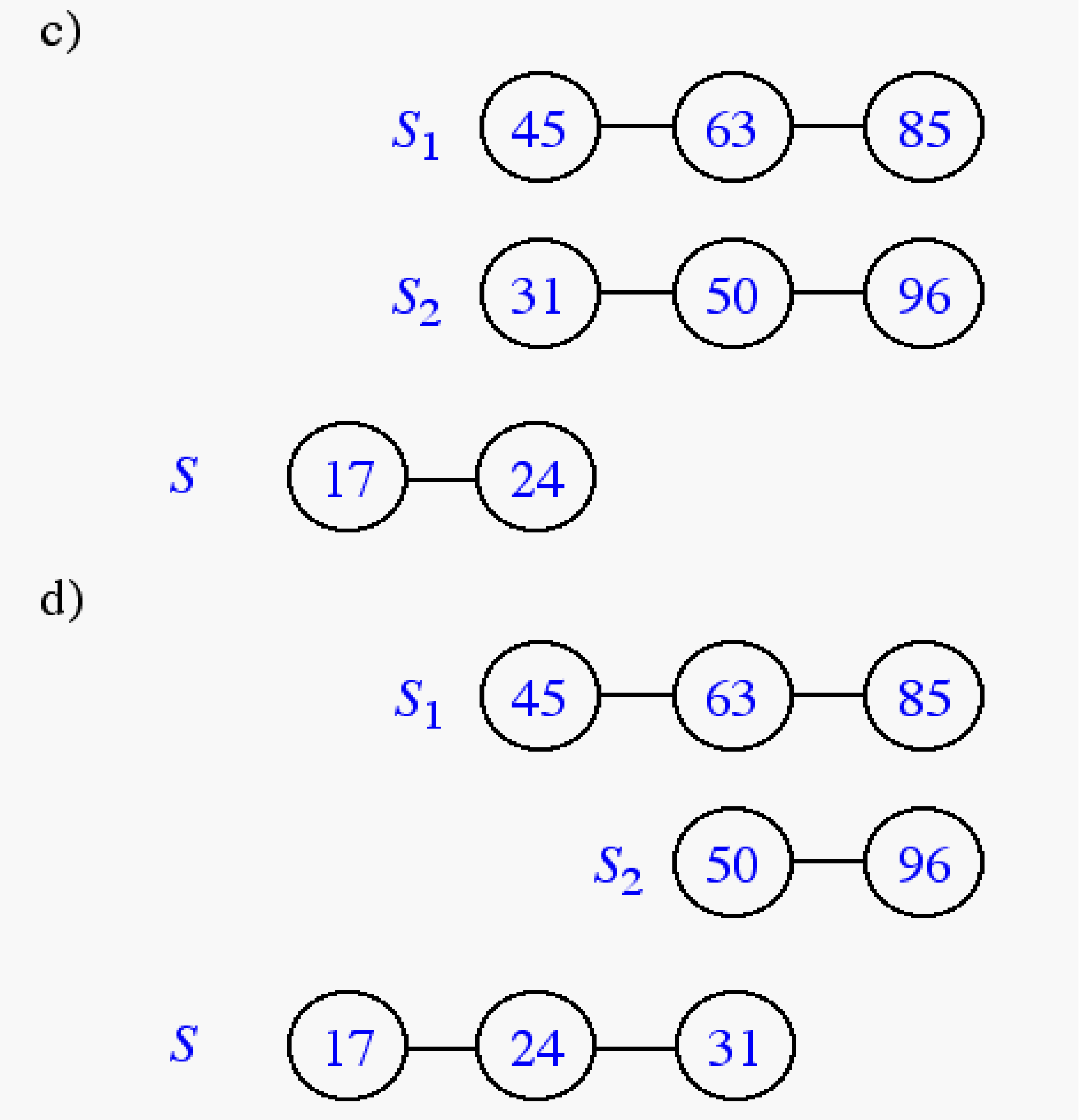


- Some pictures:

