Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

MIT-WPU
।। विश्वशान्तिर्ध्रुवं ध्रुवा ।।

Fundamentals of Data Structures

S. Y. B. Tech CSE **Semester – III**

SCHOOL OF COMPUTER ENGINEERING AND TECHNOLOGY

Searching

- Searching is the process of determining whether or not a given value exists in a data structure or a storage media.
- Two searching methods are: **linear search and binary search**.

- The linear (or sequential) search algorithm on an array is:

  1. Sequentially scan the array, comparing each array item with the searched value.

  2. If a match is found; return the index of the matched element; otherwise return –1.

Searching

- When we maintain a collection of data, one of the operations we need is a search routine to locate desired data quickly.
- Here's the problem statement:

  Given a value X, return the index of X in the array, if such X exists. Otherwise, return NOT_FOUND (-1). We assume there are no duplicate entries in the array.

- We will count the number of comparisons in the algorithms

  - The ideal searching algorithm will make the least possible number of comparisons to locate the desired data.

  – Two separate performance analysis are normally done, one for successful search and another for unsuccessful search.

Linear Search Algorithm

```
Algorithm Search(array,target,size)
  {

 for i = 1 to n do
{
     if(array[i] = target)
    {
      print("Target data found")
    break;
    }
}
if(i >= size)
     print("Target not found")
}
```

Scan the array

```
Algorithm Search(array,target,size)
  {

 for i=1 to n do
{
     if(array[i] = target)
    {
      print("Target data found")
    break;
    }
}
if(i>size)
     print("Target not found")

}
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target,size)
   {

  for i=1 to n do
 {
       if(array[i] = target)
      {
        print("Target data found")
      break;
      }
 }
 if(i>=size)
       print("Target not found")

 }
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target)
   {

 for i=1 to n do
{
     if(array[i] = target)
    {
      print("Target data found")
    break;
    }
}
if(i>=size)
     print("Target not found")

}
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target,size)
  {

 for i=1 to n do
{
     if(array[i] = target)
    {
      print("Target data found")
    break;
    }
}
if(i>=size)
     print("Target not found")
 }
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target,size)
  {

 for i=1 to n do
{
     if(array[i] = target)
{
      print("Target data found")
    break;
}
}
if(i>=size)
     print("Target not found")

}
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target,size)
  {

 for i=1 to n do
{
     if(array[i] = target)
{
      print("Target data found")
    break;
}
}
if(i>=size)
     print("Target not found")
 }
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target,size)
  {

 for i=1 to n do
{
     if(array[i] = target)
{
      print("Target data found")
    break;
}
}
if(i>=size)
    print("Target not found")
 }
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target,size)
  {

 for i=1 to n d
{
      if(array[i]
     {
       print(“Target data found”)
     break;
     }
}
if(i>=size)
      print(“Target not found”)
  }
```

Target data found

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

```
Algorithm Search(array,target,size)
  {

 for i=1 to n do
{
     if(array[i] = target)
    {
     print("Target data found")
    break
    }
}
if(i>=size)
    print("Target not found")
 }
```

array

7	12	5	22	13	32
1	2	3	4	5	6

target = 13

Linear Search Analysis: Best Case

```
Algorithm Search(array,target,size)
  {

 for i=1 to n do
{
     if(array[i] = target)
    {
      print("Target data found")
    break;
     }
}
if(i>=size)
     print("Target not found")
  }
```

Best Case:
1 comparison

Best Case: match with the first item

target = 7

7	12	5	22	13	32

Linear Search Analysis: Worst Case

```
Algorithm Search(array,target,size)
  {

 for i=1 to n do
{
     if(array[i] = target)
    {
      print("Target data found")
    break;
    }
}
if(i>=size)
     print("Target not found")
}
```

Linear Search [Animation](https://yongdanielliang.github.io/animation/web/LinearSearchNew.html) :

https://yongdanielliang.github.io/animation/web/LinearSearchNew.html

Worst Case:
N comparisons

Worst Case: match with the last item (or no match)

7	12	5	22	13	32

target = 32

Sequential search

```
int  seqsearch(int key)
{
    for i= 0 to n
     {
       if(key==a[i])
        {
          pos=i;
          flag=1;
          break;
        }
     }
   if(flag==1)
    return pos;
   else
  return -1;
}
```

```
class search
{
  int a[20],n;
  public:
   int seqsearch(int);
   void accept();
   void display();
};
```

Variations of Sequential Search

- There are three such variations:
  - Sentinel search

Sentinel search

- The algorithm ends either when the target is found or when the last element is compared
- The algorithm can be modified to eliminate the end of list test by placing the target at the end of list as just one additional entry
- This additional entry at the end of the list is called as *sentinel*

Sentinel search

```
int  seqsearch_sentinel(int key)
{
   a[n]=key;   //place target at the end of the list
  While (key!=a[i])
   {
       i++;
   }

   if(i<n)
    return i;
   else
    return -1;   //not found
}
```

- a) Write C++ program to store roll numbers of student in array who attended training program in random order. Write function for searching whether particular student attended training program or not using linear search and sentinel search.
- b) Write C++ program to store roll numbers of student array who attended training program in sorted order. Write function for searching whether particular student attended training program or not using binary search and Fibonacci search.

Linear Search Performance

- We analyze the successful and unsuccessful searches separately.
- We count how many times the search value is compared against the array elements.
- Successful Search
  - – Best Case – 1 comparison
  - – Worst Case – N comparisons (N – array size)
- Unsuccessful Search
  - – Best Case = Worst Case – N comparisons

Binary Search

- If the array is sorted, then we can apply the binary search technique.
- The basic idea is straightforward. First search the value in the middle position.
- If X is less than this value, then search the middle of the left half next.
- If X is greater than this value, then search the middle of the right half next.

Binary Search :Scenario

We have a sorted array

We want to determine if a particular element is in the array

- **Once found, print or return (index, boolean, etc.)**
- **If not found, indicate the element is not in the collection**

7	12	42	59	71	86	104	212	

Binary Search Algorithm

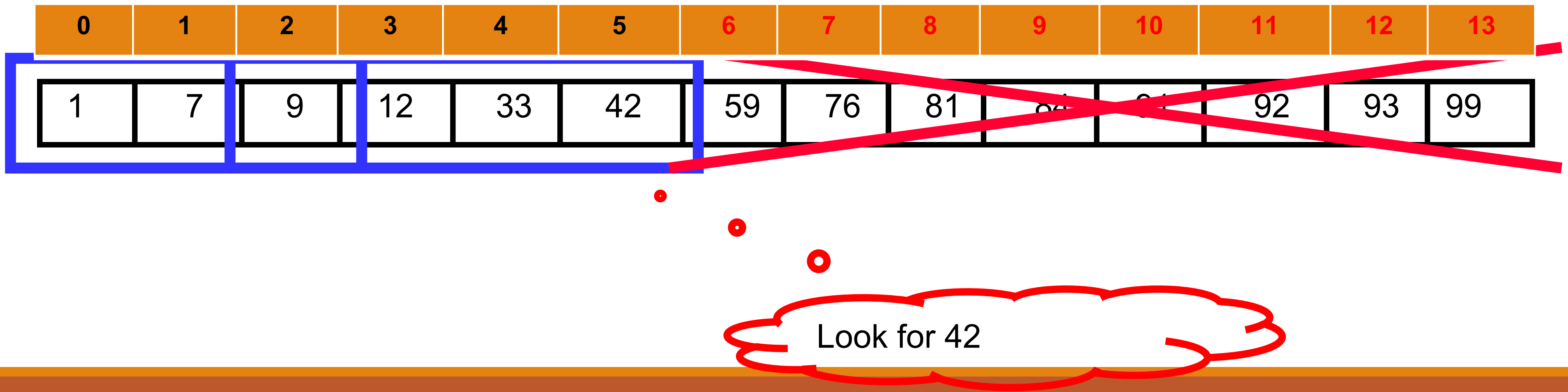
```
look at "middle" element

if no match then
   look left (if need smaller) or
        right (if need larger)
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	7	9	12	33	42	59	76	81	84	91	92	93	99

Look for 42

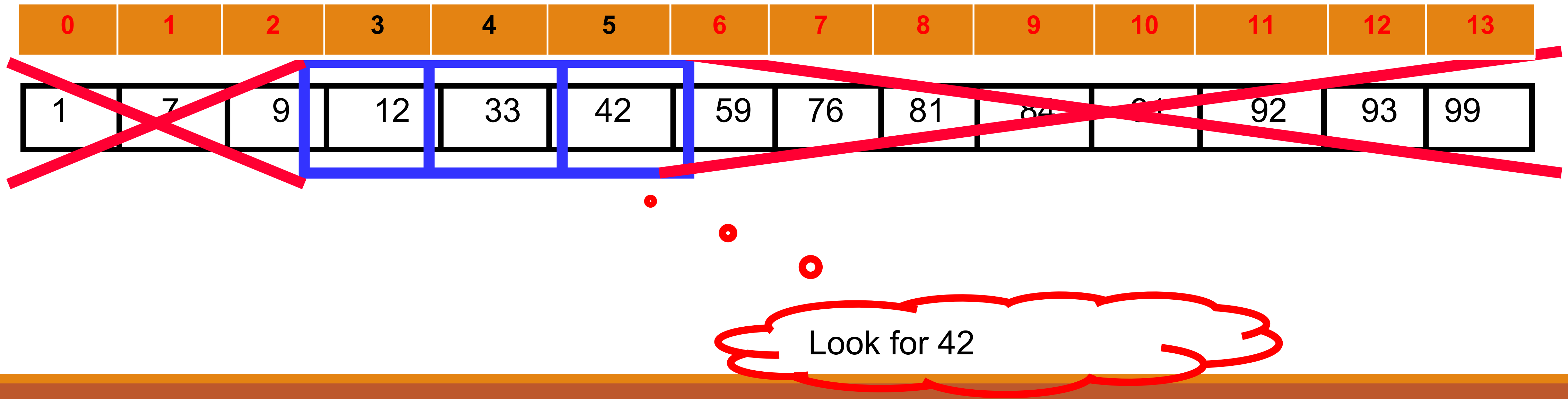
The Algorithm

```
look at "middle" element
if no match then
  look left or right
```



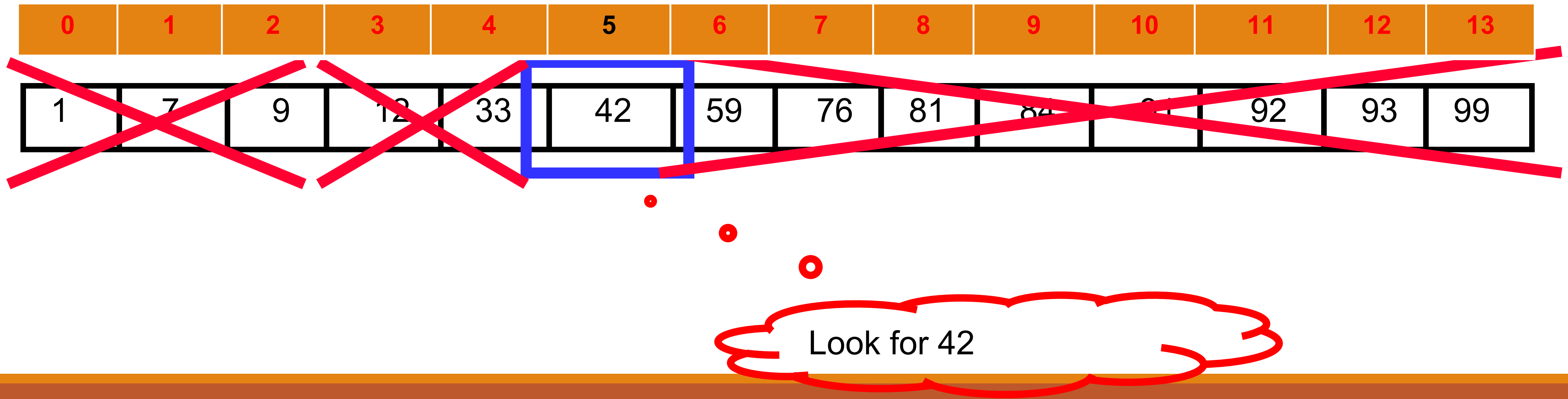
The Algorithm

```
look at “middle” element

if no match then

  look left or right
```



The Algorithm

```
look at "middle" element
if no match then
  look left or right
```



The Binary Search Algorithm

Return found or not found (true or false), so it should be a **function**.

When move ***left*** or ***right***, change the array boundaries

- We'll need a **first** and **last**

Animation:
https://www.cs.usfca.edu/~galles/visualization/Search.html

Time Complexity of Binary Search

The maximum no of elements after 1 comparison $=n/2$

The maximum number of elements after 2 comparisons $=n/2^2$

The maximum number of elements after h comparisons $=n/2^h$

For the lowest value of h elements 1 left

$n/2^h=1$ or $2^h=n$ $h=\log_2(n)=O(\log_2 n)$

Binary Search Algorithm

```
Algorithm binary_search(a[], low, high, key)
{
  while(low<=high)    {
     mid=(low+high)/2;
     if(a[mid]==key)      {
        flag=1;
         return flag; }      //end if
         else if (key<a[mid])
             high=mid-1;
        else
         low=mid+1;
   }   //end while
                                   If (flag==0)
                                   {   return flag;}

                                    }//end binary_search
```

Look for 42

1	7	9	12	33	42	59	76	81	84	91	92	93	99
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**

Binary search(recursive)

```
Algorithm  binary_search(a[], low, high, key)
{
  if(low<=high)  {
        mid=(low+high)/2;
        if(a[mid]==key)
              return mid;
         else if (key<a[mid])
              return binary_search(a,low,mid-1,key);
         else
            return binary_search(a,mid+1,high,key);}
  return -1;
}
```

Fibonacci Search

- Fibonacci search changes the binary search algorithm slightly
- Instead of halving the index for a search, a Fibonacci number is subtracted from it
- The Fibonacci number to be subtracted decreases as the size of the list decreases
- Note that Fibonacci search sorts a list in a non decreasing order
- Fibonacci search starts searching the target by comparing it with the element at *Fk*th location

Continued…

- Fibonacci numbers:

  `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`

  where each number is the sum of the preceding two.

- Recursive definition:
  - `F(0) = 0;`
  - `F(1) = 1;`
  - `F(number) = F(number-1)+ F(number-2);`

The different cases for the search are as follows:

Case 1: if equal the search terminates;

Case 2: if the target is greater and F_1 is 1, then the search terminates with an unsuccessful search;

else the search continues at the right of list with new values of low, high, and mid as

$\texttt{mid} = \texttt{mid} + F_2$, $F_1 = F_{k-4}$ and $F_2 = F_{k-5}$

Case 3: if the target is smaller and F_2 is 0, then the search terminates with unsuccessful search;

else the search continues at the left of list with new values of low, high, and mid as

$\texttt{mid} = \texttt{mid} - F_2$, $F_1 = F_{k-3}$ and $F_2 = F_{k-4}$

The search continues by either searching at the left of mid or at the right of mid in the list.

- $F_0 = 0$
- $F_1 = 1$
- $F_2 = 1$
- $F_3 = 2$
- $F_4 = 3$
- $F_5 = 5$
- $F_6 = 8$
- $F_7 = 13$
- $F_8 = 21$
- $F_9 = 34$
- $F_{10} = 55$

A=

0	1	2	3	4	5	6	7	8	9
6	14	23	36	55	67	76	78	81	89

Key =78

N=10

Compute F_k such that F_k>=10

Fib(7)=13 >10 hence k=7

Compute initial values of mid

Mid=n-F_{k-2}+ 1 F_1=F_{k-2} F_2=F_{k-3}

The target to be searched is compared with A[mid]

$F_{1=}$ fibo(7-2)=fibo(5)=5 $F_{2=}$ fibo(7-3)=fibo(4)=3

Mid=10-F_{k-2}+1=10-5+1=6 (76)

1. 78>A[6-1] (If f1!=1) mid=mid+F_2 =6+3=9
   F_1= F_1 – F_2= 5-3 so, F_1 =2
   F_2= F_2 - F1= 3 – 2 = **1**

2. 78<a[9-1] mid=mid- F_2 =9-1=**8**
   t= F_1 – F_2= 2 – 1 =1
   F_1= F_2 so, F_1 = F_2 **$F_{1=}$ 1**
   F_2 =t F_2=1

3. 78 <a[8] mid=mid-F_2 = 8-1 =7

4. 78==a[8-1]

- $F_0 = 0$
- $F_1 = 1$
- $F_2 = 1$
- $F_3 = 2$
- $F_4 = 3$
- $F_5 = 5$
- $F_6 = 8$
- $F_7 = 13$
- $F_8 = 21$
- $F_9 = 34$
- $F_{10} = 55$

A=6,14,23,36,55,67,76,78,81,89

Key =81

N=10

Compute F_k such that F_k>=10

Fib(7)=13 >10 hence k=7

Compute initial values of mid

Mid=n-F_{k-2}+ 1

$F_1=F_{k-2}$

$F_2=F_{k-3}$

The target to be searched is compared with A[mid]

$F_{1=}$ fibo(7-2)=fibo(5)=5

$F_{2=}$ fibo(7-3)=fibo(4)=3

Mid=10-F_{k-2}+1=10-5+1=6 (76)

1. 81>A[6] mid=mid+ F_2 =6+3=9

   $F_1=f_{k-4}=F_{7-4}=F_3$ F_1 =2

   $F_2=F_{k-5}=F_{2} =1$ F_2=1

2. 81<a[9] mid=mid-F2=9-1=8

   $F_1=f_{k-3}=F_{7-3}=F_4$ F_1 =3

   $F_2=F_{k-4}=F_{7-4}$ =F3 F_2=2

3. 81=a[8]

