Divide & Conquer Approach (Part-I)

Md. Tanvir Rahman
Lecturer, Dept. of ICT
Mawlana Bhashani Science and Technology University

Divide and Conquer

- Divide the problem into a number of subproblem.
- Conquer (solve) the sub-problems **recursively**
- Combine (merge) solutions to subproblems into a solution to the original problem

If you want to learn more, watch this: https://www.youtube.com/watch?v=2Rr2tW9zvRg

Divide and Conquer Algorithms

- Example
  - Binary Search
  - Merge Sort
  - Quick Sort (Will discuss in Part-II)
  - Heap Sort (Will discuss in Part-II)
  - A lot more! Find yourself! (Will discuss in Part-II)

Binary Search

Binary Search

- It can be implemented on sorted lists only
- It is also known as Half-interval search as it eliminates one half of the elements after each comparison
- The only disadvantage of it is that it only works on a sorted list

Process

- Eliminates one half of the elements after each comparison.
- Locates the middle of the array
- Compares the value at that location with the search key.
- If they are equal - done!
- Otherwise, decides which half of the array contains the search key.
- Repeat the search on that half of the array and ignore the other half.
- The search continues until the key is matched or no elements remain to be searched.

code of Binary Search

```
int array_size;
int a[arraySize]; //sorted array
int key , low = 0, middle, high = (array_size - 1);
while (low <= high)
{
    middle = (low + high) / 2;
    if (key == a[middle])
    {
        print “Found element“;
        break;
    }
    else if (key < a [middle])
        high = middle -1;      // search low end of array
    else
        low = middle + 1;      // search high end of array
}
```

For Lab: You can find the solution using (i. Iterative approach & ii. Recursion)
https://www.tutorialspoint.com/binary-search-recursive-and-iterative-in-c-program

Pseudocode of BINARY-SEARCH

```
BINARY-SEARCH(x, T, p, r)
low = p
high = max(p, r + 1)
while low < high
    mid = ⌊(low + high)/2⌋
    if x ≤ T[mid]
        high = mid
    else low = mid + 1
return high
```

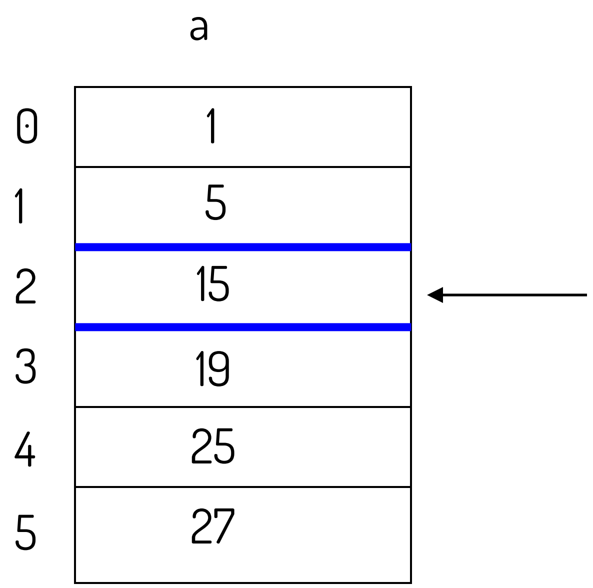
Binary Search Example

	a
0	1
1	5
2	15
3	19
4	25
5	27
6	29
7	31
8	33
9	45
10	55
11	88
12	100

search key = 19

- middle of the array
- compare a[6] and 19
- 19 is smaller than 29 so the next search will use the lower half of the array

Binary Search Pass 2

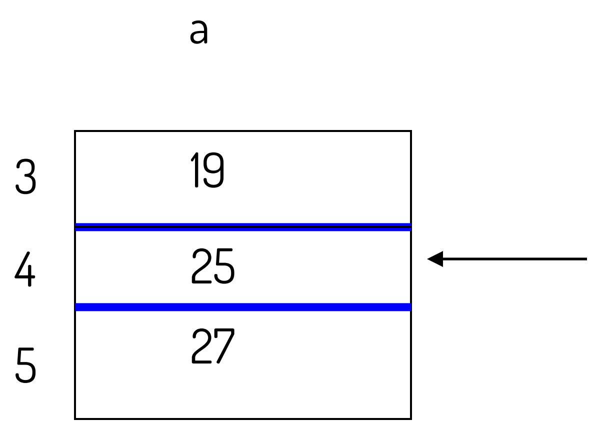


search key = 19

- use this as the middle of the array
- Compare a[2] with 19
- 15 is smaller than 19 so use the top half for the next pass

Binary Search Pass 3

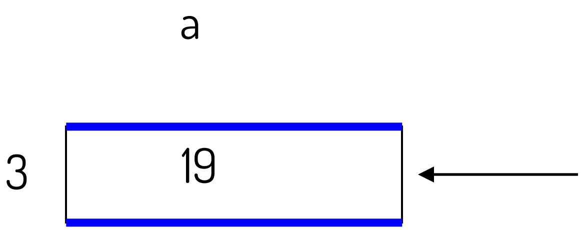
search key = 19



- Use this as the middle of the array
- Compare a[4] with 19
- 25 is bigger than 19 so use the bottom half

Binary Search Pass 4

search key = 19



- use this as the middle of the array
- Compare a[3] with 19
- Found!!

Binary Search Example-2

	a
0	1
1	5
2	15
3	19
4	25
5	27
6	29
7	31
8	33
9	45
10	55
11	88
12	100

search key = 18

- middle of the array
- compare a[6] and 18
- 18 is smaller than 29 so the next search will use the lower half of the array

Binary Search Pass 2

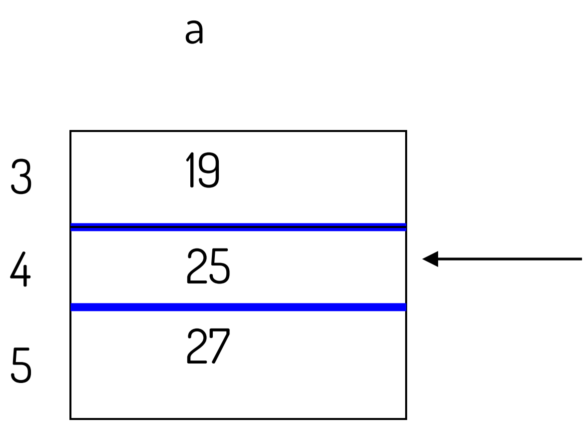
	a
0	1
1	5
2	15
3	19
4	25
5	27

search key = 18

- use this as the middle of the array
- Compare a[2] with 18
- 15 is smaller than 18 so use the top half for the next pass

Binary Search Pass 3

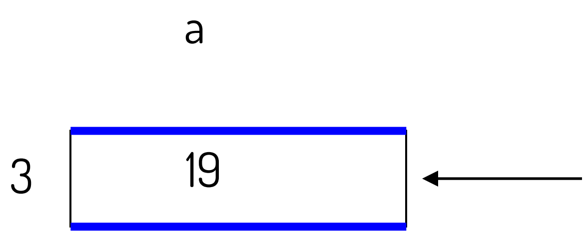
search key = 18



- Use this as the middle of the array
- Compare a[4] with 18
- 25 is bigger than 18 so use the lower half

Binary Search Pass 4

search key = 18



- use this as the middle of the array
- Compare a[3] with 18
- Does not match and no more elements to compare.
- Not Found!!