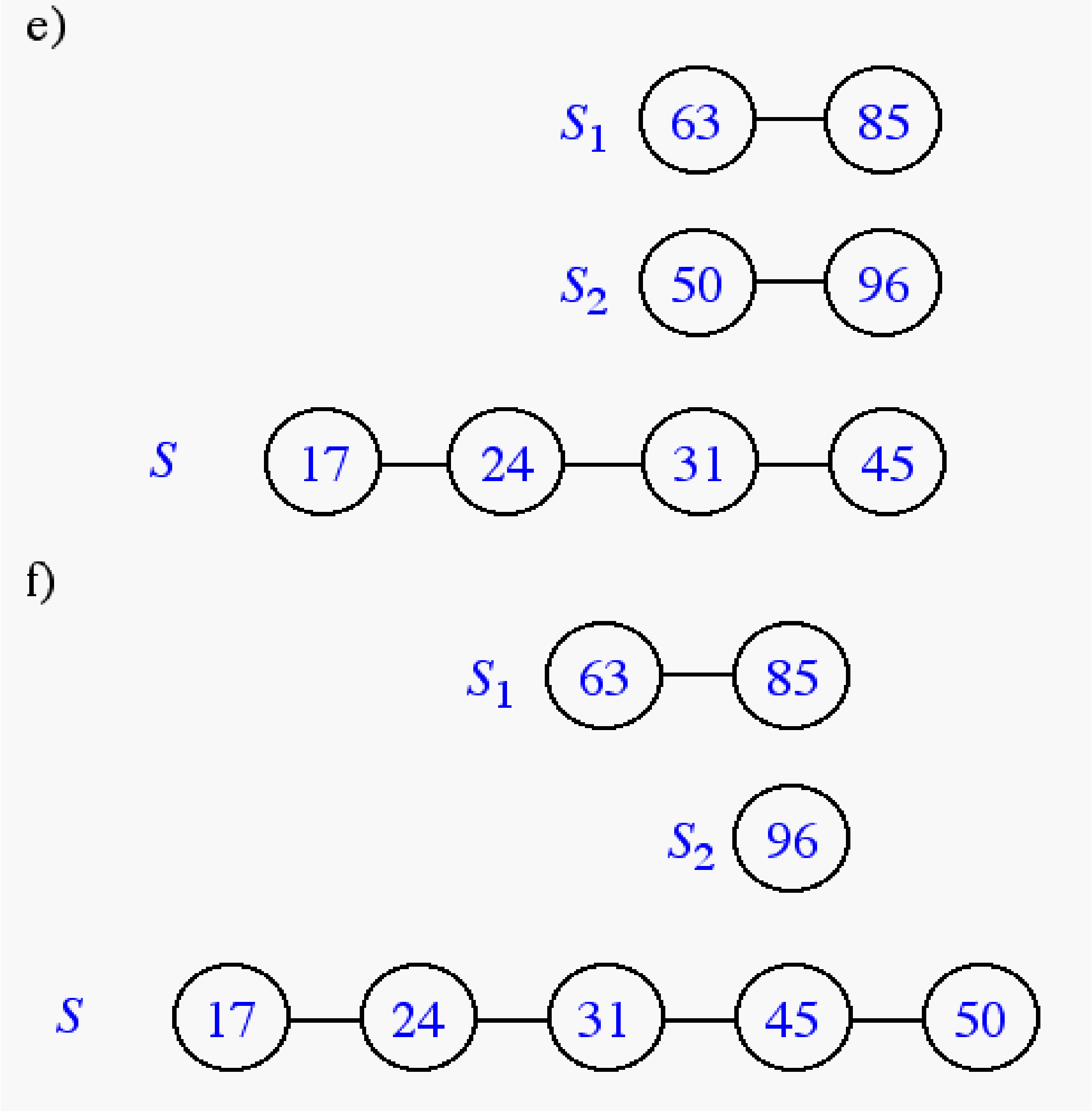
a)

b)

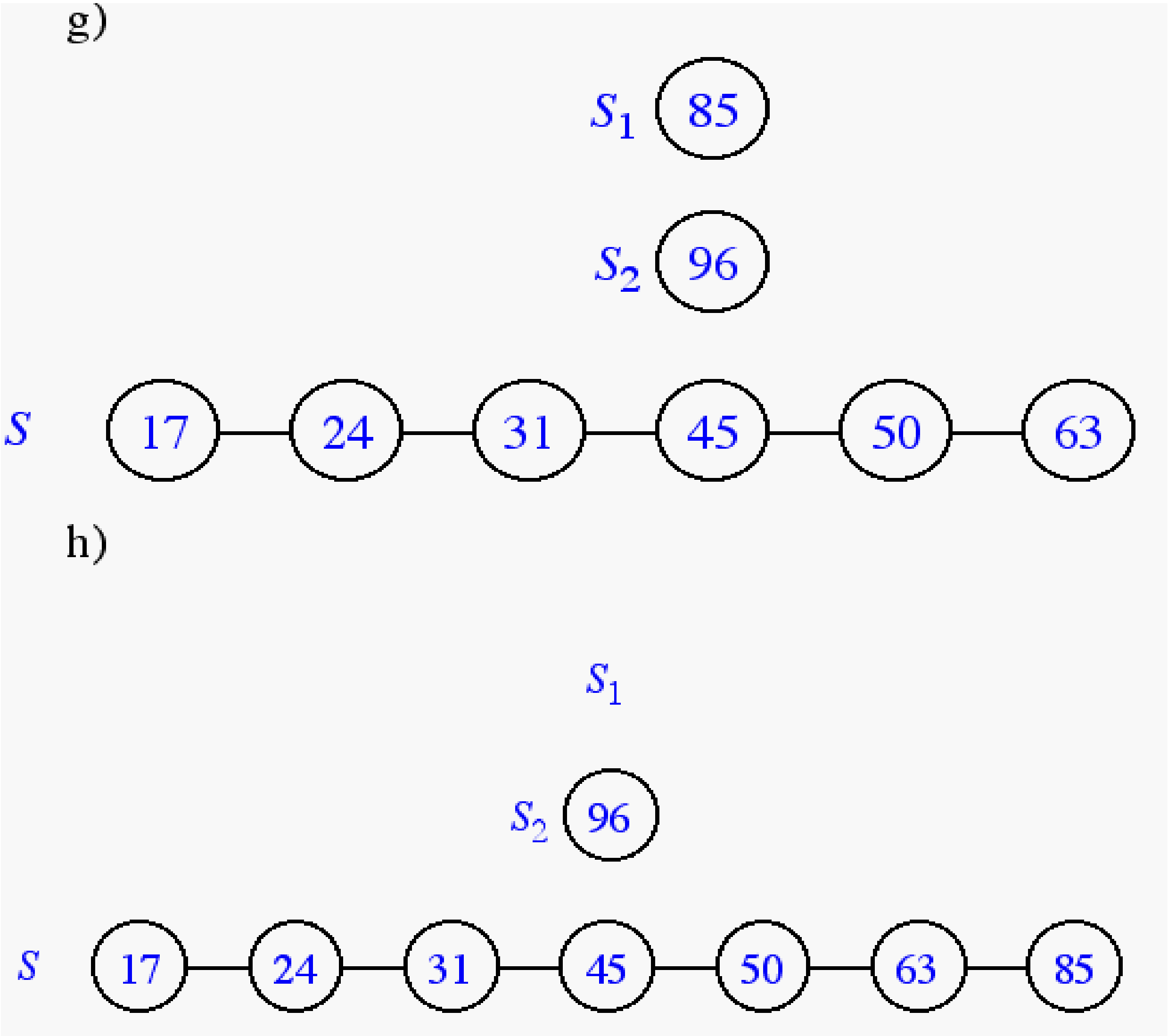
Merging Two Sequences (cont.)