```
Algorithm fib_search(a,n)
{
    find fk >= n;
    initially f1= f_{k-2} ; f2=f_{k-3};
    mid=n-f_{k-2}+1
    while key != a[mid-1]
    {
    if (mid<0 or key>a[mid-1])
        {
        if f1==1  return -1;
            mid=mid+f2;
            f1=f1-f2
            f2=f2-f1
        }
    else
        {
            if f2==0
              return -1;
            mid = mid - f2
            t = f1 - f2
            f1 = f2
          f2 = t
        }
    }
      return mid - 1
}
```

Time Complexity of Fibonacci Search

- **Fibonacci search is more efficient than binary search for large- sized lists**

- **However, it is inefficient in case of small lists**

- **The number of comparisons is of the order of *n, and the time complexity is O(log(n))***

Sorting

General Sort Concepts

Sort Order :

- Data can be ordered either in ascending order or in descending order
- The order in which the data is organized, either ascending order or descending order, is called sort order

Sort Stability

- **A sorting method** is said to be stable if at the end of the method, identical elements occur in the same relative order as in the original unsorted set
- Sort Efficiency
- Sort efficiency is a measure of the relative efficiency of a sort

Continued…

Passes

- During the sorted process, the data is traversed many times
- Each traversal of the data is referred to as a sort pass
- In addition, the characteristic of a sort pass is the placement of one or more elements in a sorted list

Sorting

Sorting takes an unordered collection and makes it an ordered one.

1	2	3	4	5	6
77	42	35	12	101	5

1	2	3	4	5	6
5	12	35	42	77	101

Sorting

- ***Sorting*** is a process that organizes a collection of data into either ascending or descending order.

- An ***internal sort*** requires that the collection of data fit entirely in the computer's main memory.

- We can use an ***external sort*** when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk.

In Place Sort

- The amount of extra space required to sort the data is constant with the input size.

Stable sort algorithms

- A stable sort keeps equal elements in the same order
- This may matter when you are sorting data according to some characteristic

  Example: sorting students by test scores

original array		stably sorted	
Ann	98	Ann	98
Bob	90	Joe	98
Dan	75	Bob	90
Joe	98	Sam	90
Pat	86	Pat	86
Sam	90	Zöe	86
Zöe	86	Dan	75

Unstable sort algorithms

- An unstable sort may or may not keep equal elements in the same order
- Stability is usually not important, but sometimes it is important

Ann	98	Joe	98
Bob	90	Ann	98
Dan	75	Bob	90
Joe	98	Sam	90
Pat	86	Zöe	86
Sam	90	Pat	86
Zöe	86	Dan	75
original array		unstably sorted	

Types of Sorting Algorithms

There are many, many different types of sorting algorithms, but the primary ones are:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Shell Sort
- Heap Sort
- Quick Sort
- Radix Sort

Bubble sort("Bubbling Up" the Largest Element)

Traverse a collection of elements

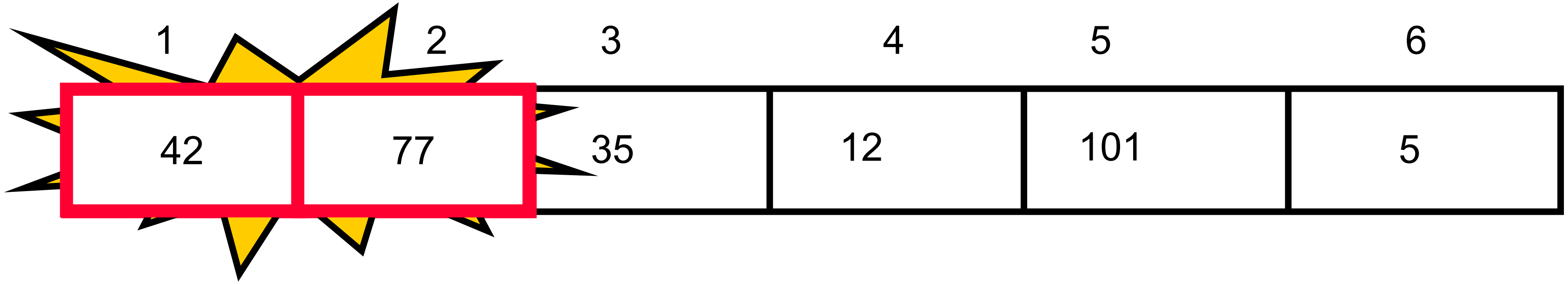
- **Move from the front to the end**
- **"Bubble" the largest value to the end using pair-wise comparisons and swapping**

1	2	3	4	5	6
77	42	35	12	101	5

"Bubbling Up" the Largest Element

Traverse a collection of elements

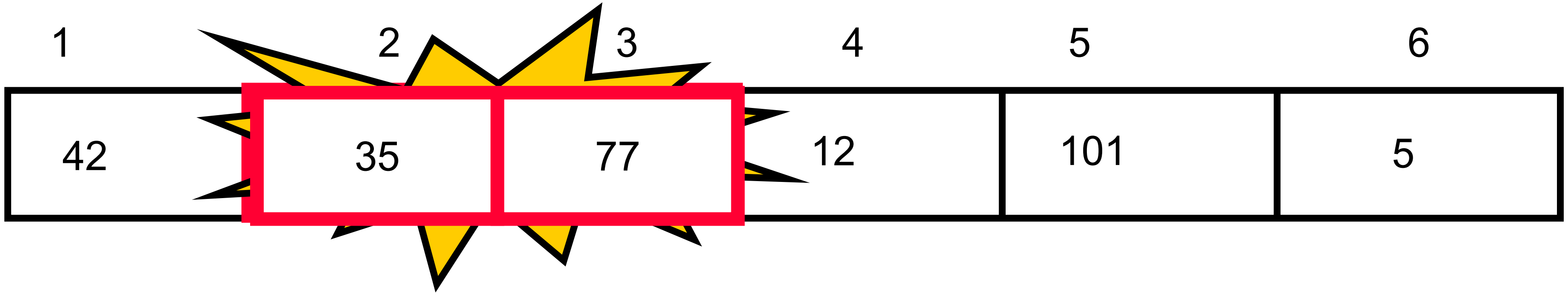
- Move from the front to the end
- “Bubble” the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

Traverse a collection of elements

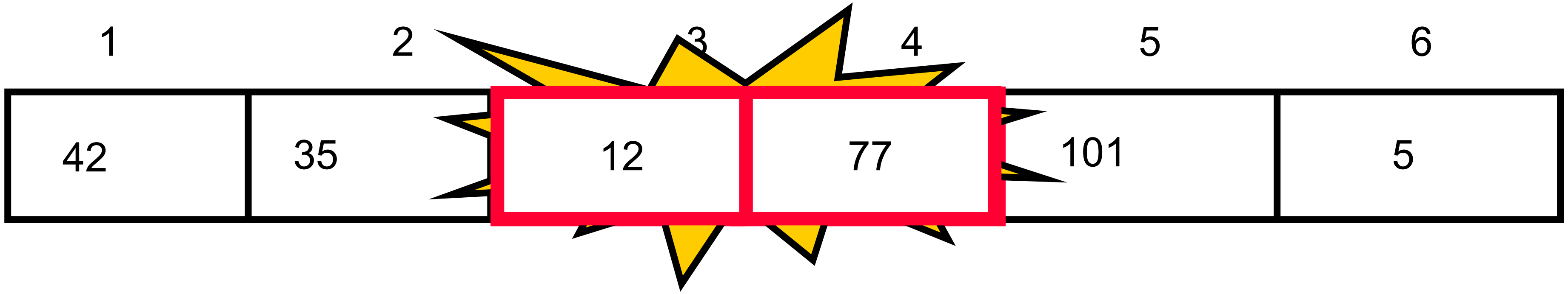
- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

Traverse a collection of elements

- **Move from the front to the end**
- **“Bubble” the largest value to the end using pair-wise comparisons and swapping**



"Bubbling Up" the Largest Element

Traverse a collection of elements

- **Move from the front to the end**
- **“Bubble” the largest value to the end using pair-wise comparisons and swapping**

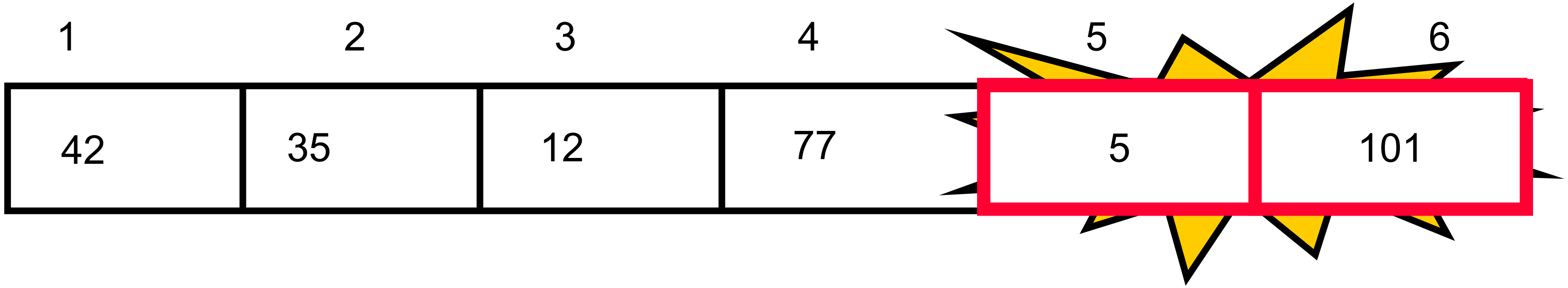
1	2	3	4	5	6
42	35	12	77	101	5

No need to swap

"Bubbling Up" the Largest Element

Traverse a collection of elements

- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping



"Bubbling Up" the Largest Element

Traverse a collection of elements

- Move from the front to the end
- “Bubble” the largest value to the end using pair-wise comparisons and swapping

1	2	3	4	5	6
42	35	12	77	5	$\boxed{101}$

Largest value correctly placed

"Bubbling" All the Elements

N - 1

1	2	3	4	5	6
42	35	12	77	5	101

1	2	3	4	5	6
35	12	42	5	77	101

1	2	3	4	5	6
12	35	5	42	77	101

1	2	3	4	5	6
12	5	35	42	77	101

1	2	3	4	5	6
5	12	35	42	77	101

Reducing the Number of Comparisons

1	2	3	4	5	6
77	42	35	12	101	5

1	2	3	4	5	6
42	35	12	77	5	101

1	2	3	4	5	6
35	12	42	5	77	101

1	2	3	4	5	6
12	35	5	42	77	101

1	2	3	4	5	6
12	5	35	42	77	101

Bubble sort

```
Algorithm  bubble()
{
        for  i =0 to n-1
         {
          for   j=0   to n-i-1
           {
               if  a[j]>a[j+1]
               {
                swap(a[j],a[j+1])
             }
         }
      display(a,n);
}
```

Analysis of Bubble Sort

The time complexity for each of the cases is given by the following:

Average-case complexity = $O(n^2)$

Best-case complexity = $O(n^2)$

Worst-case complexity = $O(n^2)$

Selection Sort

Idea:

- Find the smallest element in the array
- Exchange it with the element in the first position
- Find the second smallest element and exchange it with the element in the second position
- Continue until the array is sorted

Disadvantage:

- Running time depends only slightly on the amount of order in the file

Selection Sort