Binary Search (Example-3)

- 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.

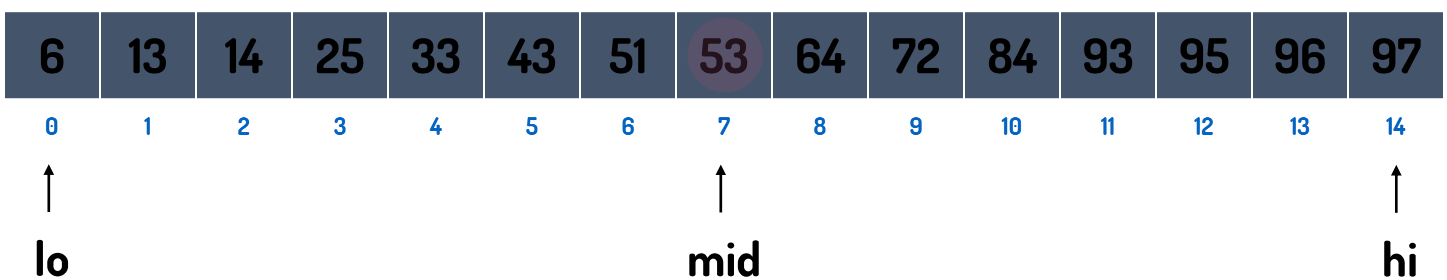
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo														↑ hi

Binary Search (Example-3)

- 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.

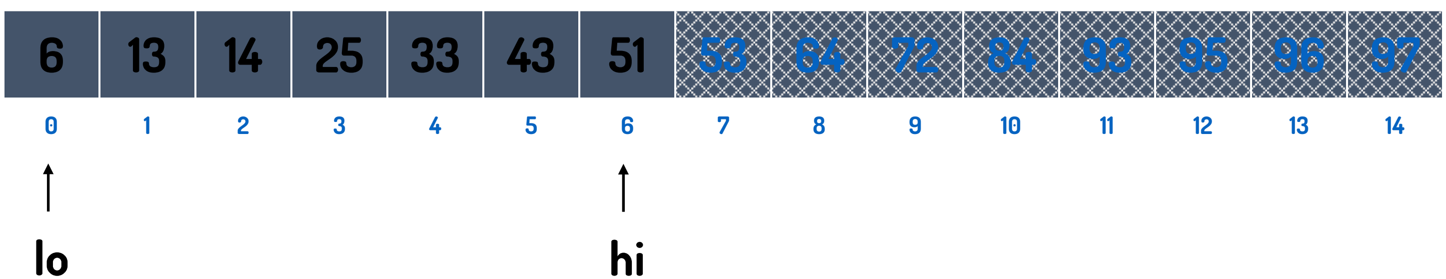


Binary Search (Example-3)

- 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.

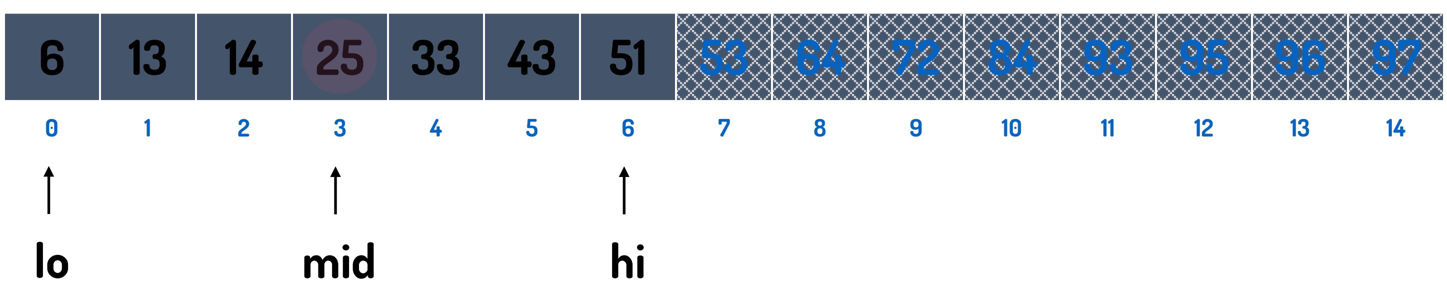


Binary Search (Example-3)

- 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.

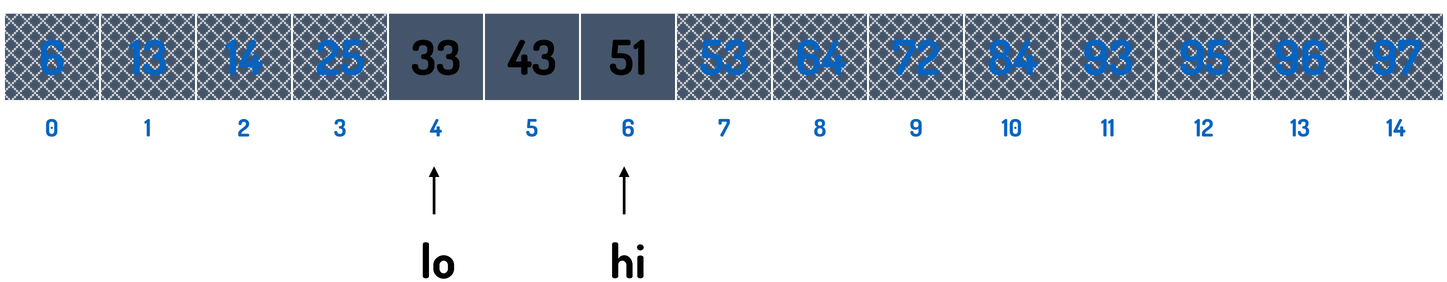


Binary Search (Example-3)

- 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.

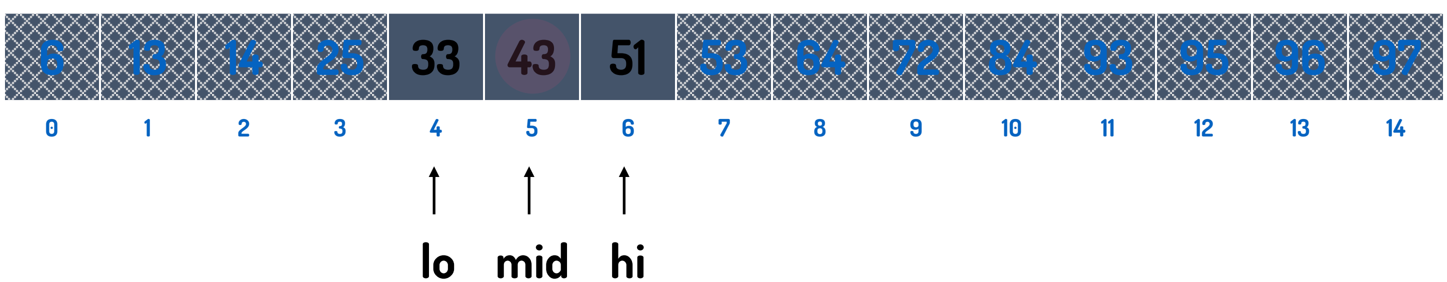


Binary Search (Example-3)

- 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 (Example-3)

- 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]`.