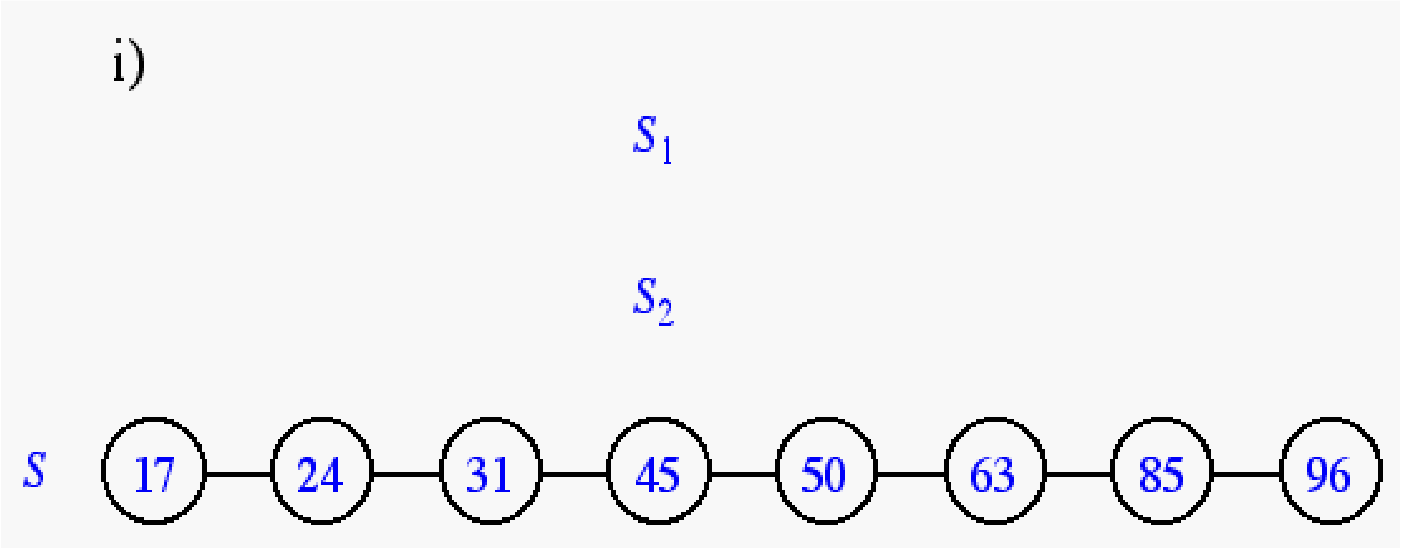
Merging Two Sequences (cont.)



Merging Two Sequences (cont.)