```
void selectionsort()
{
    for i = 0 to n-2 {
     minpos = i;
        for  j = i+1  to n-1  {
            if a[j] < a[minpos]  {
        minpos = j; }
    }
        if (minpos!=i ){
            temp = a[i];
        a[i] = a[minpos];
        a[minpos] = temp;
    }
}
```

8	4	6	9	2	3	1

Example

8	4	6	9	2	3	1

1	4	6	9	2	3	8

1	2	6	9	4	3	8

1	2	3	9	4	6	8

1	2	3	4	9	6	8

1	2	3	4	6	9	8

1	2	3	4	6	8	9

1	2	3	4	6	8	9

Analysis of Selection Sort

Best Case: $O(n^2)$

Worst Case: $O(n^2)$

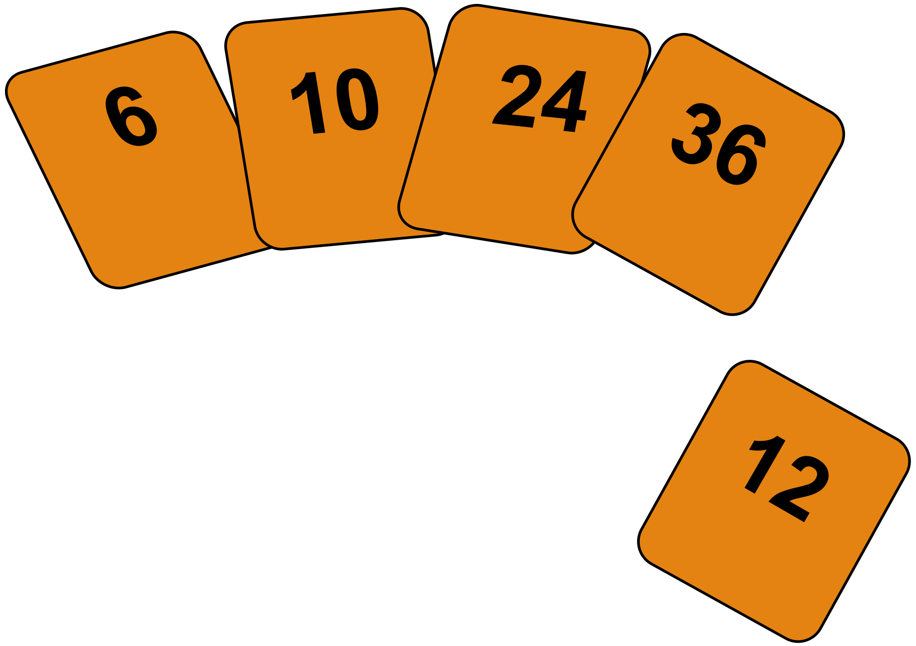
Average case: $O(n^2)$

Insertion Sort

Idea: like sorting a hand of playing cards

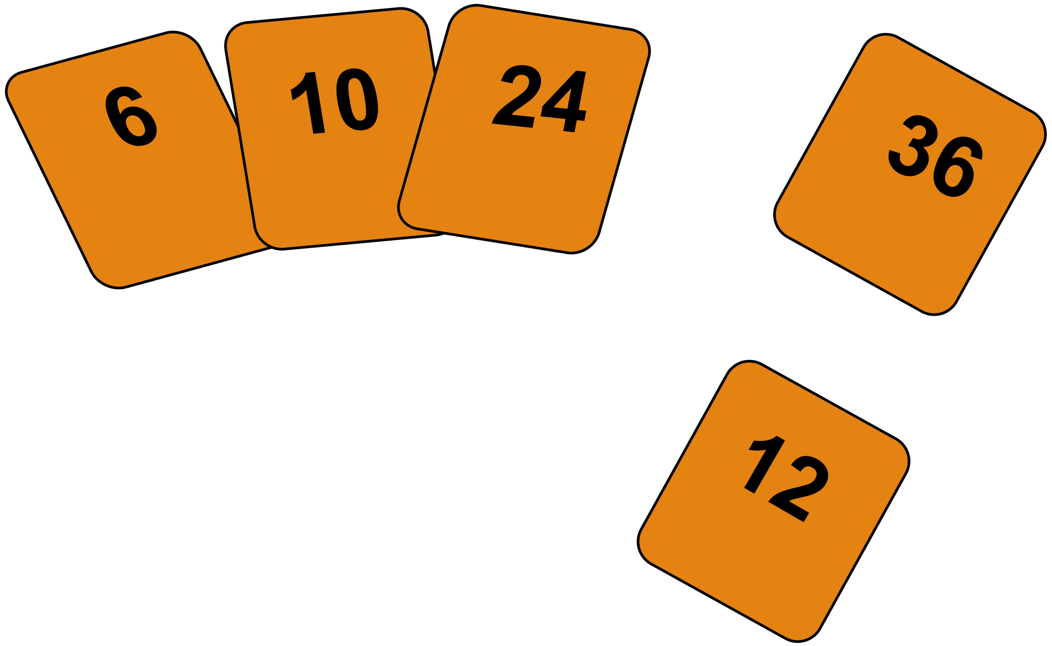
- Start with an empty left hand and the cards facing down on the table.
- Remove one card at a time from the table, and insert it into the correct position in the left hand
  - compare it with each of the cards already in the hand, from right to left
- The cards held in the left hand are sorted
  - these cards were originally the top cards of the pile on the table

Insertion Sort

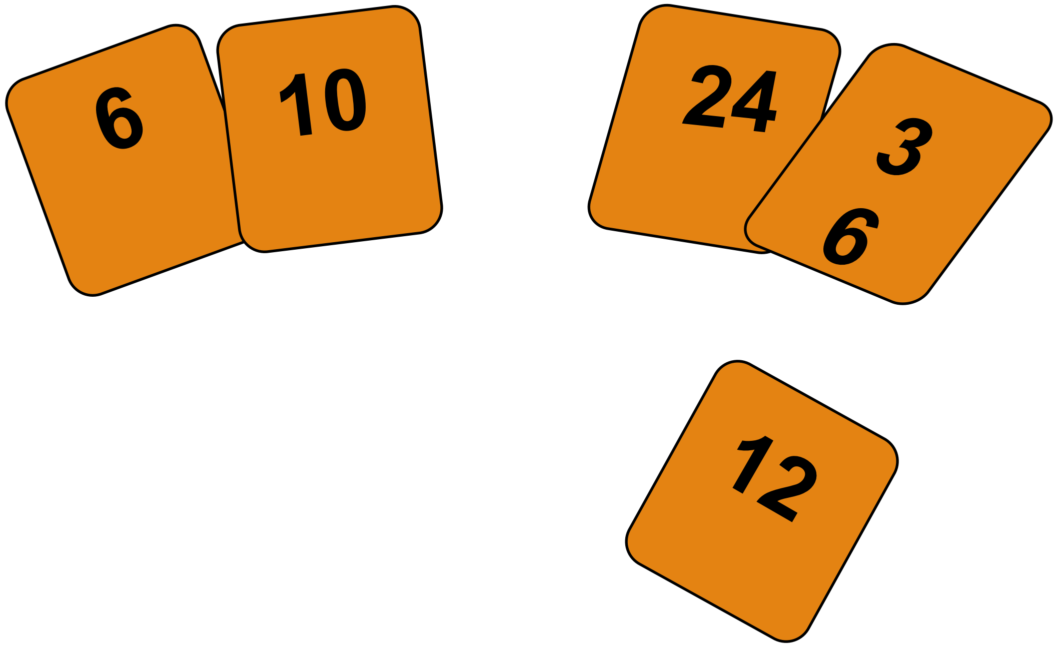


To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort



Insertion Sort

input array

5 2 4 6 1 3

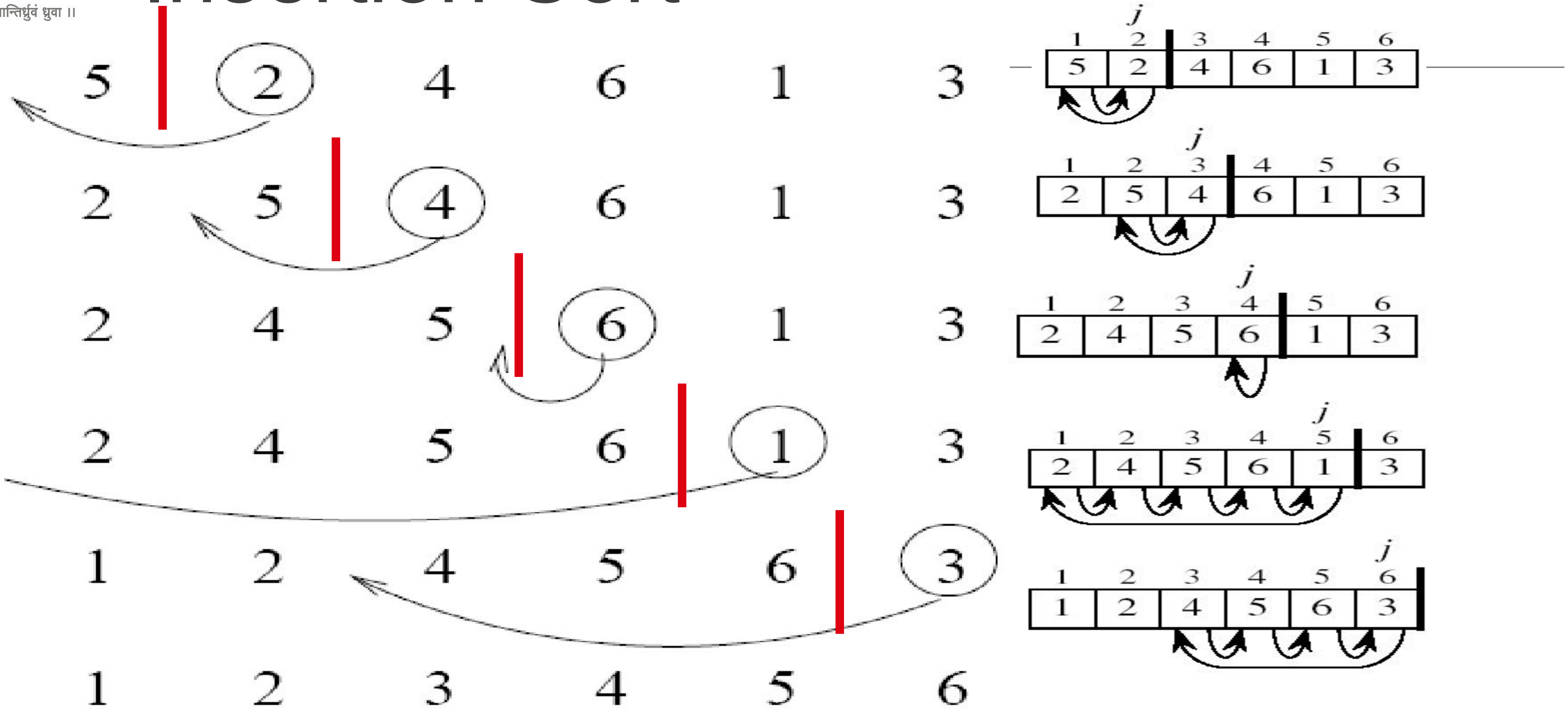
at each iteration, the array is divided in two sub-arrays:

left sub-array | right sub-array

2 5 | 4 6 1 3

sorted | unsorted

Insertion Sort



Insertion Sort

```
void insertionSort( arr[],  n)
{

  for (i = 1; i < n; i++)
  {
     key = arr[i];
     j = i-1;

/* Move elements of arr[0..i-1], that are greater than
key, to one position ahead of their current position */
      while (j >= 0 && arr[j] > key)
      {
         arr[j+1] = arr[j];
         j = j-1;
      }
      arr[j+1] = key;
   }
}
```

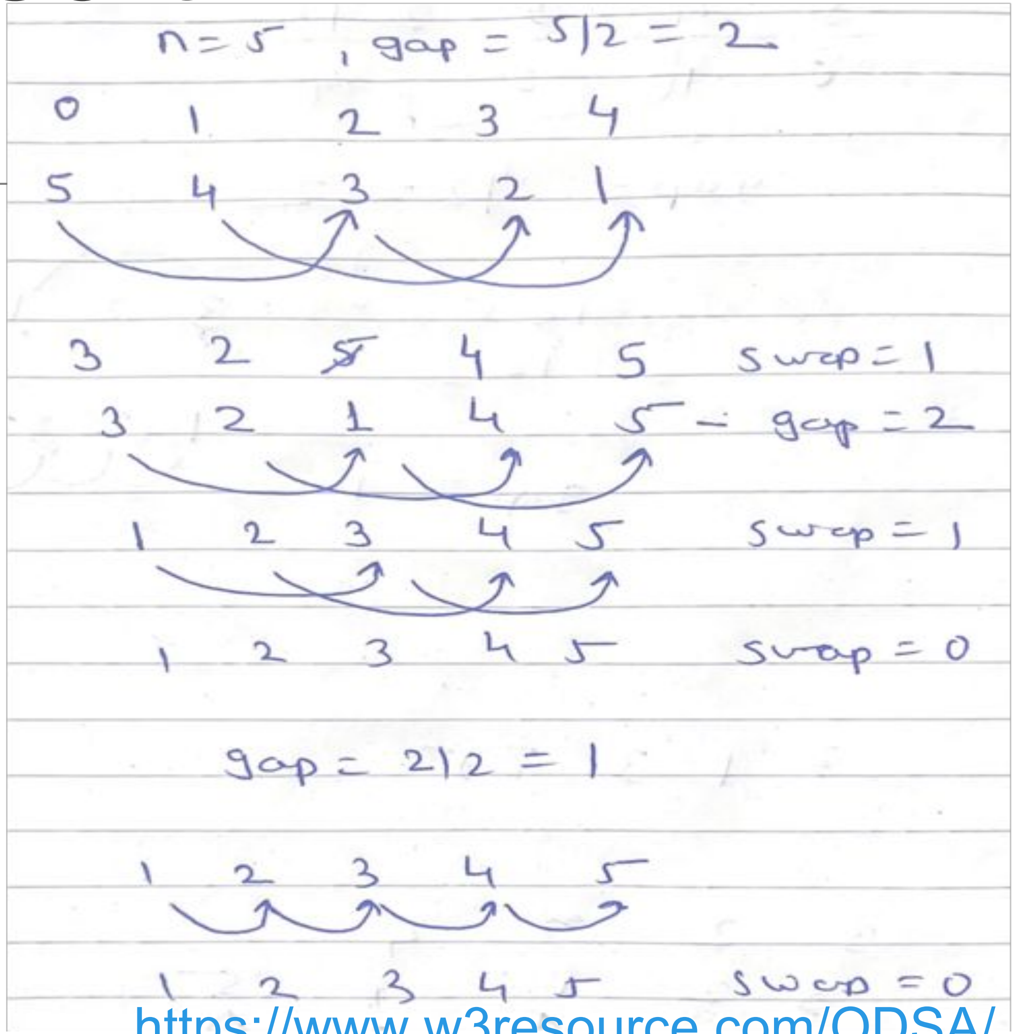
Analysis of Insertion Sort

If the data is initially sorted, only one comparison is made on each pass so that the sort time complexity is O(n)

The number of interchanges needed in both the methods is on the average $(n^2)/4$, and in the worst cases is about $(n^2)/2$

Shell sort

```
void shell_sort(int A[],int n]
{
    gap=n/2;
    do
    {
        do
        {
            swapped=0;
            for(i = 0; i < n- gap ; i++)
                if(A[ i ] > A[ i + gap ])
                {
                    swap();
                    swapped=1;
                }
        } while(swapped == 1);
    }while((gap=gap/2) >= 1);
}
```



https://www.w3resource.com/ODSA/AV/Sorting/shellsortAV.html

Shell sort- Complexity Analysis

1. Complexity in the **Best Case: O(n*Log n)**

   The total number of comparisons for each interval (or increment) is equal to the size of the array when it is already sorted.

1. Complexity in the **Average Case: O(n*log n)**

   It's somewhere around O. (n1.25)

1. Complexity in the **Worst-Case** Scenario: Less Than or Equal to **O (n^2)**

   Shell sort's worst-case complexity is always less than or equal to **O. (n^2)**

The degree of complexity is determined by the interval picked. The above complexity varies depending on the increment sequences used. The best increment sequence has yet to be discovered.

General Concept of Divide & Conquer

Given a function to compute on *n* inputs, the divide-and-conquer strategy consists of:

- splitting the inputs into k distinct subsets, $1<k\leq n$, yielding k subproblems.
- solving these subproblems
- combining the subsolutions into solution of the whole.
- if the subproblems are relatively large, then divide_Conquer is applied again.
- if the subproblems are small, the are solved without splitting.

Control Abstraction for Divide and Conquer

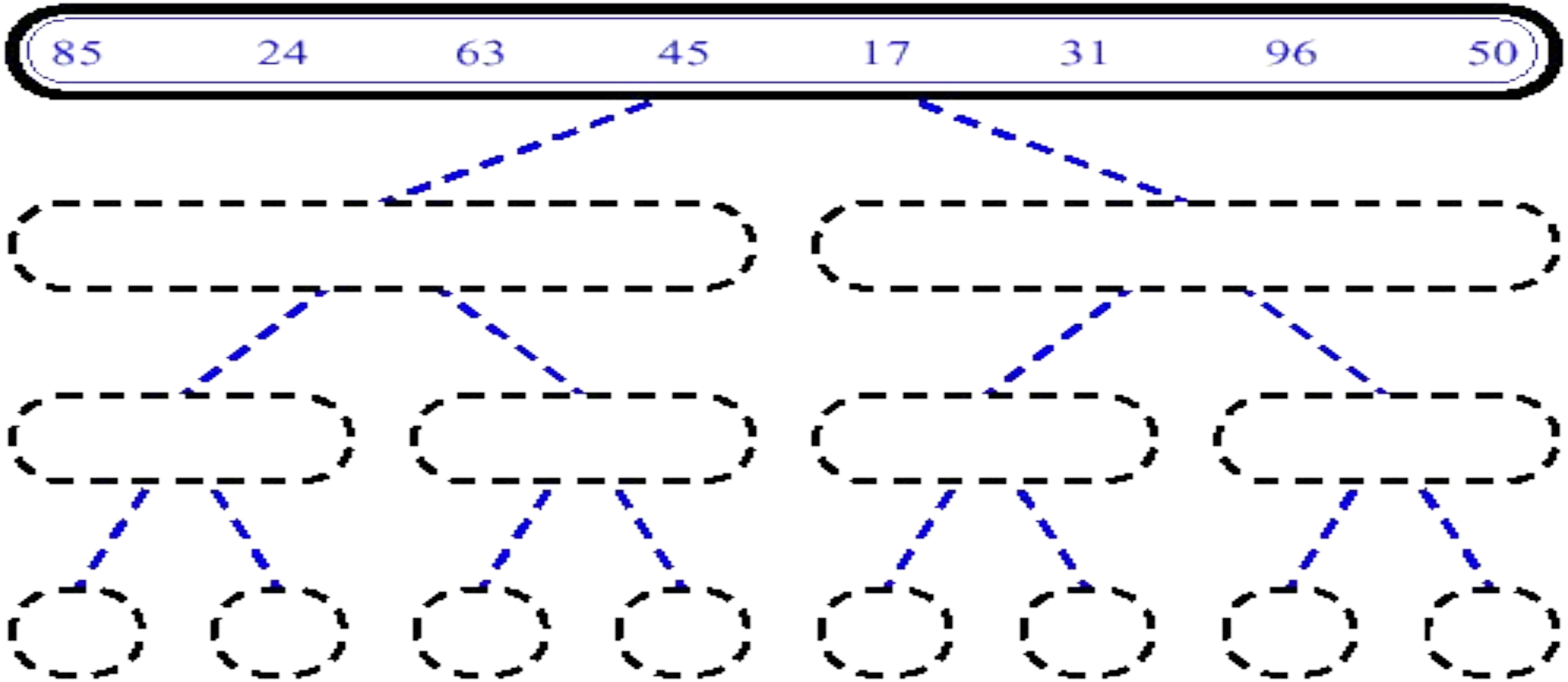
```
Divide_Conquer(problem P)
{
   if Small(P) return S(P);
   else {

     divide P into smaller instances P1, P2, ..., Pk, k≥1;