- Ex. Binary search for 33.

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

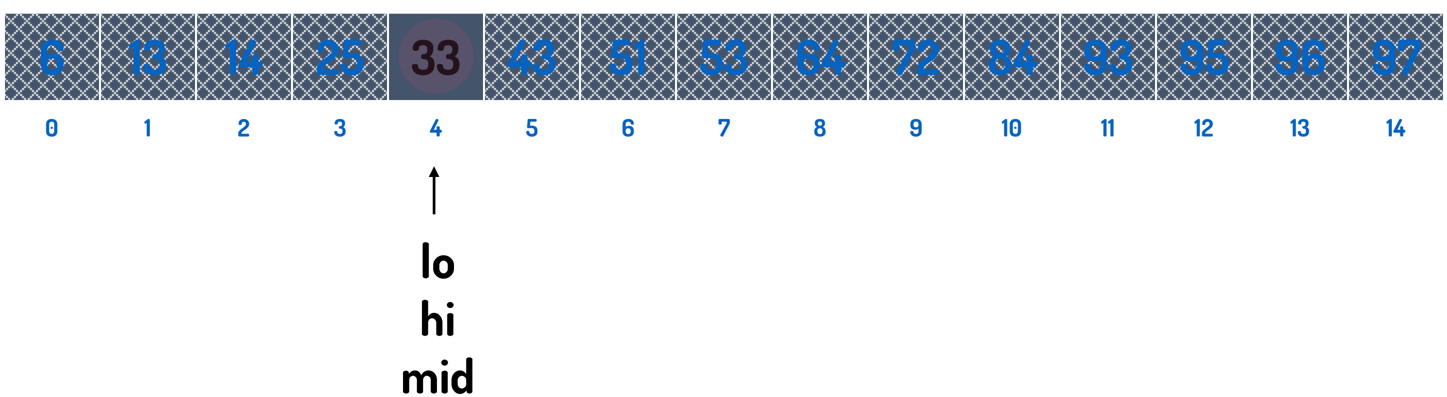
lo
hi

Binary Search (Example-3)

- 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.

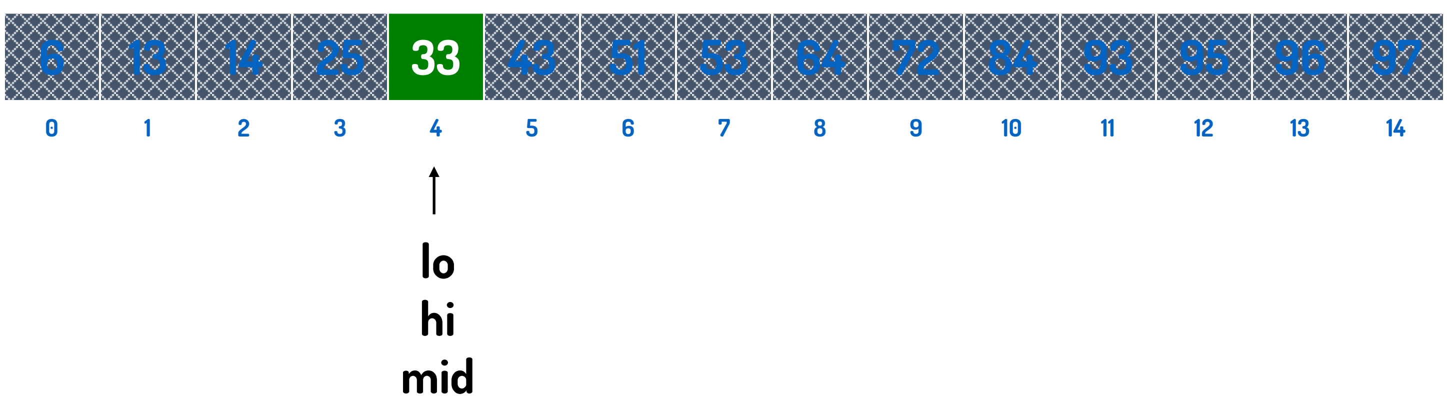


Binary Search (Example-3)

- 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.



Worst Case Complexity Analysis

Let us now carry out an Analysis of this method to determine its time complexity. Let us examine the operations for a specific case, where the number of elements in the array n is 64.

When n= 64, BinarySearch is called to reduce size to n=32

When n= 32, BinarySearch is called to reduce size to n=16

When n= 16, BinarySearch is called to reduce size to n=8

When n= 8, BinarySearch is called to reduce size to n=4

When n= 4, BinarySearch is called to reduce size to n=2

When n= 2, BinarySearch is called to reduce size to n=1

Contd.

Thus we see that BinarySearch function is called 6 times (6 elements of the array were examined) for n =64. [Note that 64 = 2ˆ6]

Let us consider a more general case where n is still a power of 2. Let us say n = 2ˆk .

Following the above argument for 64 elements, it is easily seen that after k searches, the while loop is executed k times and n reduces to size 1.

Let us assume that each run of the while loop involves at most 5 operations.

Thus total number of operations: 5k.

The value of k can be determined from the expression

$2^k = n$

Taking log of both sides

$k = \log n$

Contd.

Thus total number of operations = 5 log n.

We conclude from there that the time complexity of the Binary search method is $O(\log n)$, which is much more efficient than the Linear Search method.

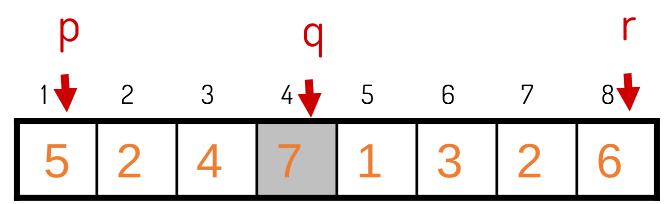
Reference 2 (For more clear understanding about the analysis): https://www.youtube.com/watch?v=TomQQb2kJvc

Merge Sort

Merge Sort Approach

- To sort an array $A[p\,..\,r]$:
- **Divide**
  - Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each
- **Conquer**
  - Sort the subsequences recursively using merge sort
  - When the size of the sequences is 1 there is nothing more to do
- **Combine**
  - Merge the two sorted subsequences