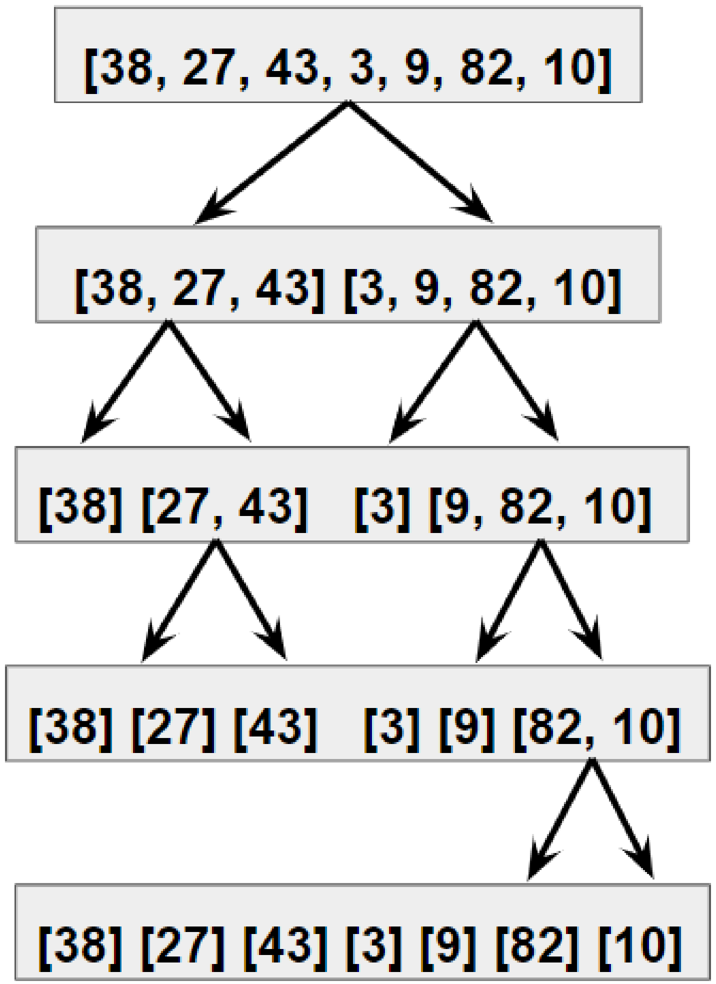


Merging Two Sequences (cont.)



Merge sort

Step 1: (Recursive Divide)



[38, 27, 43, 3, 9, 82, 10]

[38, 27, 43] [3, 9, 82, 10]

[38] [27, 43]   [3] [9, 82, 10]

[38] [27] [43]   [3] [9] [82, 10]

[38] [27] [43] [3] [9] [82] [10]

Step 2: (Conquer)

[27] [38] [3] [43] [9] [10] [82]

Step 3: Combine (Merge)

[27, 38] [3, 43] [9] [10, 82]

Final Sorted Array

[3, 9, 10, 27, 38, 43, 82]

Merge sort

```
#Procedure for MergeSort
MergeSort(arr):
    if length(arr) <= 1:
        return arr
    middle = length(arr) / 2
    left_half = MergeSort(arr[:middle])
    right_half = MergeSort(arr[middle:])
    return Merge(left_half, right_half)
```

```
#Procedure for Merge
Merge(left, right):
    result = []
    left_index = right_index = 0
    while left_index < length(left) and right_index < length(right):
        if left[left_index] < right[right_index]:
            result.append(left[left_index])
            left_index += 1
        else:
            result.append(right[right_index])
            right_index += 1
        result.extend(left[left_index:])
    result.extend(right[right_index:])
    return result
```

Asymptotic analysis of Merge Sort

It involves understanding its time complexity, which is consistently O(n log n) in the worst, average, and best cases. Let's break down the analysis step by step.

Time Complexity Analysis:

- **Divide:** Dividing the array of size n takes $O(1)$ time.
- **Conquer:** The recursive calls on subproblems occur until each sublist contains only one element, resulting in $O(\log n)$ levels of recursion.
- **Combine (Merge):** Merging two sorted sublists of size $n/2$ takes $O(n)$ time.

Overall Time Complexity of Merge Sort can be expressed using following **recurrence relation**

$$T(n) = 2T(n/2) + O(n). \longrightarrow n\log n$$

Asymptotic analysis of Merge Sort

It involves understanding its time complexity, which is consistently O(n log n) in the worst, average, and best cases. Let's break down the analysis step by step.

Time Complexity Analysis:

- **Divide:** Dividing the array of size n takes $O(1)$ time.
- **Conquer:** The recursive calls on subproblems occur until each sublist contains only one element, resulting in $O(\log n)$ levels of recursion.

- **Combine (Merge):** Merging two sorted sublists of size $n/2$ takes $O(n)$ time.

Overall Time Complexity of Merge Sort can be expressed using following **recurrence relation** (discussed in the later)

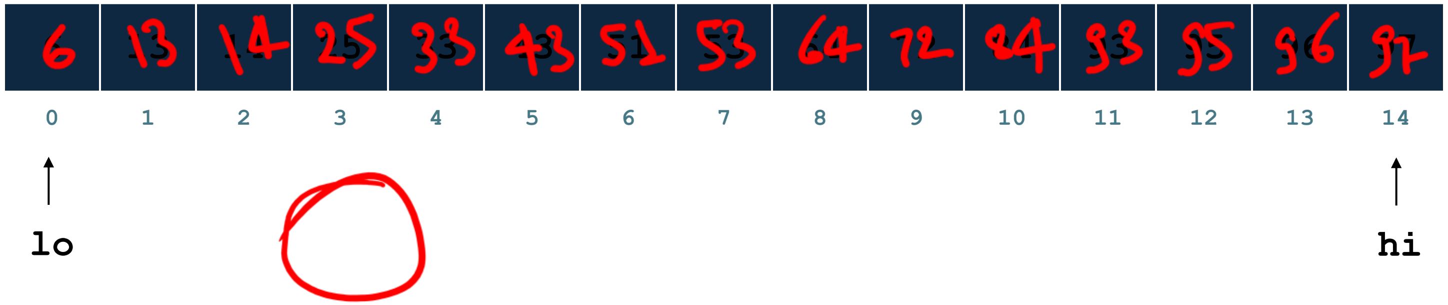
$$T(n) = 2T(n/2) + O(n).$$

Binary Search

$T(n) = T(n/2) + O(1)$

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` $\leq$ `value` $\leq$ `a[hi]`.