     Apply Divide_Conquer to each of these subproblems;

     return Combine(Divide_Conque(P1), Divide_Conque(P2),...,
   Divide_Conque(Pk));
   }
}
```

Three Steps of The Divide and Conquer Approach

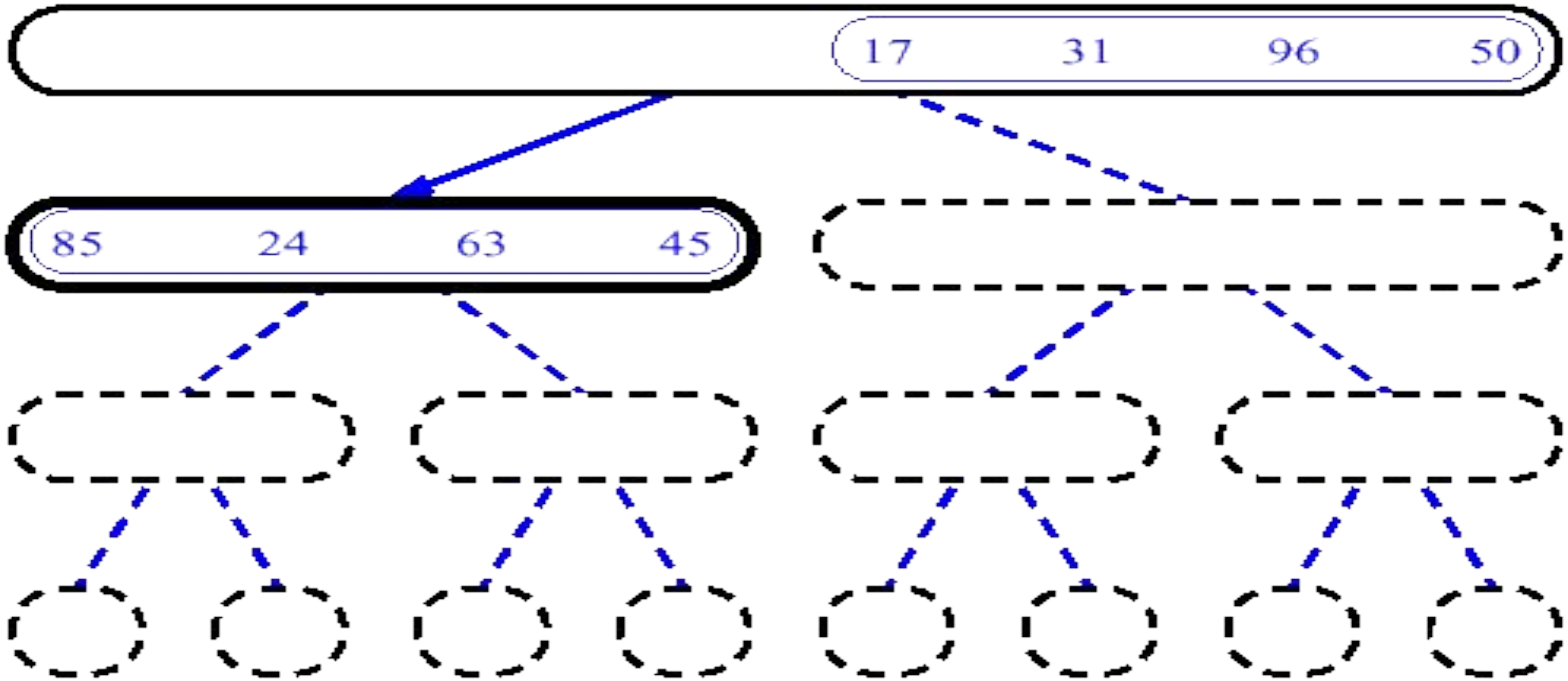
The most well known algorithm design strategy:

1. **Divide** the problem into two or more smaller subproblems.
2. **Conquer** the subproblems by solving them recursively.
3. **Combine** the solutions to the subproblems into the solutions for the original problem.

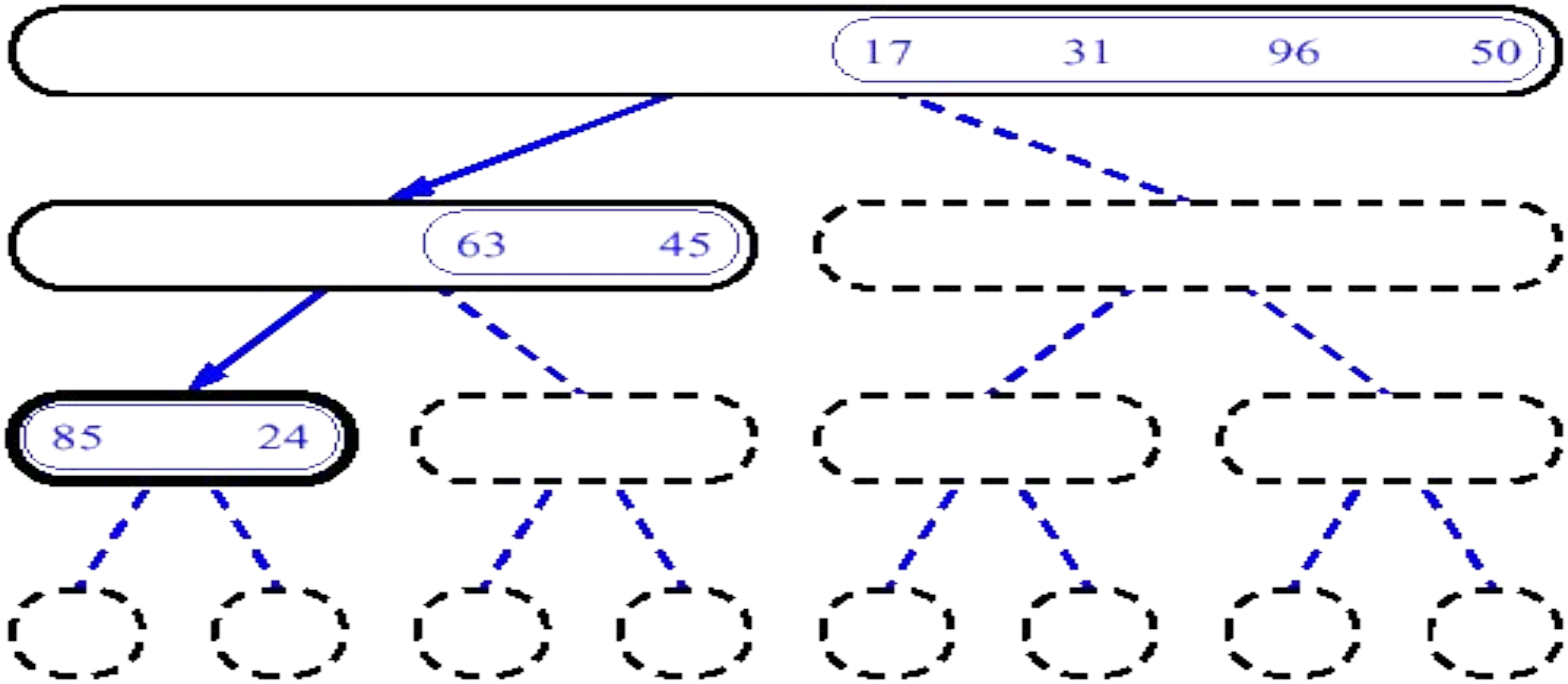
MergeSort (Example) - 1



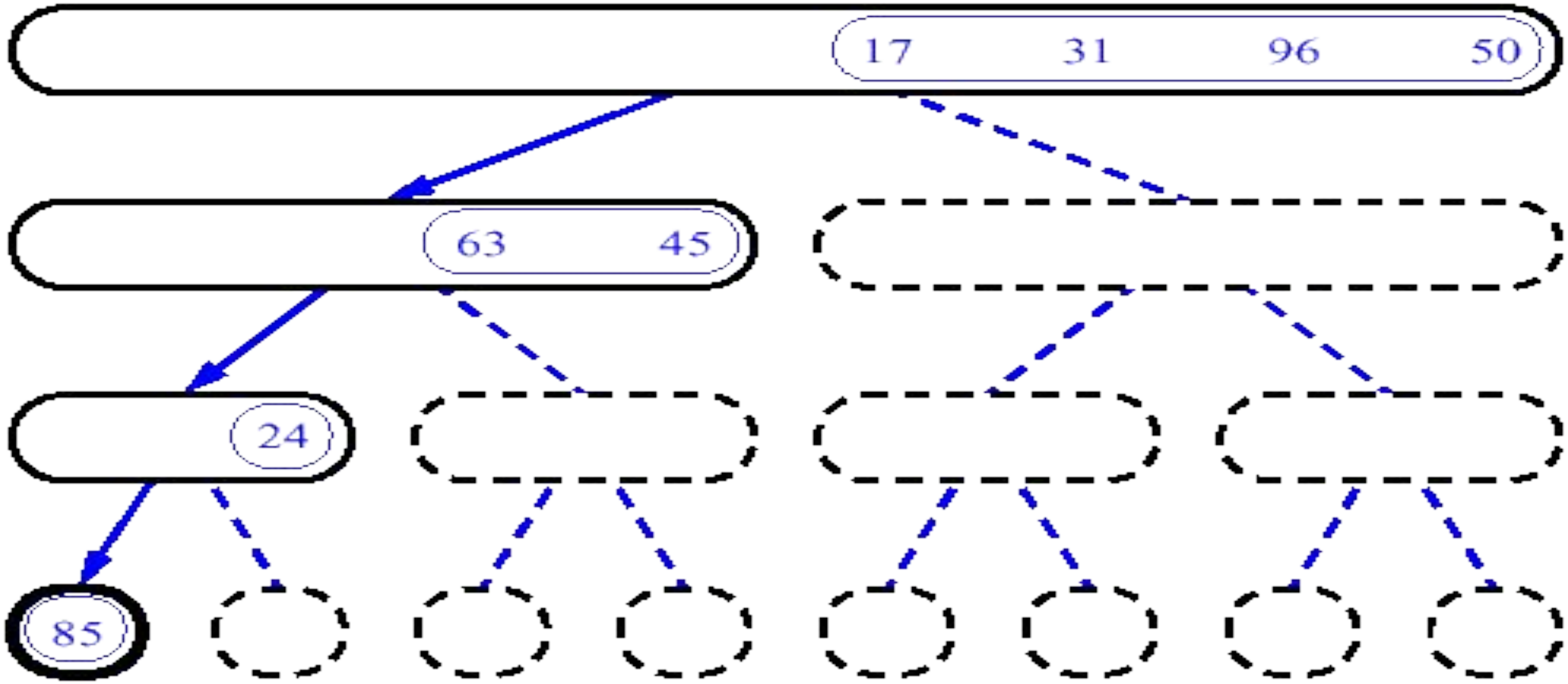
MergeSort (Example) - 2