Merge Sort



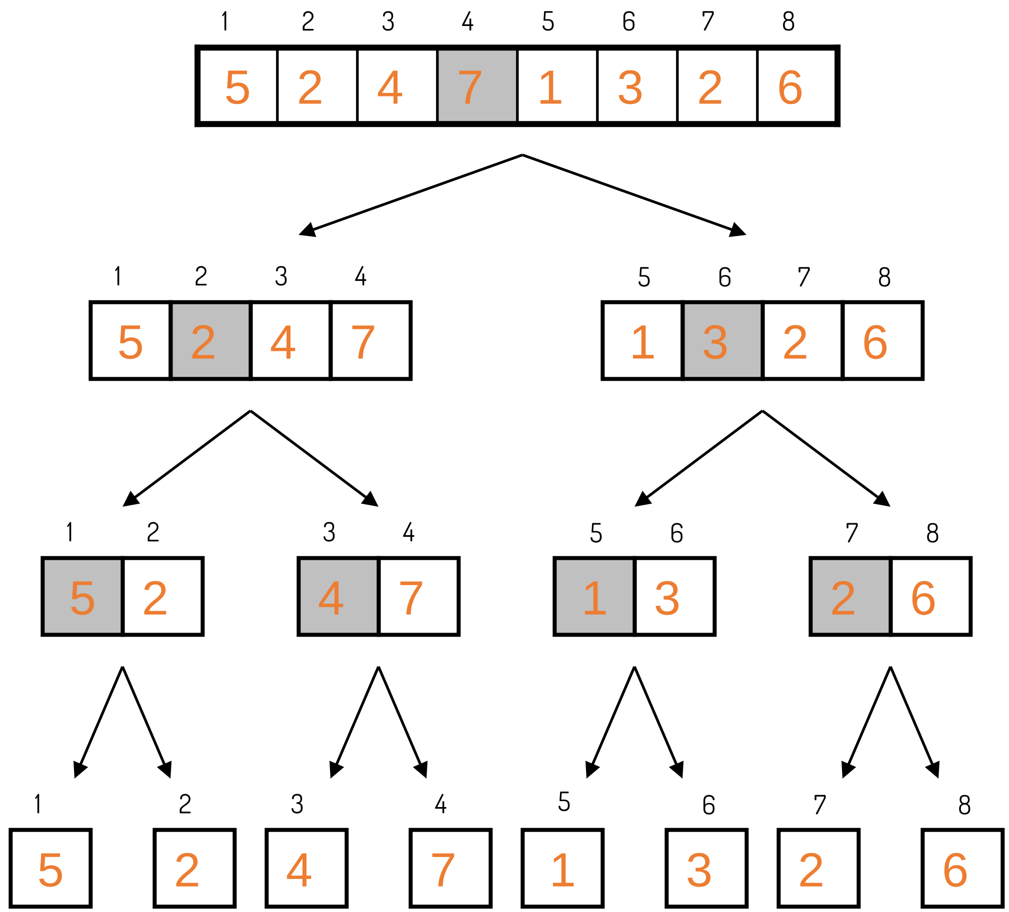

```
MERGE_SORT(A, p, r)

  if p < r                              ▷ Check for base case
    then q ← ⌊(p + r)/2⌋                ▷ Divide
         MERGE-SORT(A, p, q)            ▷ Conquer
         MERGE-SORT(A, q + 1, r)        ▷ Conquer
         MERGE(A, p, q, r)              ▷ Combine
```

- Initial call: MERGE-SORT(A, 1, n)

Example – **n** Power of 2 (Even Number of Elements)