6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

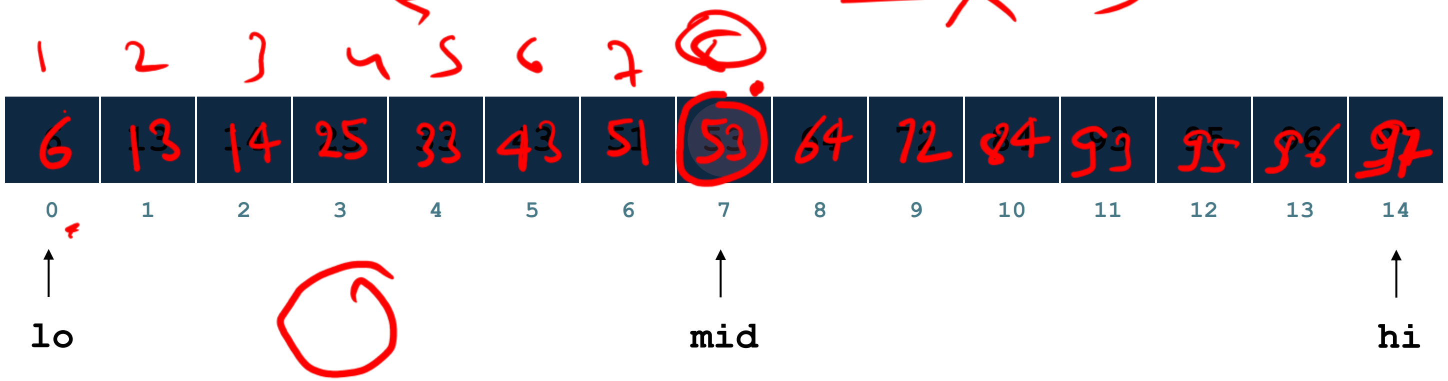
- Ex. Binary search for 33.

Binary Search → Divide & Conquer → the array should be sorted

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists. 15

- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

$\lceil \frac{15}{2} \rceil = 8$

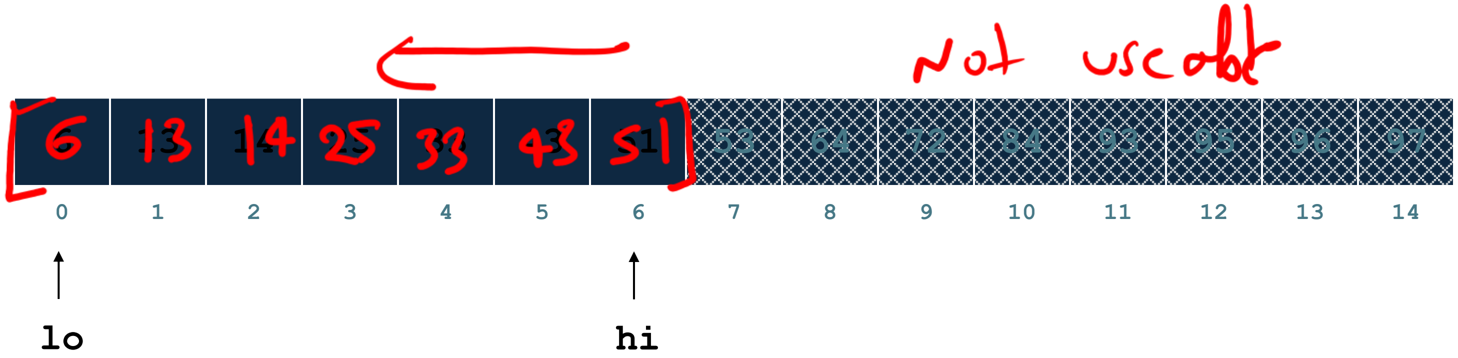


- Ex. Binary search for 33.

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.



Not usable

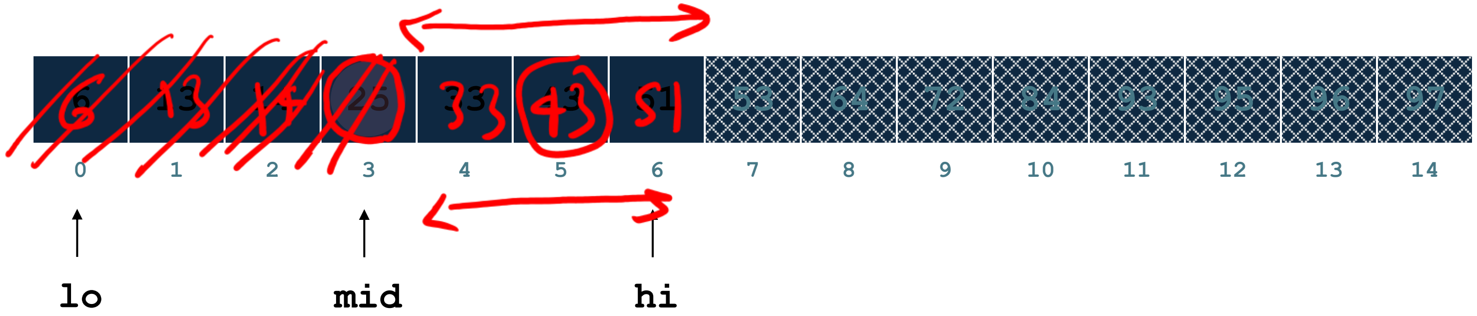
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- Ex. Binary search for 33.