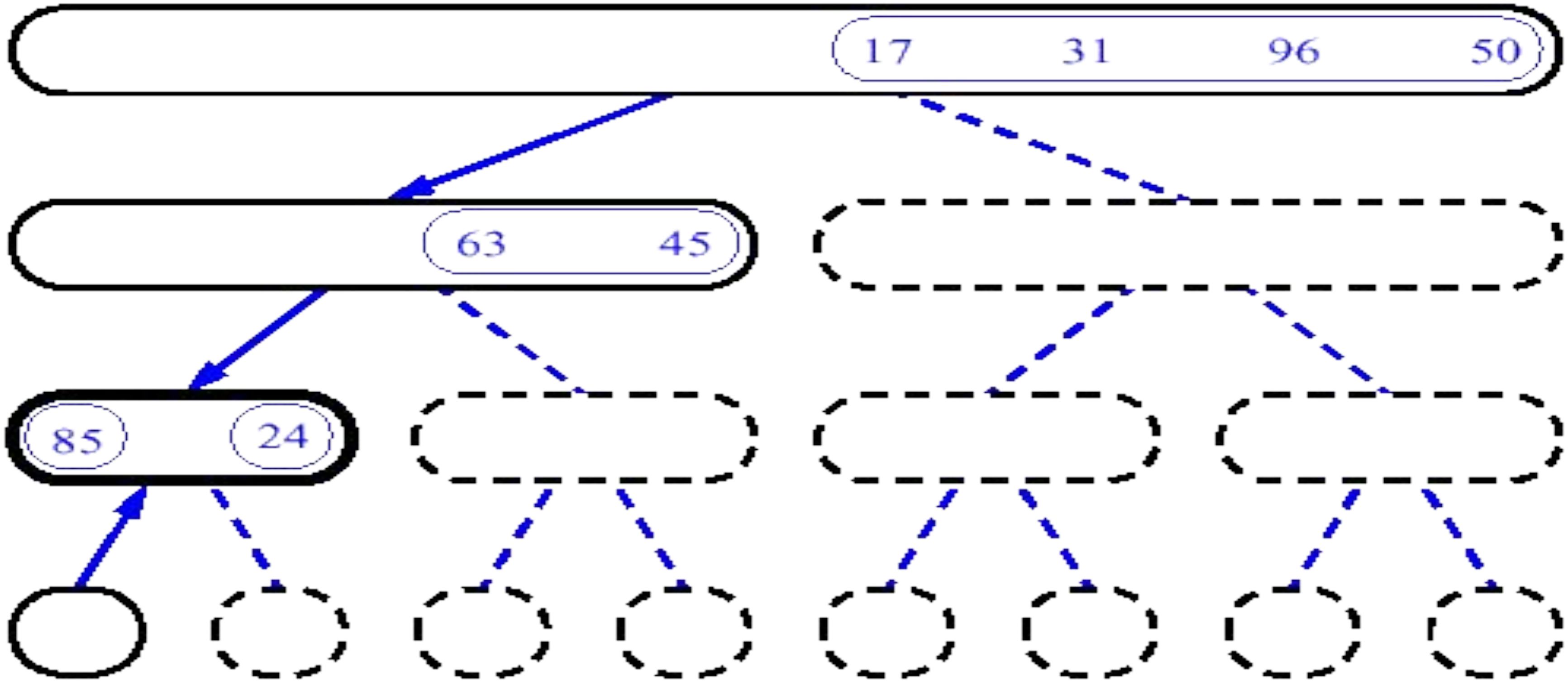
MergeSort (Example) - 3



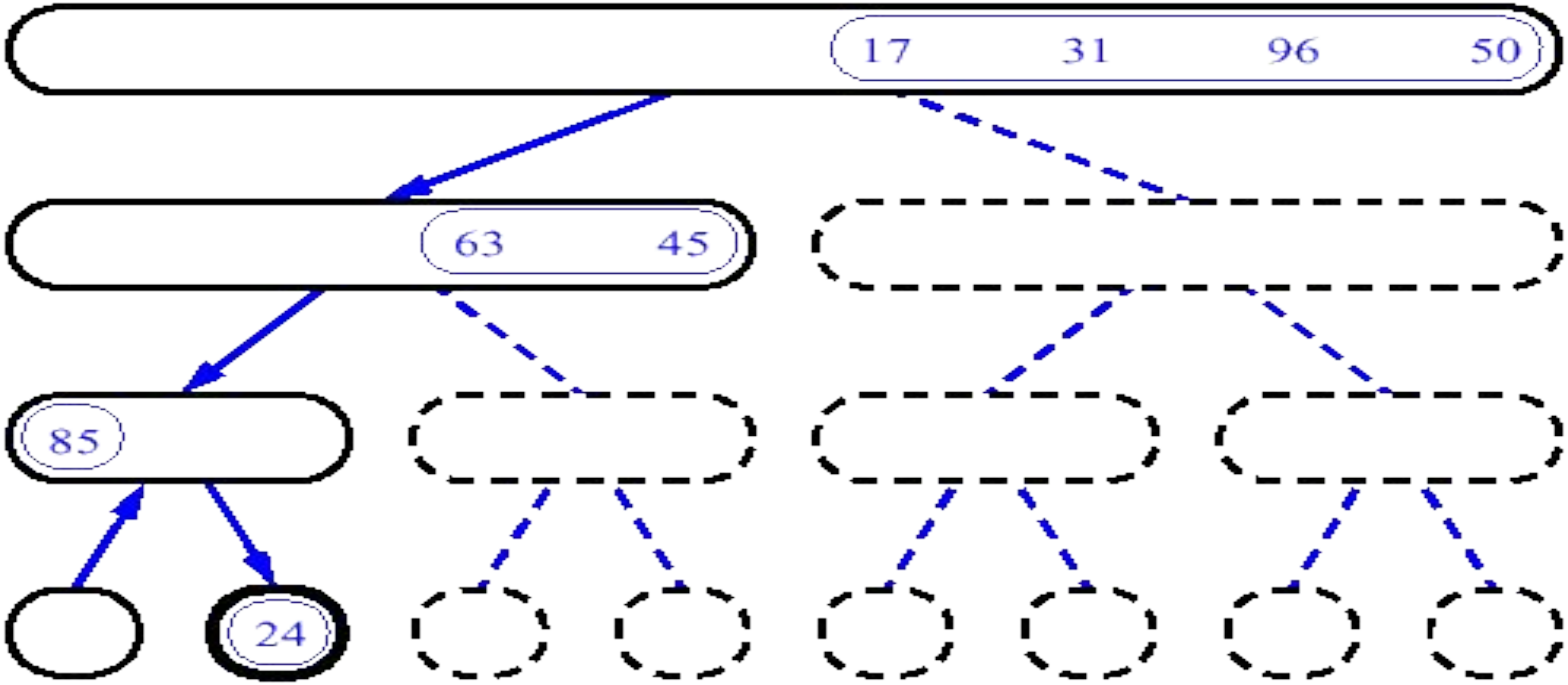
MergeSort (Example) - 4



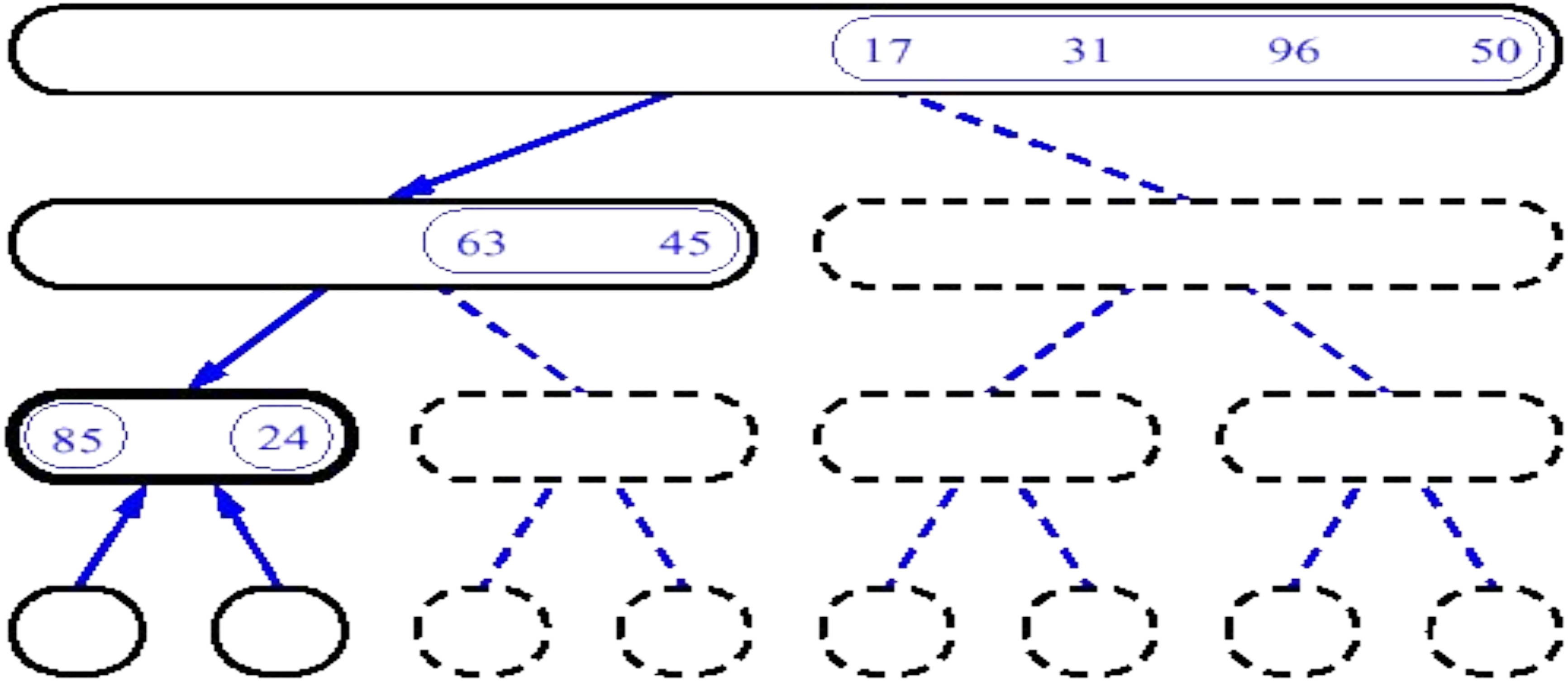
MergeSort (Example) - 5



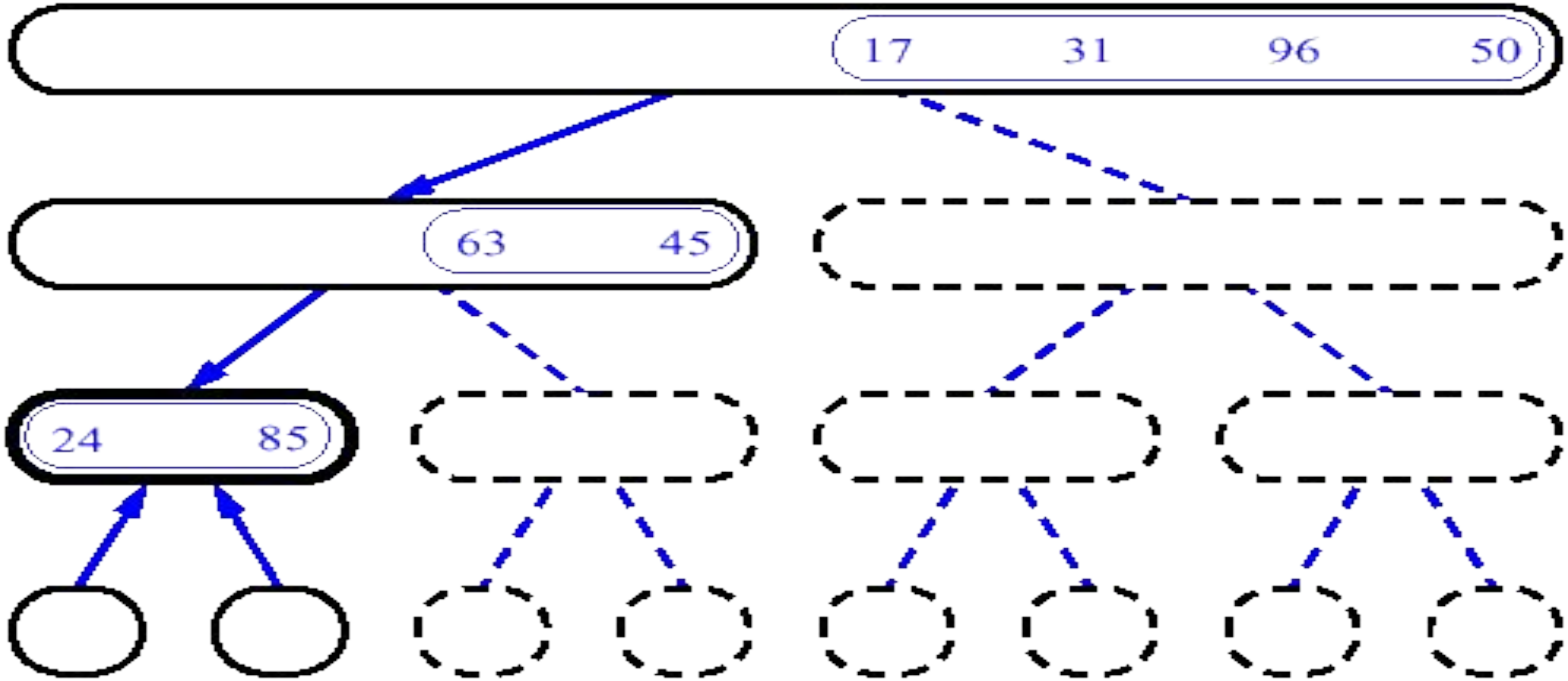
MergeSort (Example) - 6



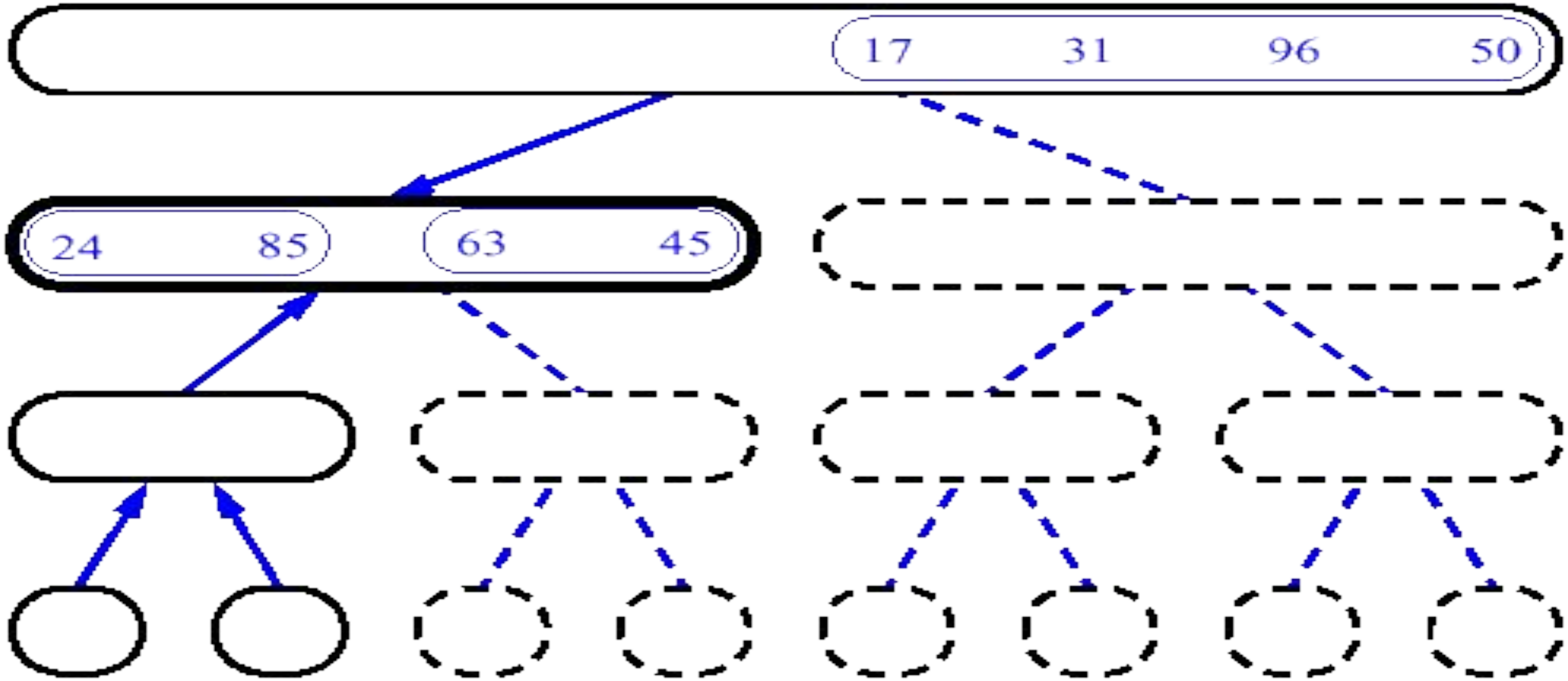
MergeSort (Example) - 7



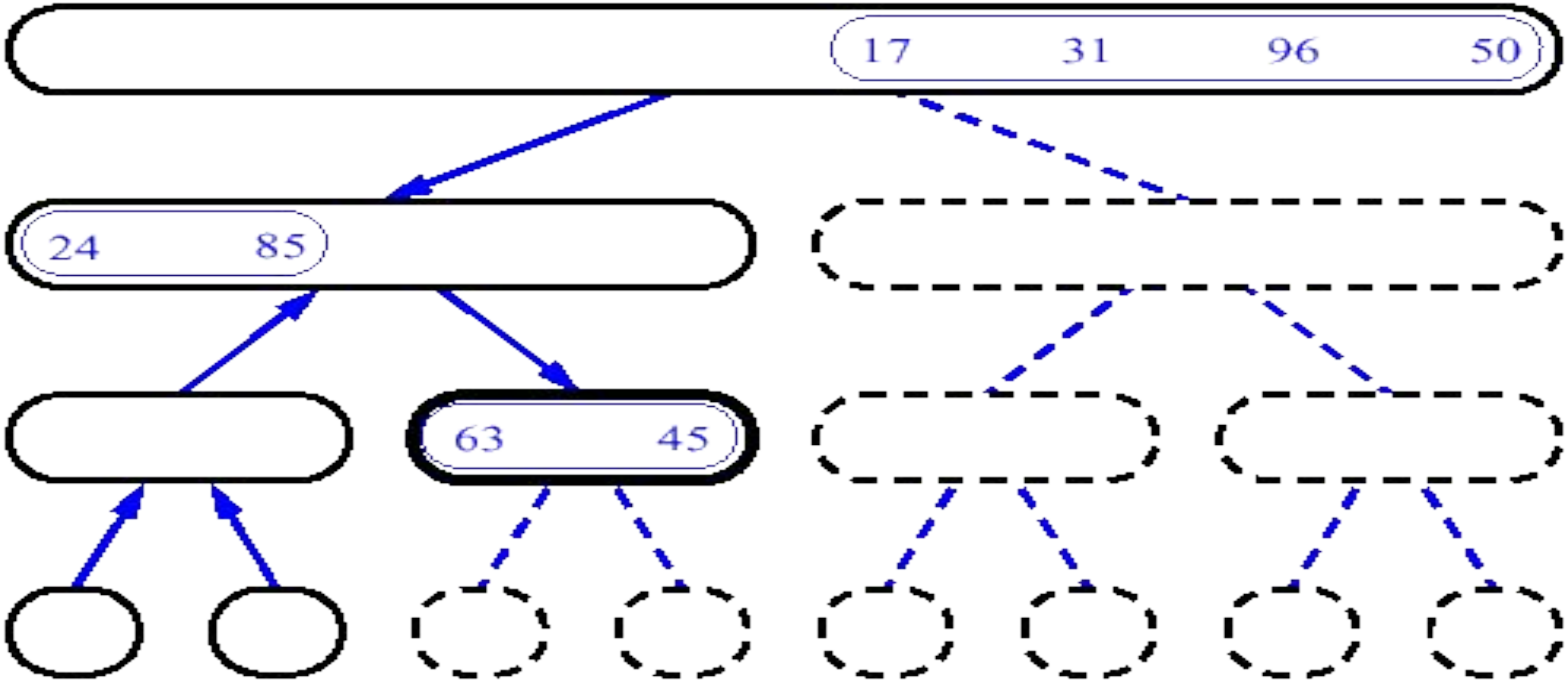
MergeSort (Example) - 8



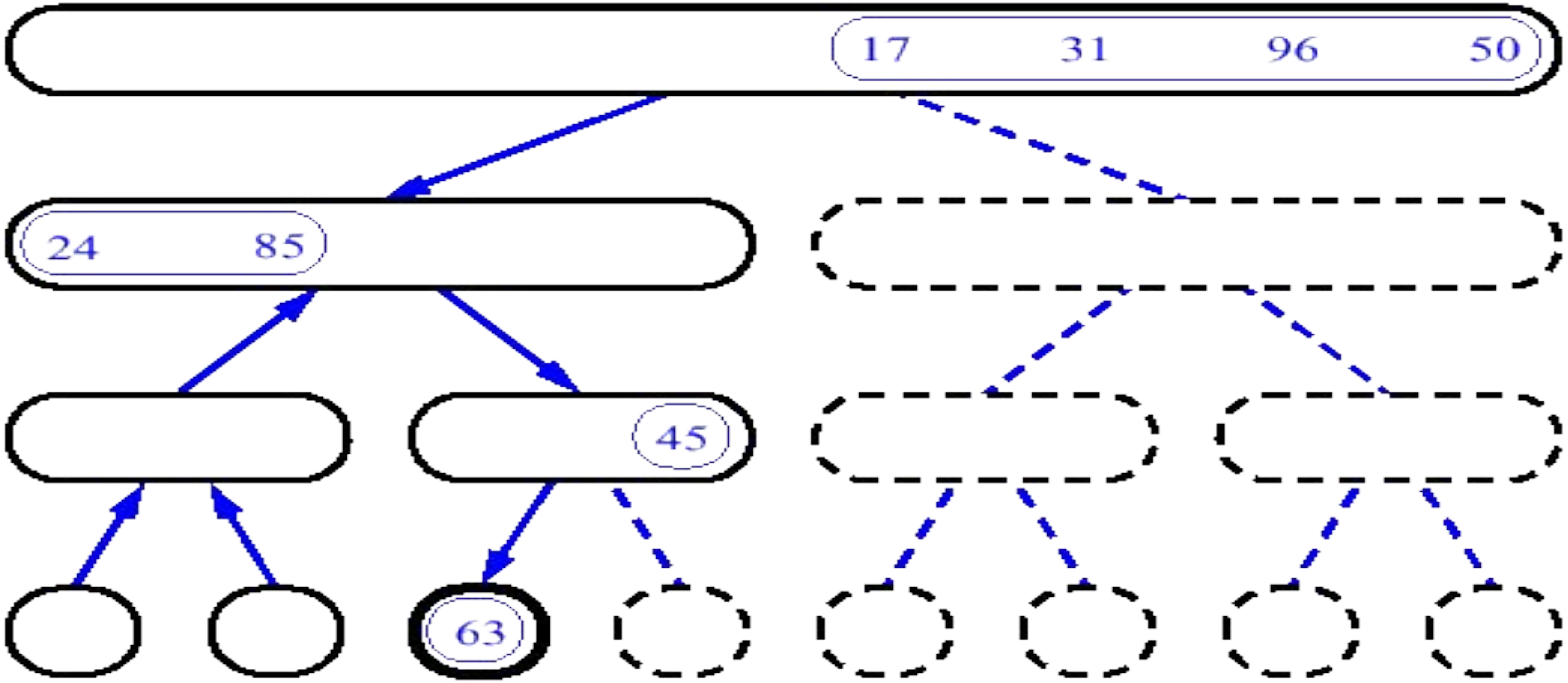
MergeSort (Example) - 9



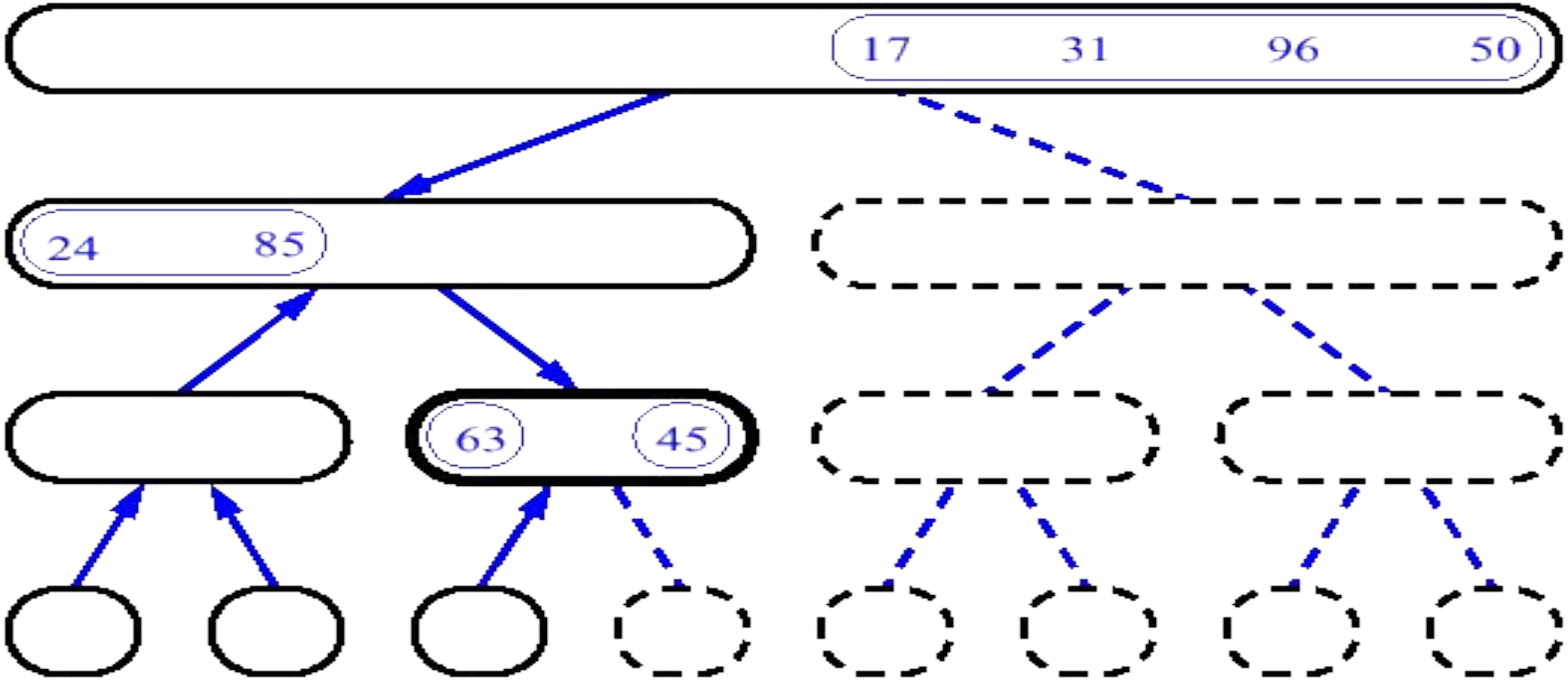
MergeSort (Example) - 10



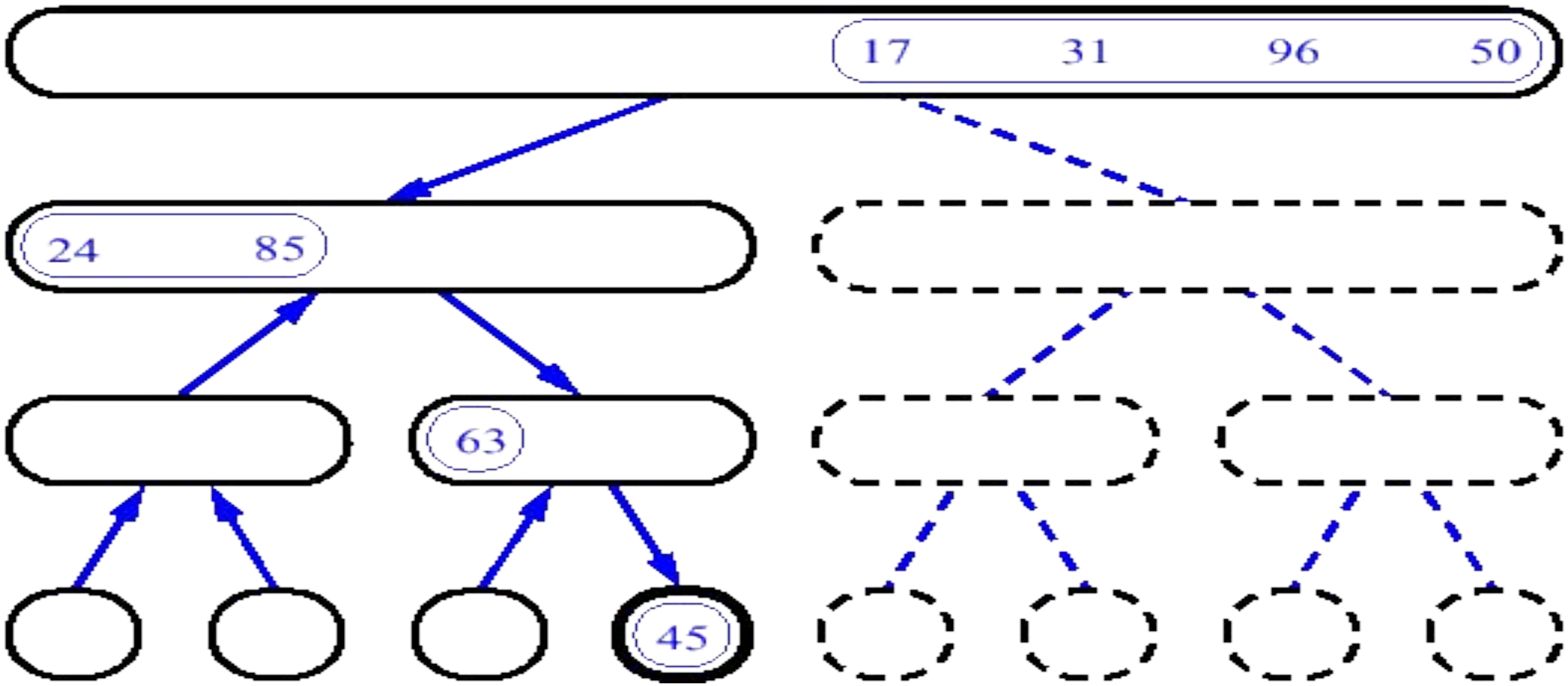
MergeSort (Example) - 11



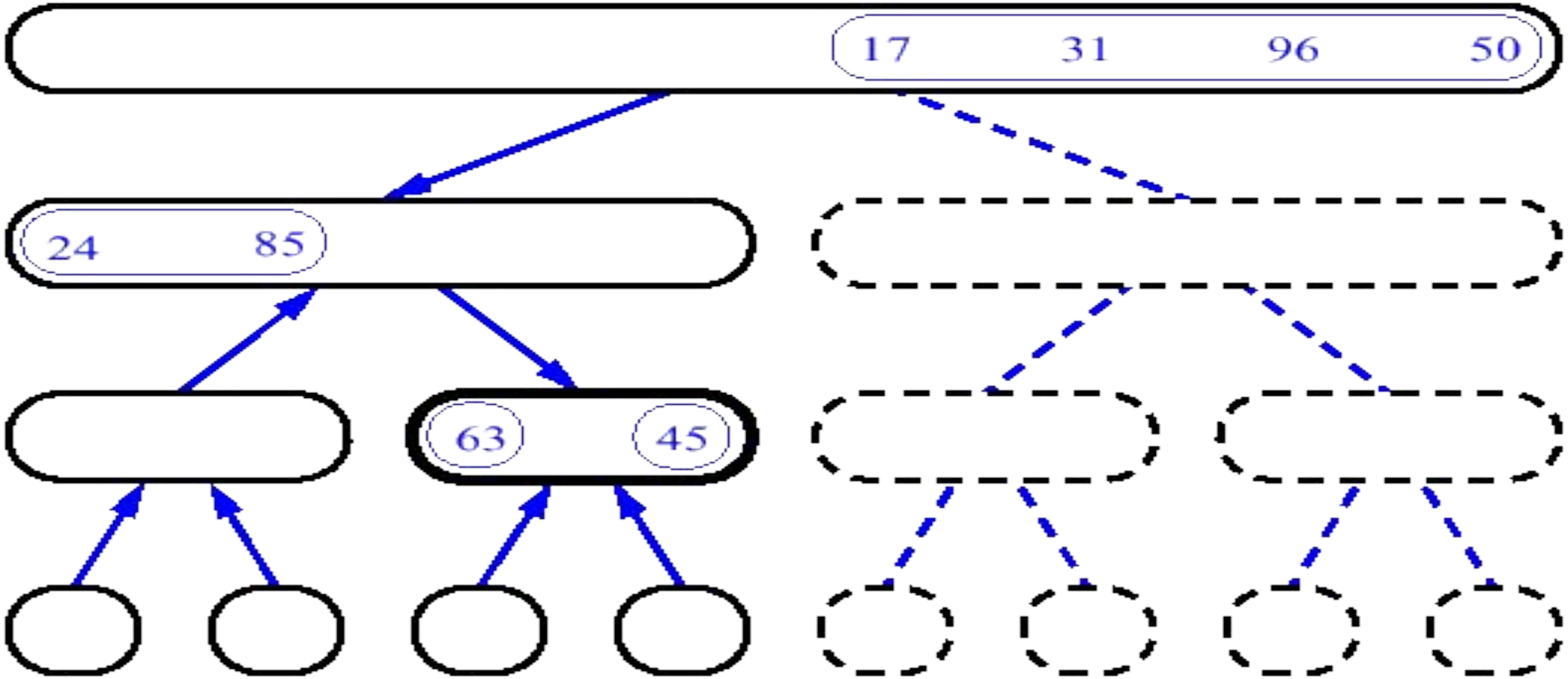