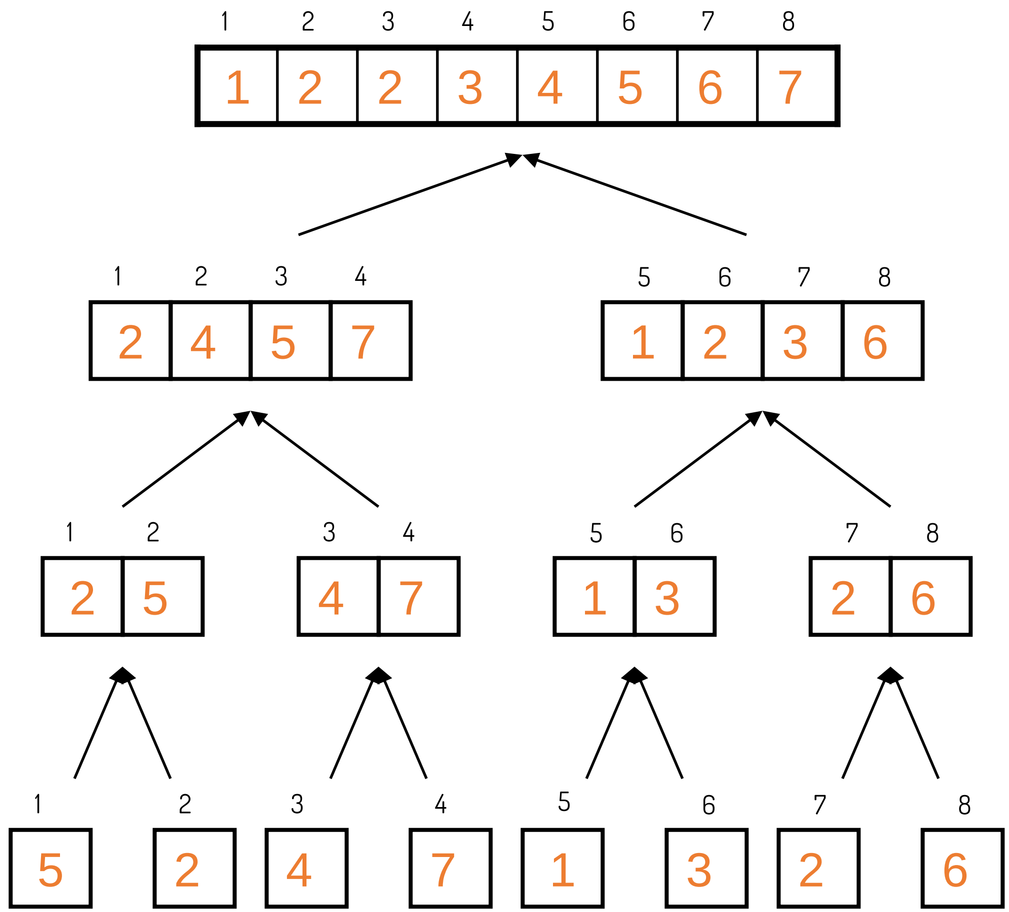
Divide



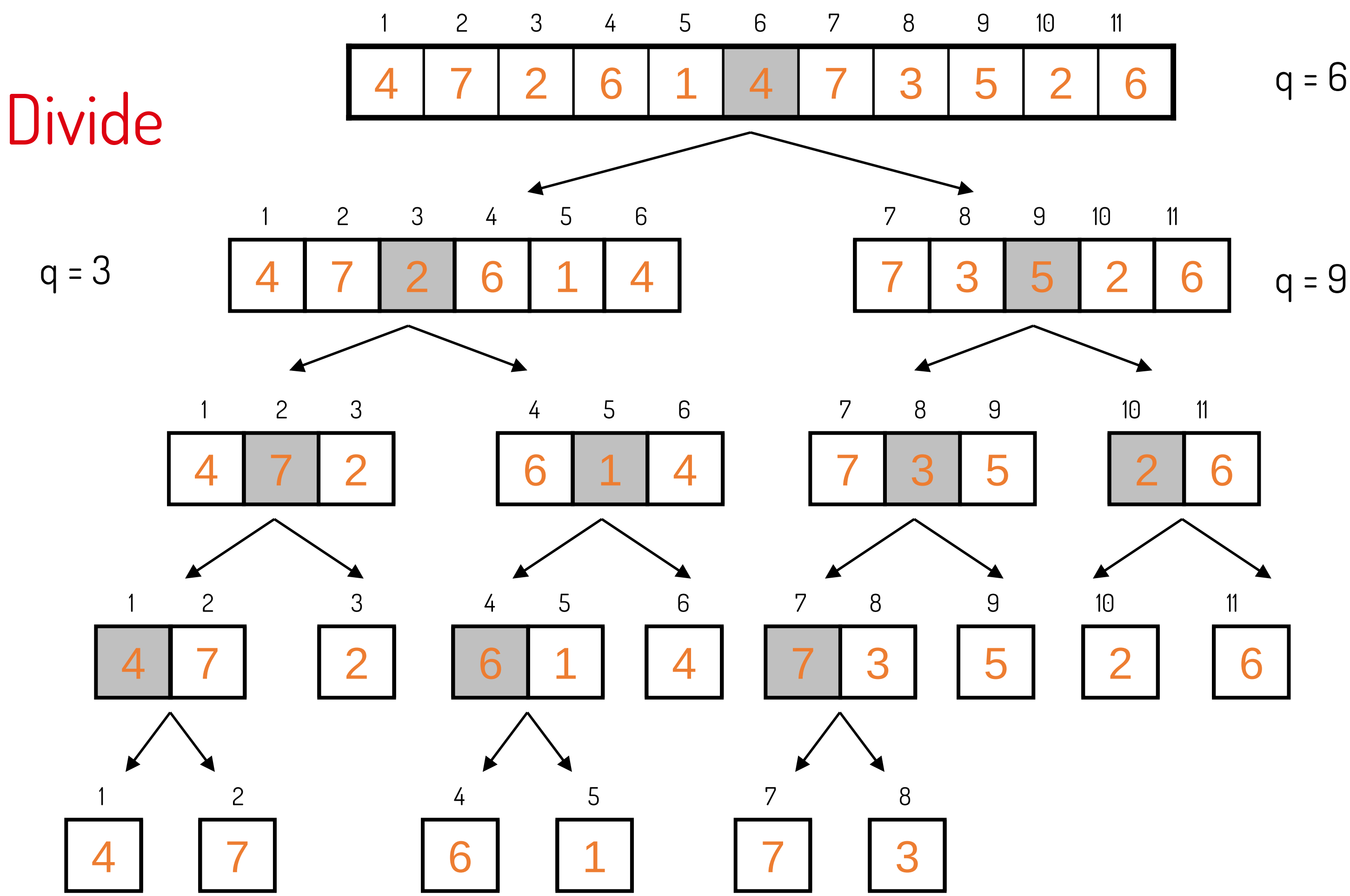
q = 4

Example – **n** Power of 2

Conquer and Merge

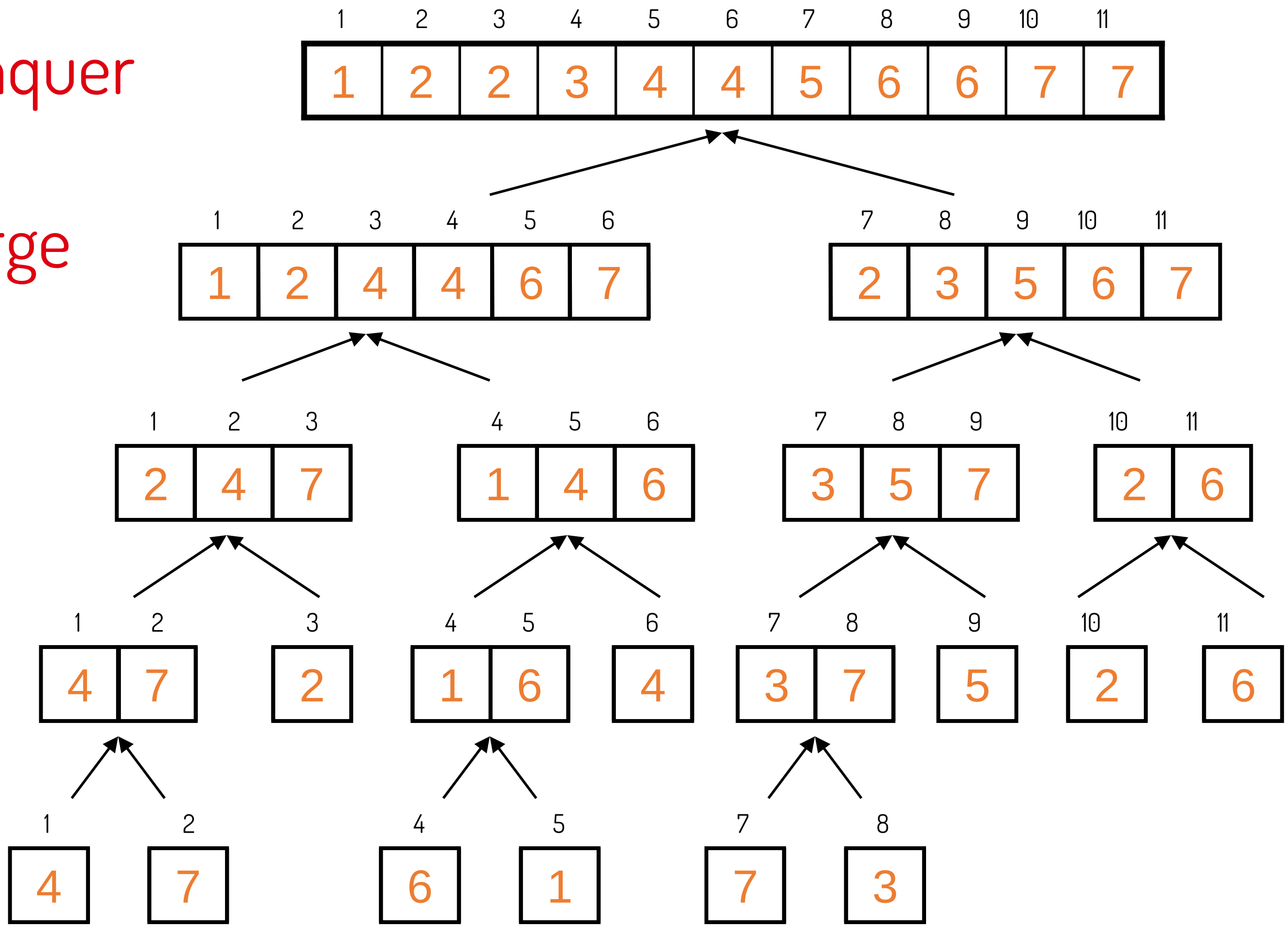


Example – *n* Not a Power of 2 (Odd number of elements)