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

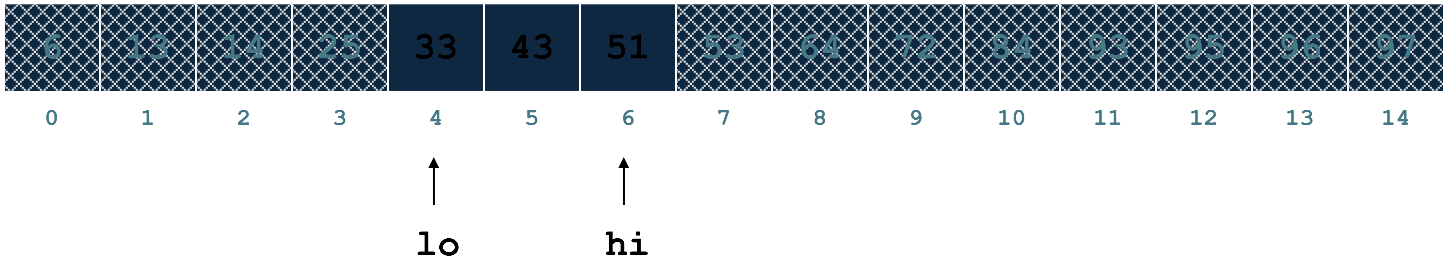


- Ex. Binary search for 33.

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.



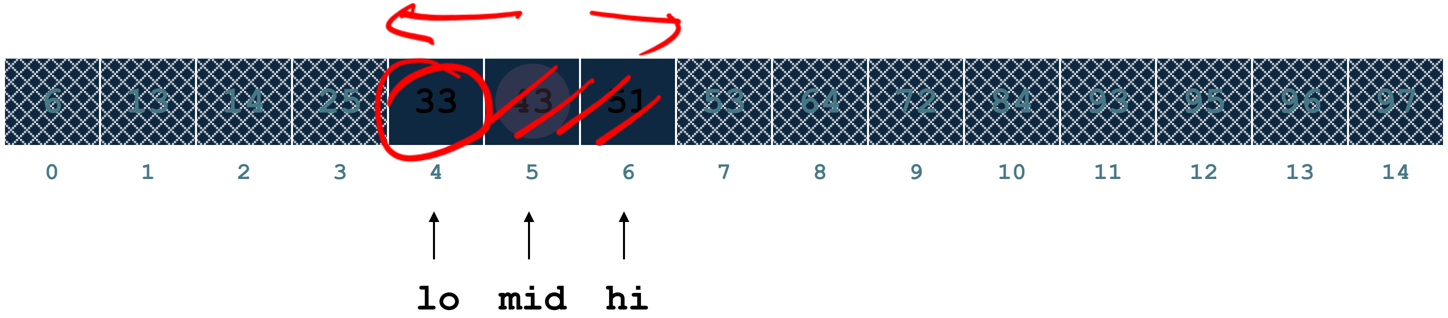
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- Ex. Binary search for 33.

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

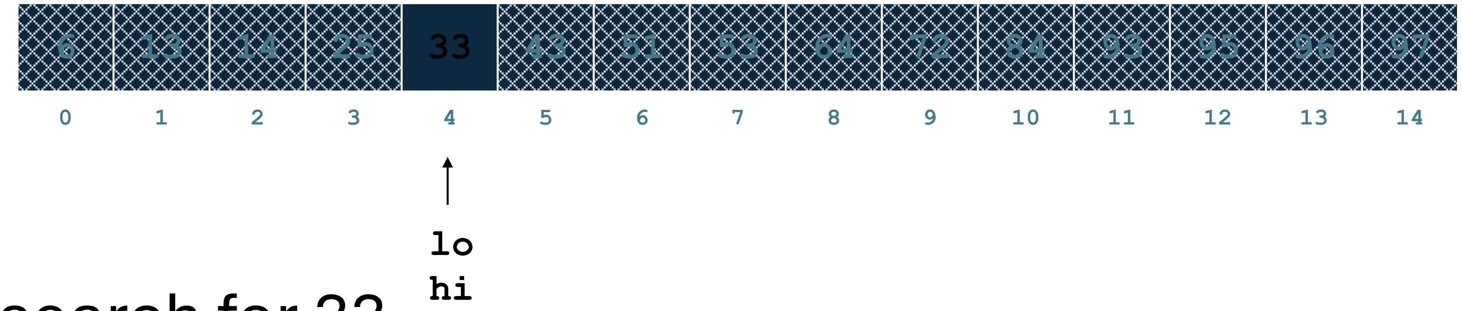


- Ex. Binary search for 33.

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.



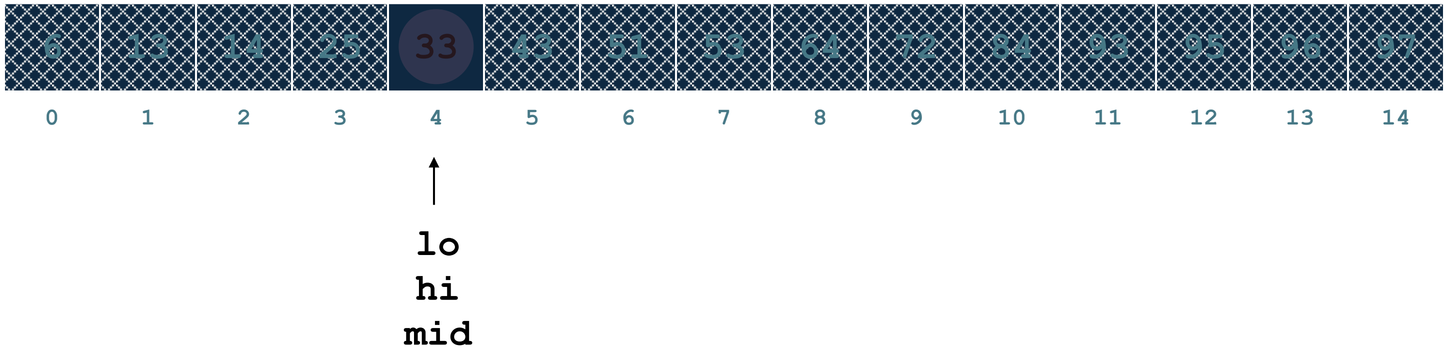
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