MergeSort (Example) - 12



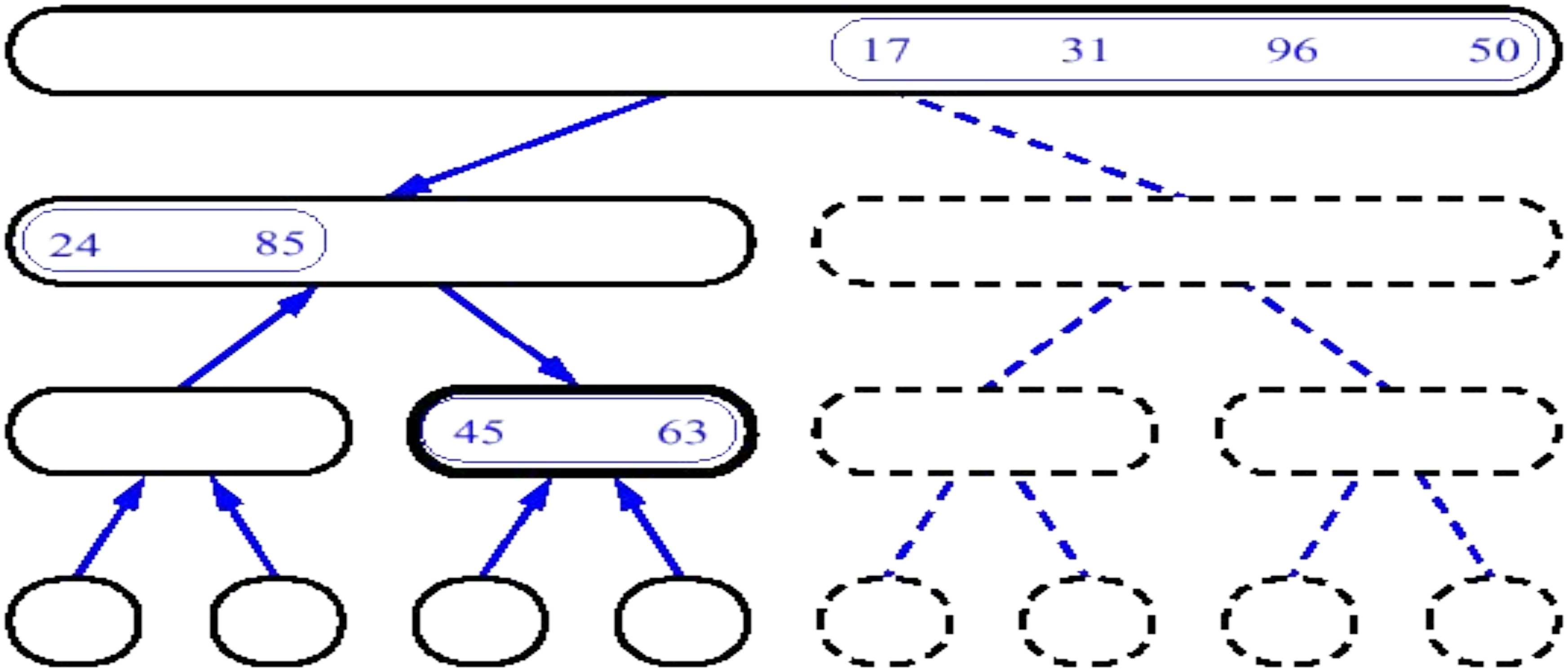
MergeSort (Example) - 13



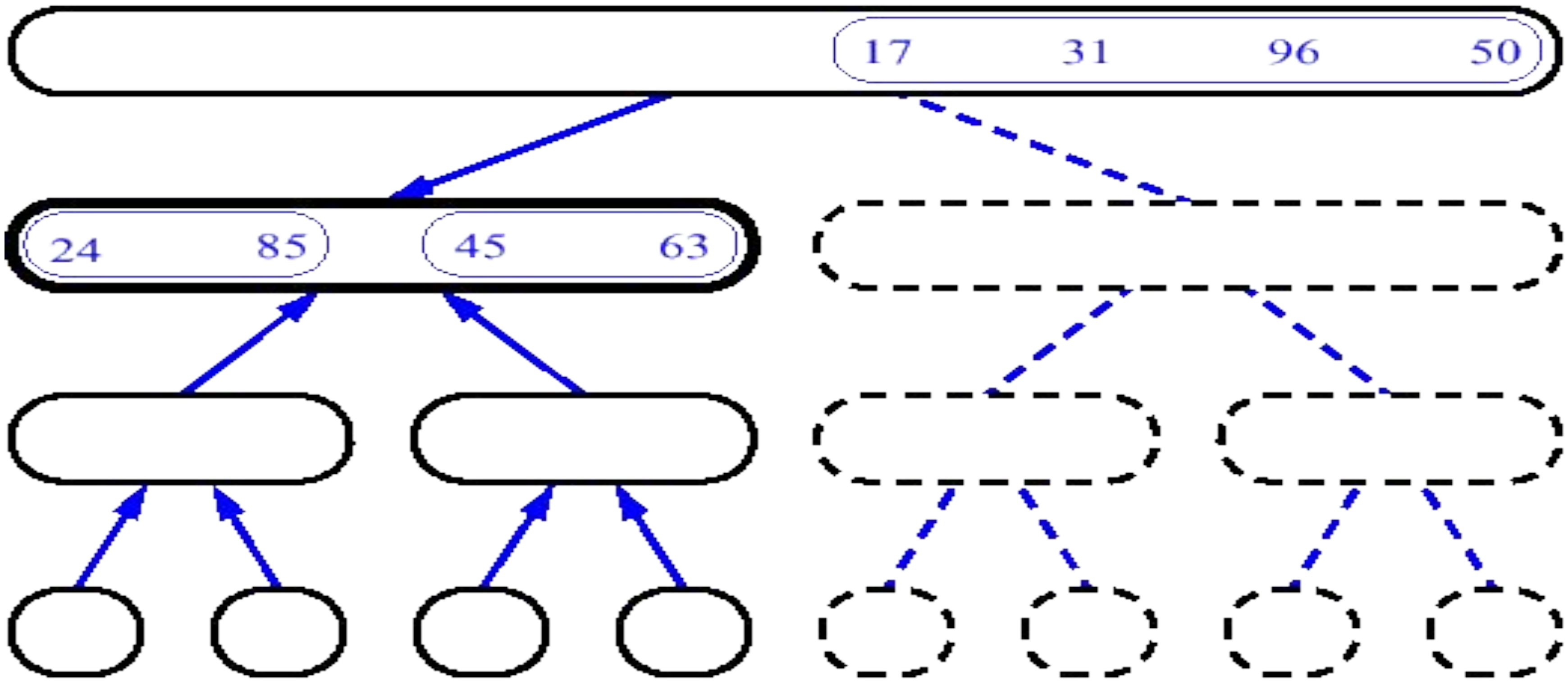
MergeSort (Example) - 14



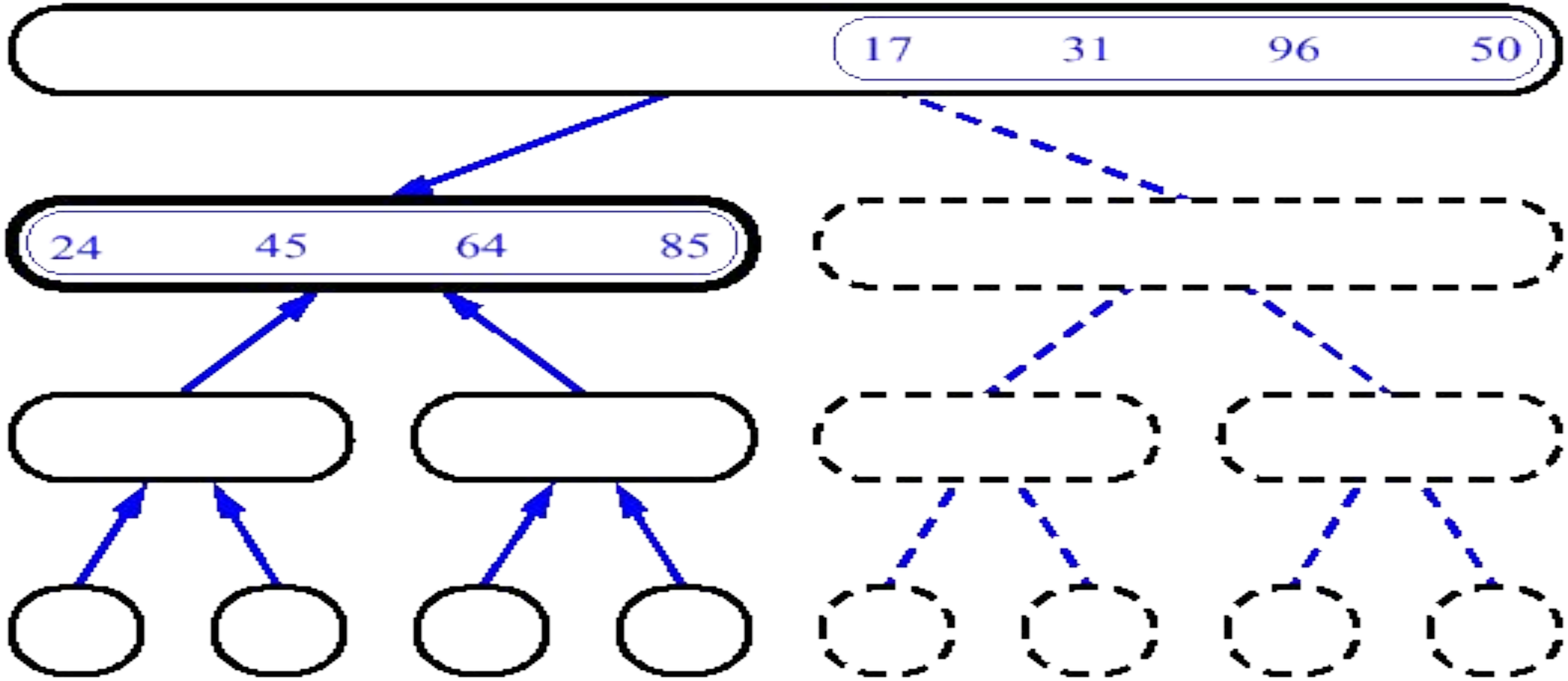
MergeSort (Example) - 15



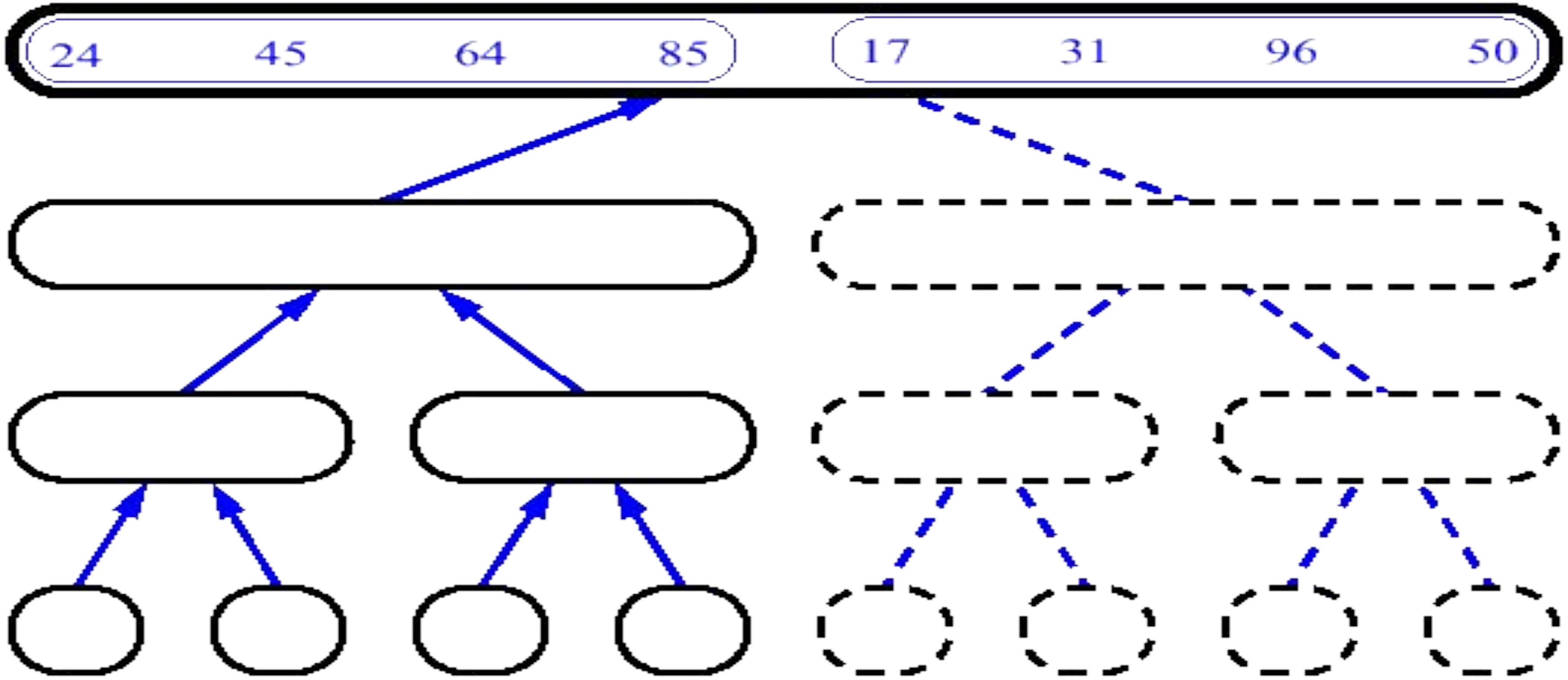
MergeSort (Example) - 16



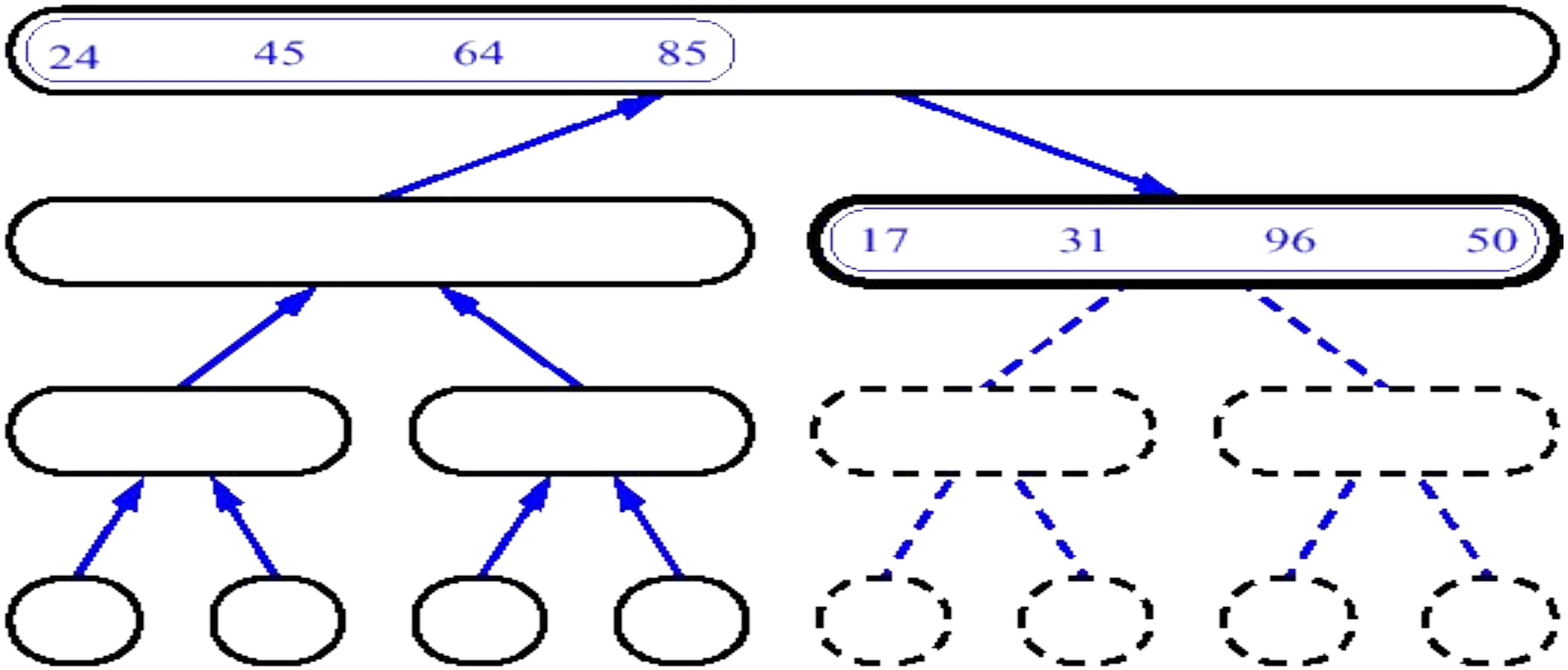
MergeSort (Example) - 17



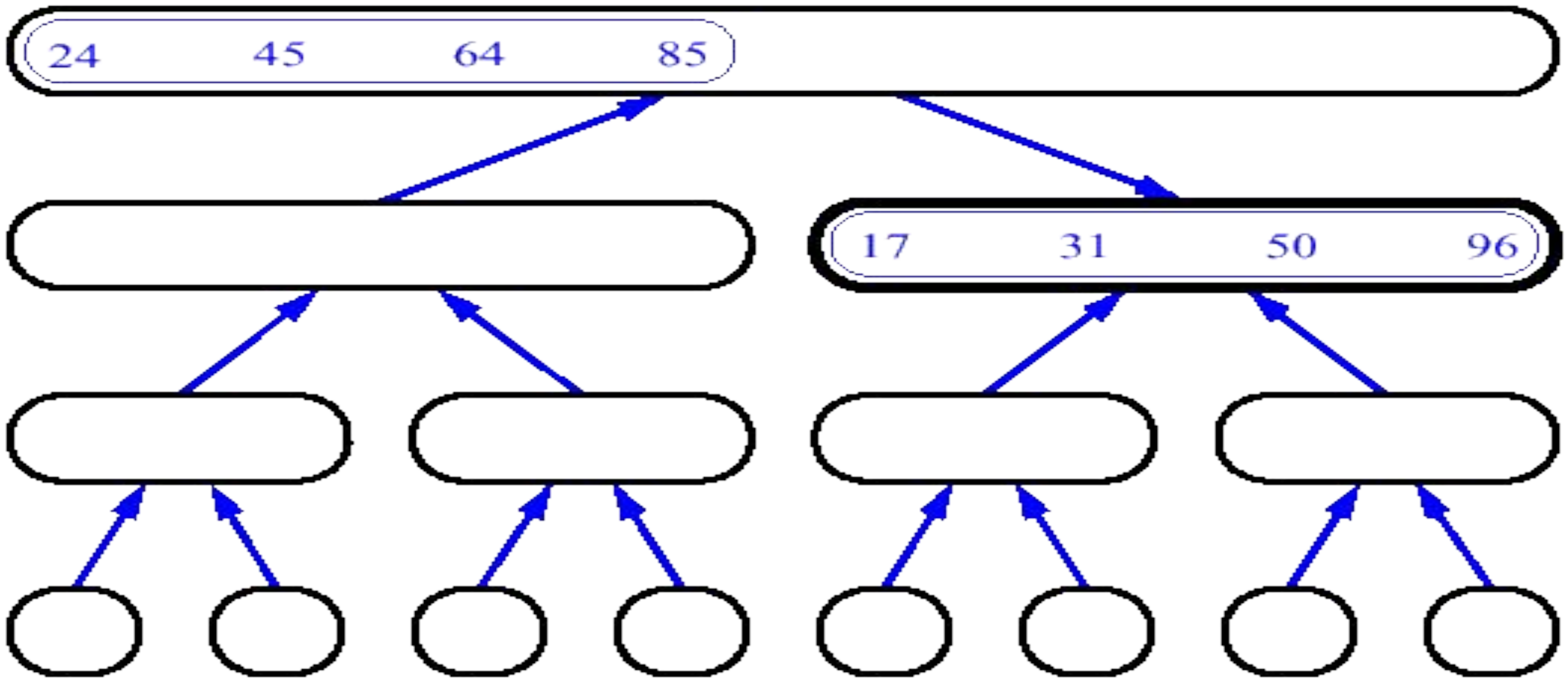
MergeSort (Example) - 18



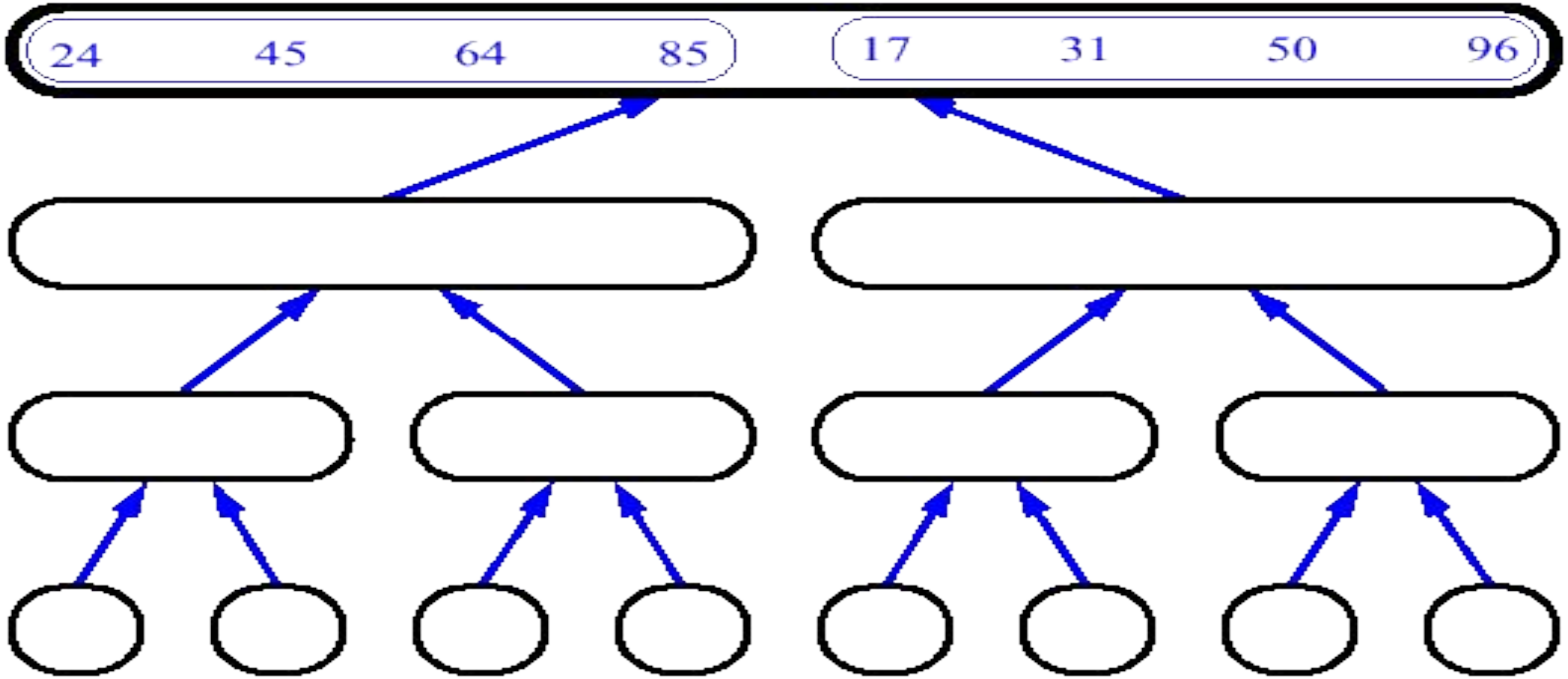
MergeSort (Example) - 19



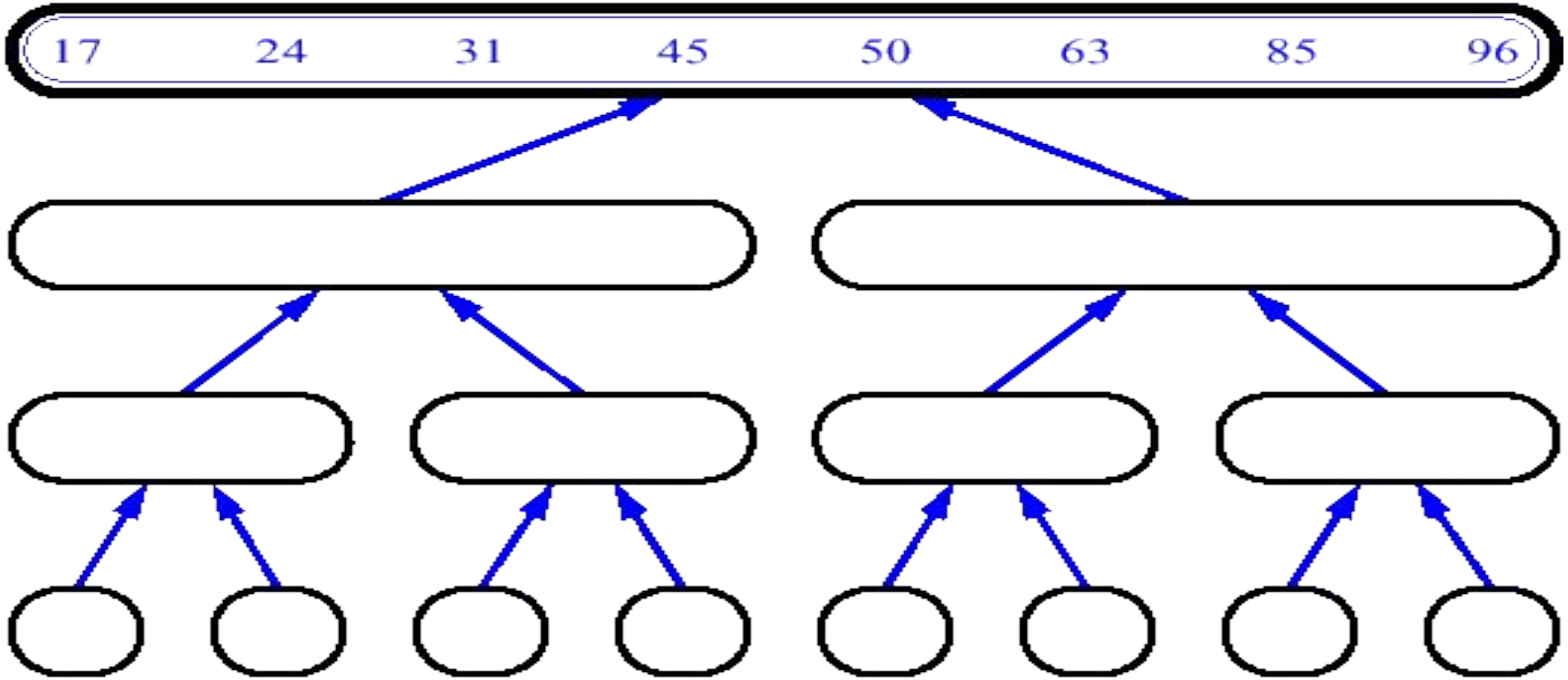
MergeSort (Example) - 20



MergeSort (Example) - 21

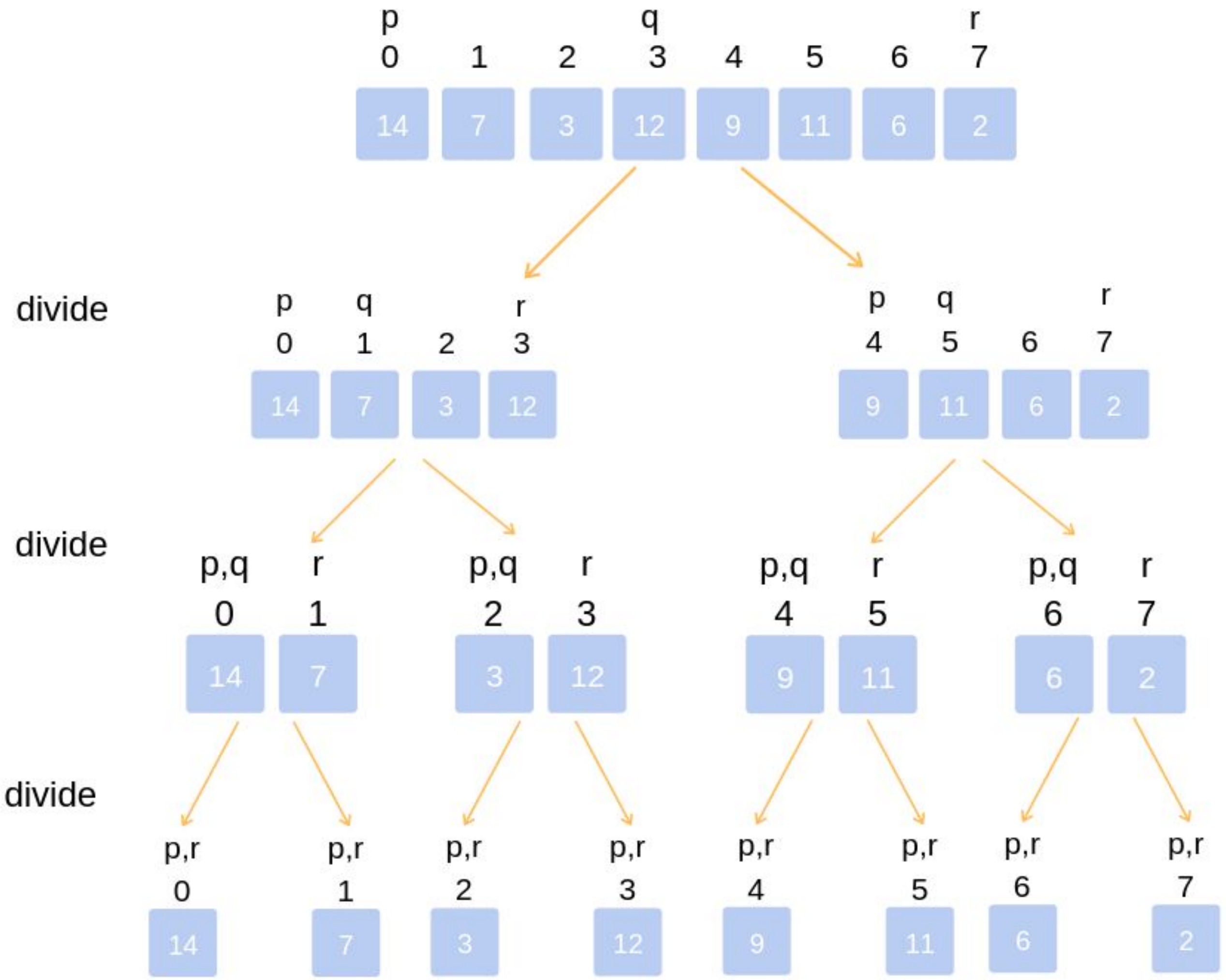


MergeSort (Example) - 22