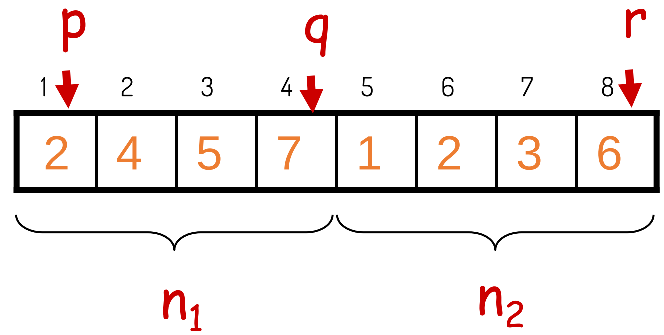


Example – **n** Not a Power of 2

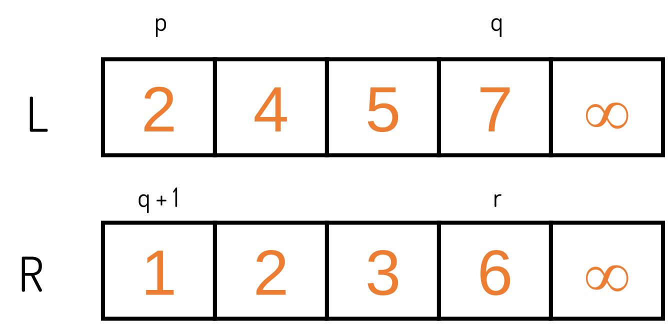
Conquer and Merge



Merge - Pseudocode

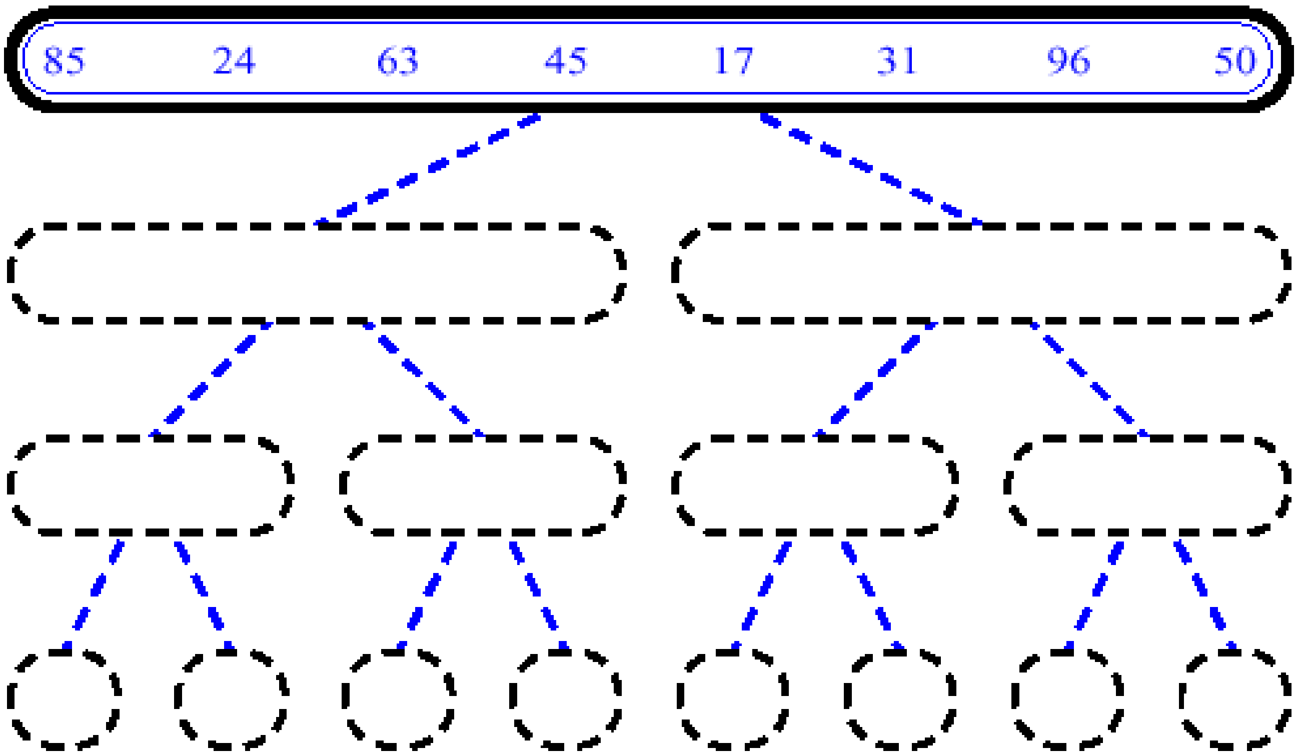



```
MERGE(A, p, q, r)
n1 = q - p + 1