- Ex. Binary search for 33.

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

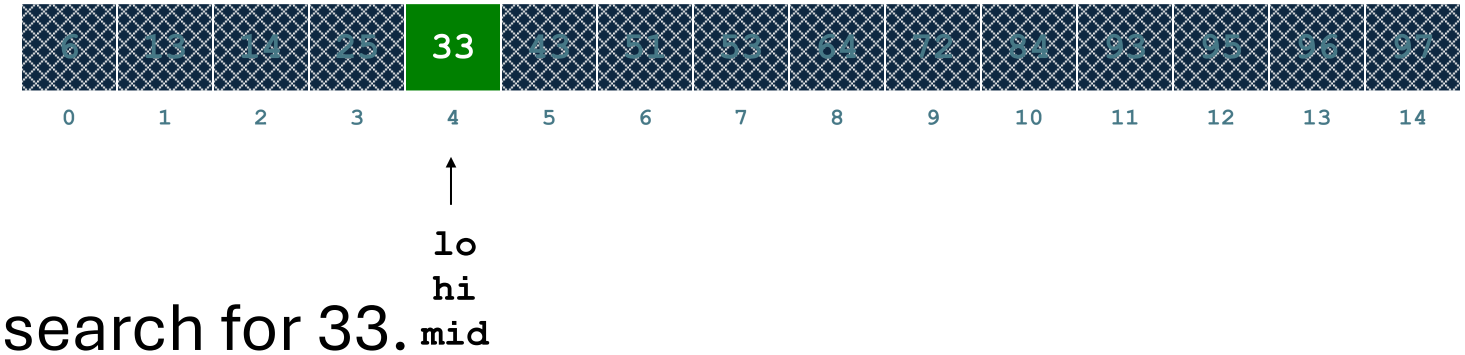


- Ex. Binary search for 33.

Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.



- Ex. Binary search for 33.

key = 33

$$A = \begin{array}{ccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 6, & 13, & 14, & 25, & 33, & 43, & 51, & 53, & 64, & 72, & 84, & 93, & 95, & 96, & 97 \end{array}$$

①

$\lceil n/2 \rceil = \lceil 15/2 \rceil = 8$ $\boxed{T(n) = T(n/2) + O(1)}$ $\underline{O(\log n)}$

A[mid] ⇒ 33   mid > 33 leftward

②

$$\begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ [6, & 13, & 14, & 25, & 33, & 44, & 51] \end{array}$$

discarded

$\lceil n/2 \rceil = \lceil 7/2 \rceil = 4$

25 = 33   33 > 25 Right

③

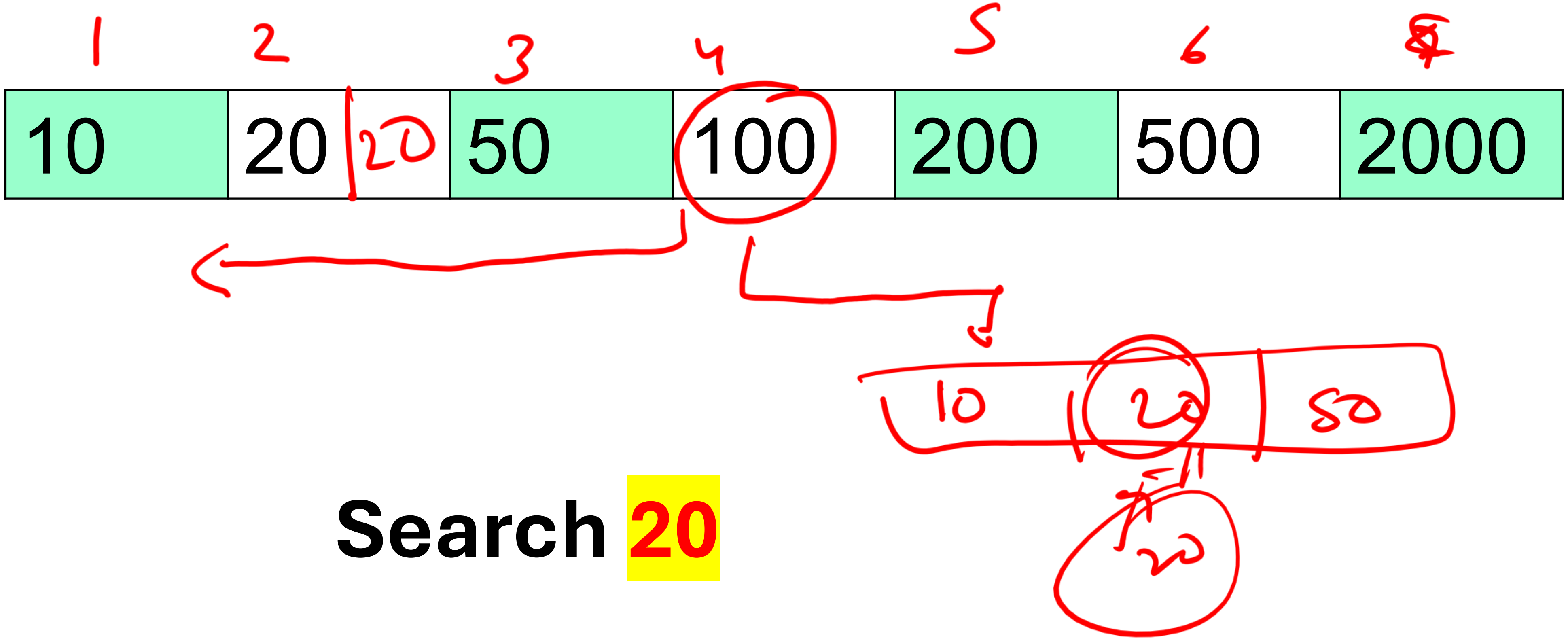
$$\begin{array}{ccc} 1 & 2 & 3 \\ [33 & 44 & 51] \end{array}$$