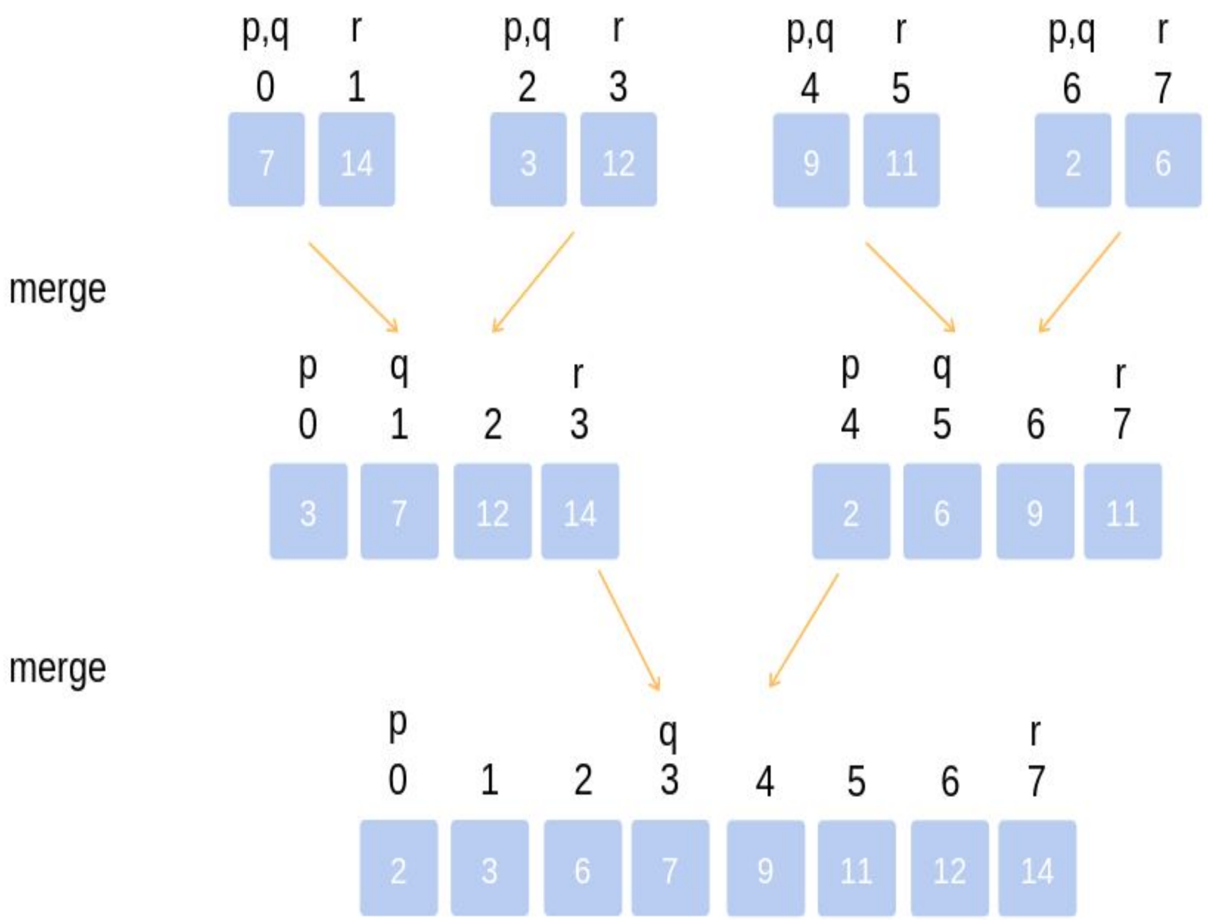


Merge Sort Algorithm

Algorithm MergeSort($low, high$)
// $a[low : high]$ is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
  if ($low < high$) **then** // If there are more than o
  {
    // Divide P into subproblems.
      // Find where to split the set.
        $mid := \lfloor (low + high)/2 \rfloor$;
    // Solve the subproblems.
      MergeSort(low, mid);
      MergeSort($mid + 1, high$);
    // Combine the solutions.
      Merge($low, mid, high$);
  }
}



```
Algorithm Merge(low, mid, high)
// a[low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid + 1 : high]. The go
// is to merge these two sets into a single set residing
// in a[low : high]. b[ ] is an auxiliary global array.
{
    h := low; i := low; j := mid + 1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h];h := h + 1;
        }
        else
        {
            b[i] := a[j]; j := j + 1;
        }
        i := i + 1;
    }
    if (h > mid) then
        for k := j to high do
        {
            b[i] := a[k]; i := i + 1;
        }
    else
        for k := h to mid do
        {
            b[i] := a[k]; i := i + 1;
        }
    for k := low to high do a[k] := b[k];
}
```



Analysis of Merge Sort

Worst Case:O(***nlogn***)

Averagevcade :O(***n logn***)