n2 = r - q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i - 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

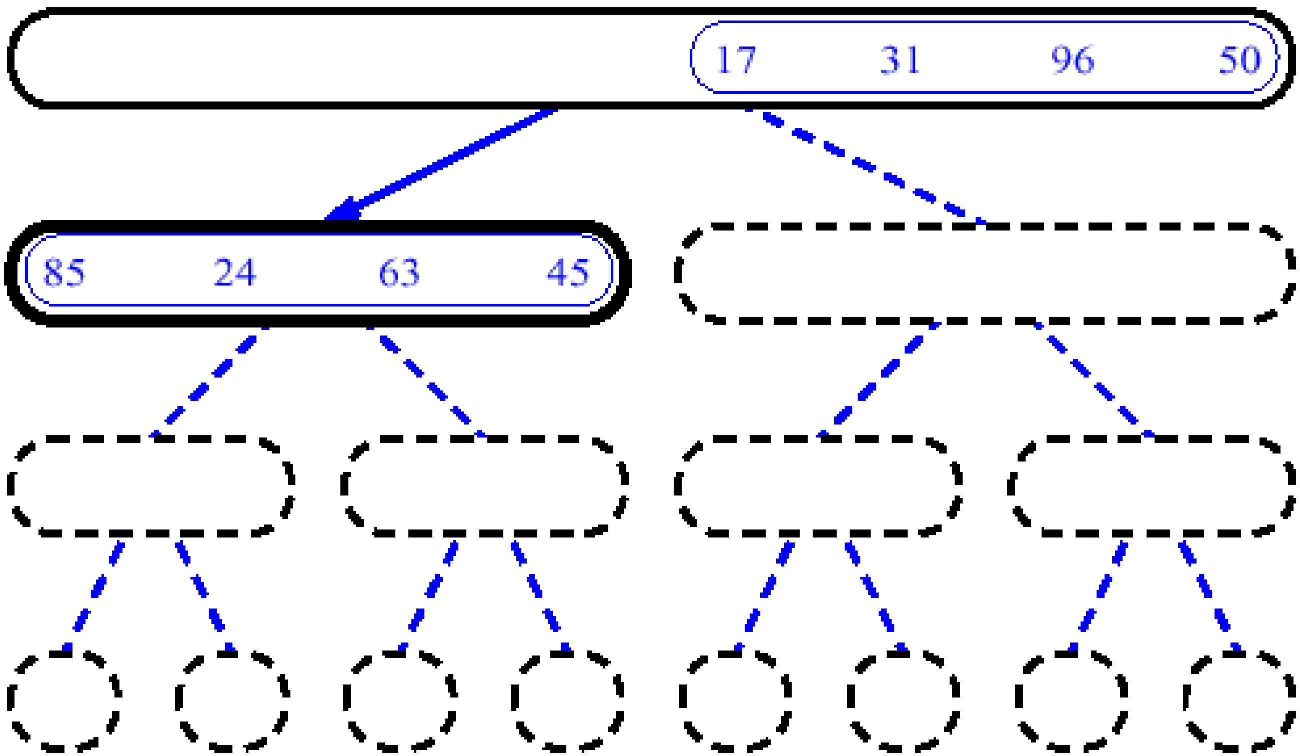


For Lab:
https://www.tutorialspoint.com/data_structures_algorithms/merge_sort_program_in_c.htm