$\lceil n/2 \rceil = \lceil 3/2 \rceil = 2$

44 = 33   33 < 44 leftward

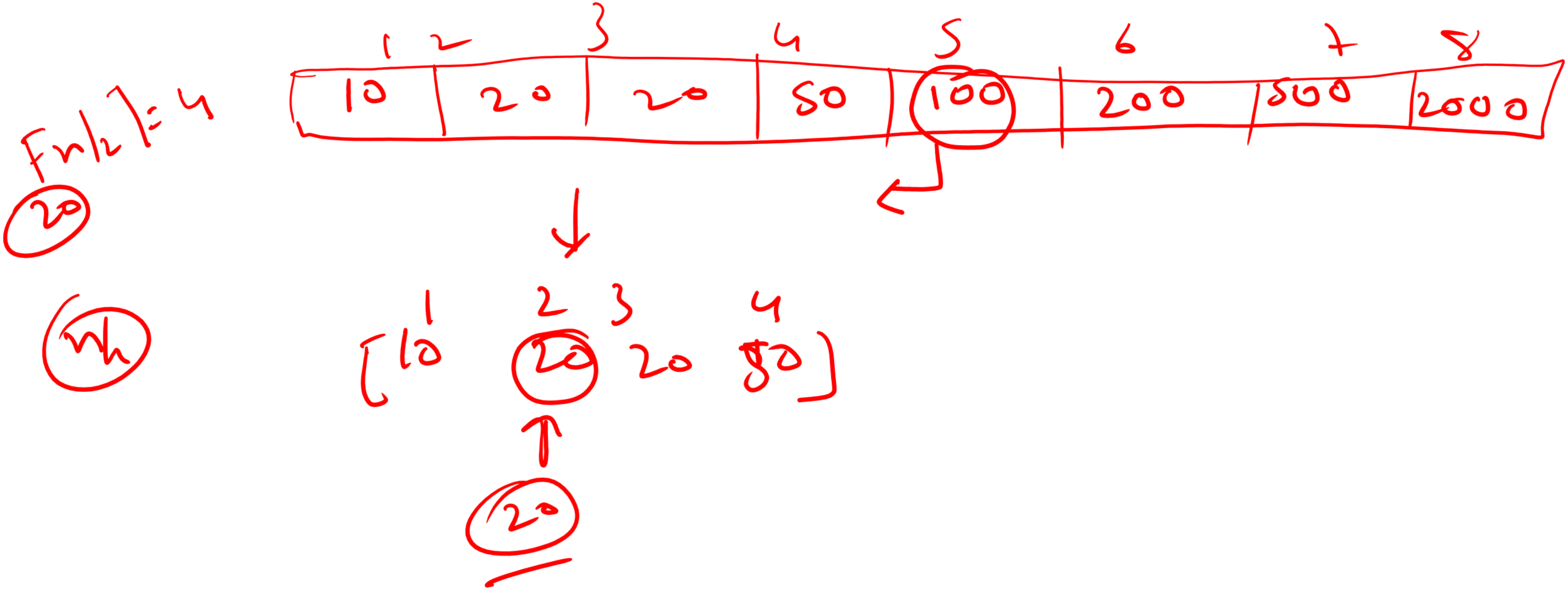
④

$\boxed{33} = 33$ ⇒ equal [illegible]



10	20	50	100	200	500	2000

Search 20



1	2	3	4	5	6	7	8
10	20	20	50	100	200	500	2000

Example: Binary Search

- Searching for an element k in a sorted array A with n elements
- Idea:
    - Choose the middle element A[n/2]
    - **If k == A[n/2],** we are done
    - **If k $\overset{>}{<}$ A[n/2],** search for k between A[0] and A[n/2 -1]
    - **If k > A[n/2],** search for k between A[n/2 + 1] and A[n-1]
    - Repeat until either k is found, or no more elements to search
- Requires a smaller number of comparisons than linear search in the worst case ($\log_2 n$ instead of n)