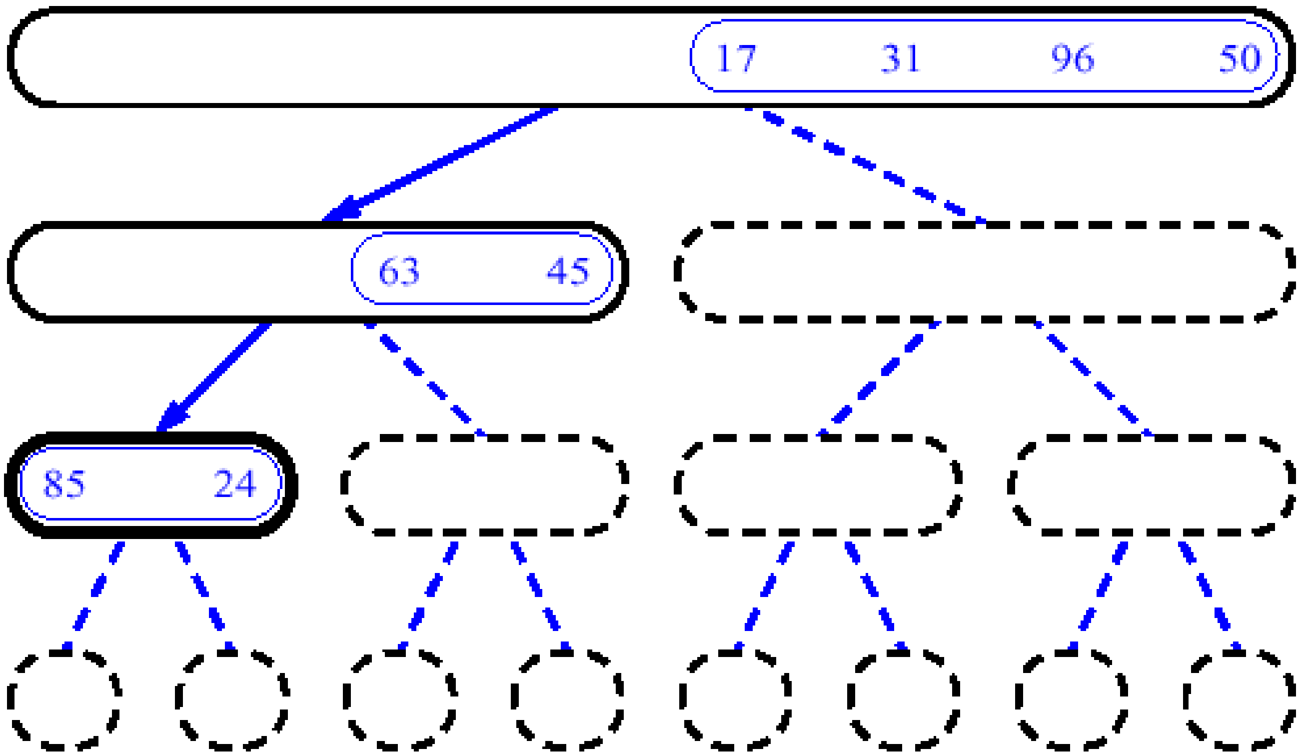
MergeSort (Example) - 1



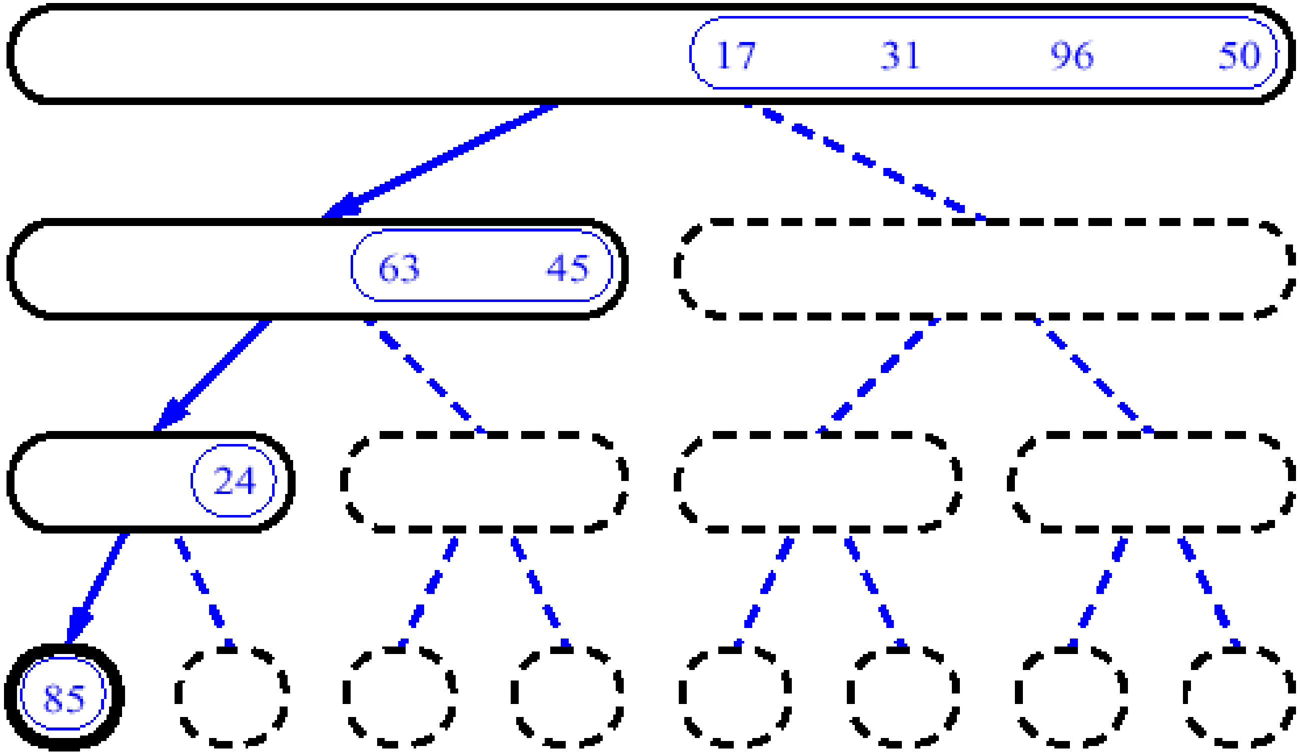
MergeSort (Example) - 2



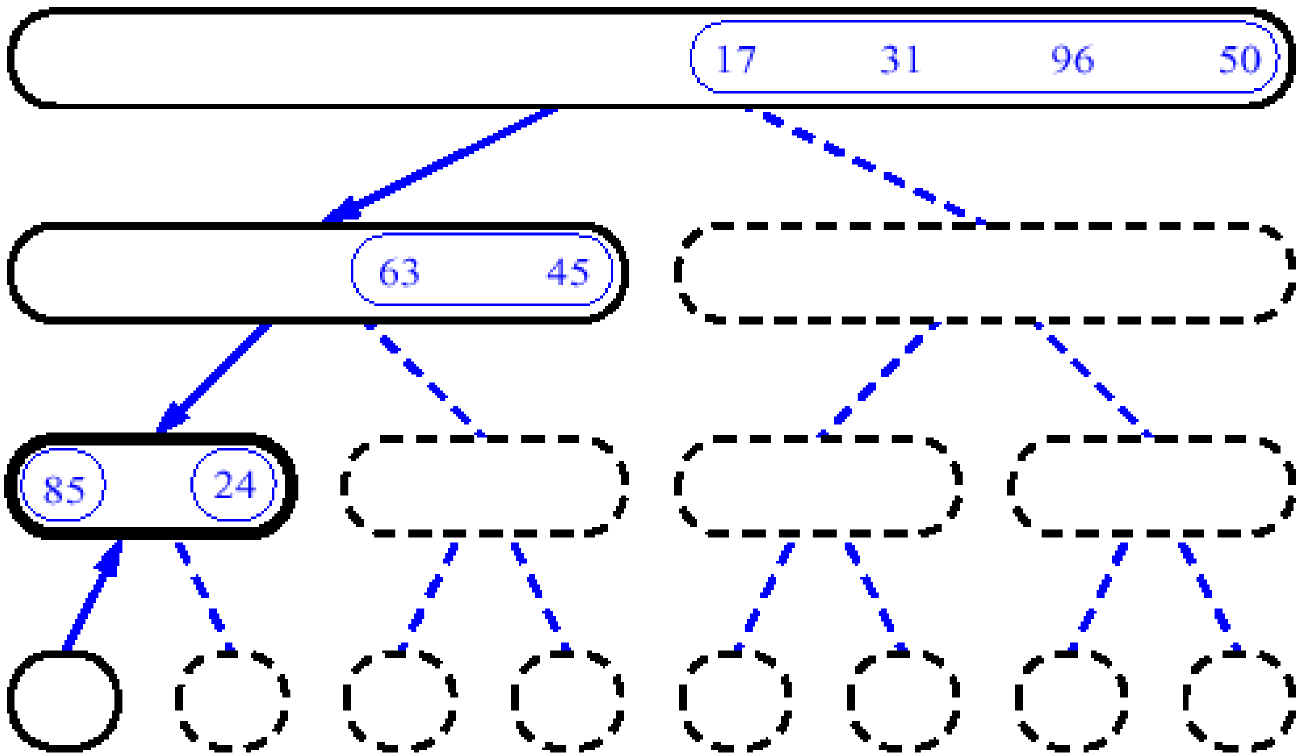
MergeSort (Example) - 3



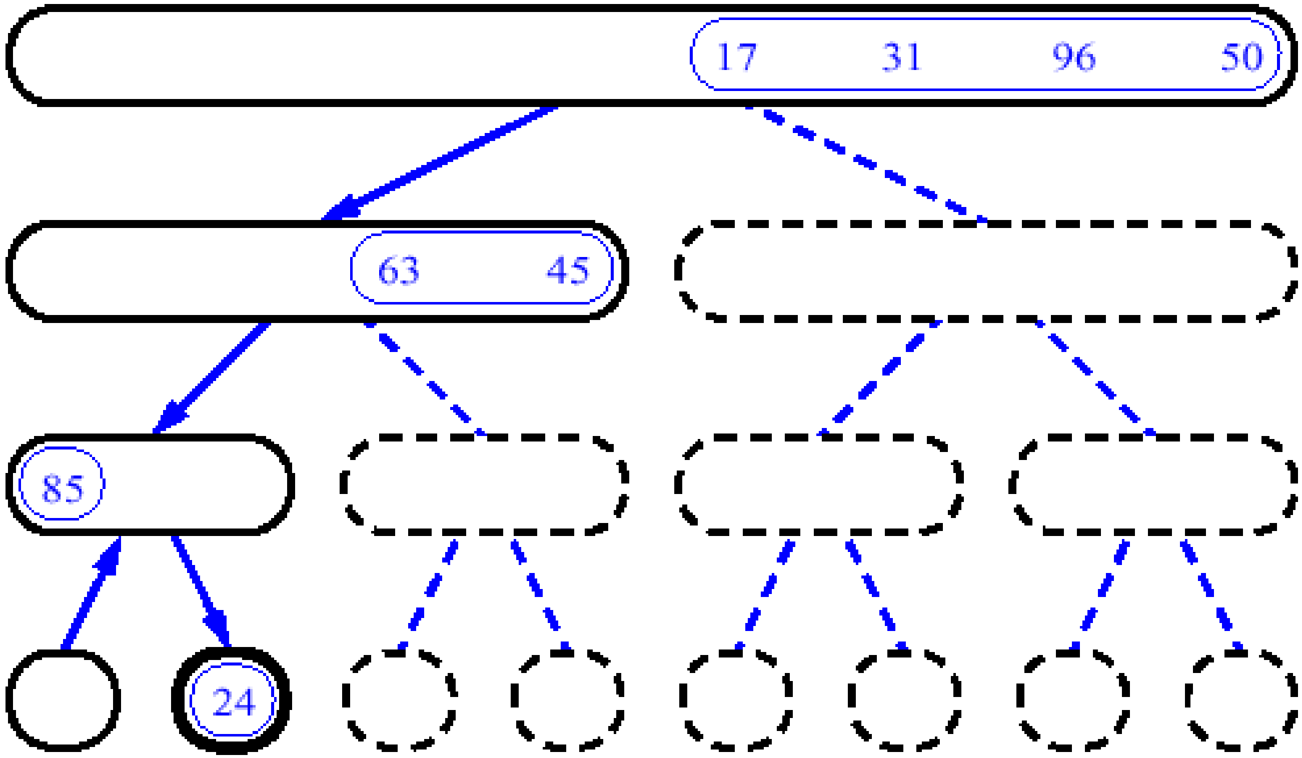
MergeSort (Example) - 4



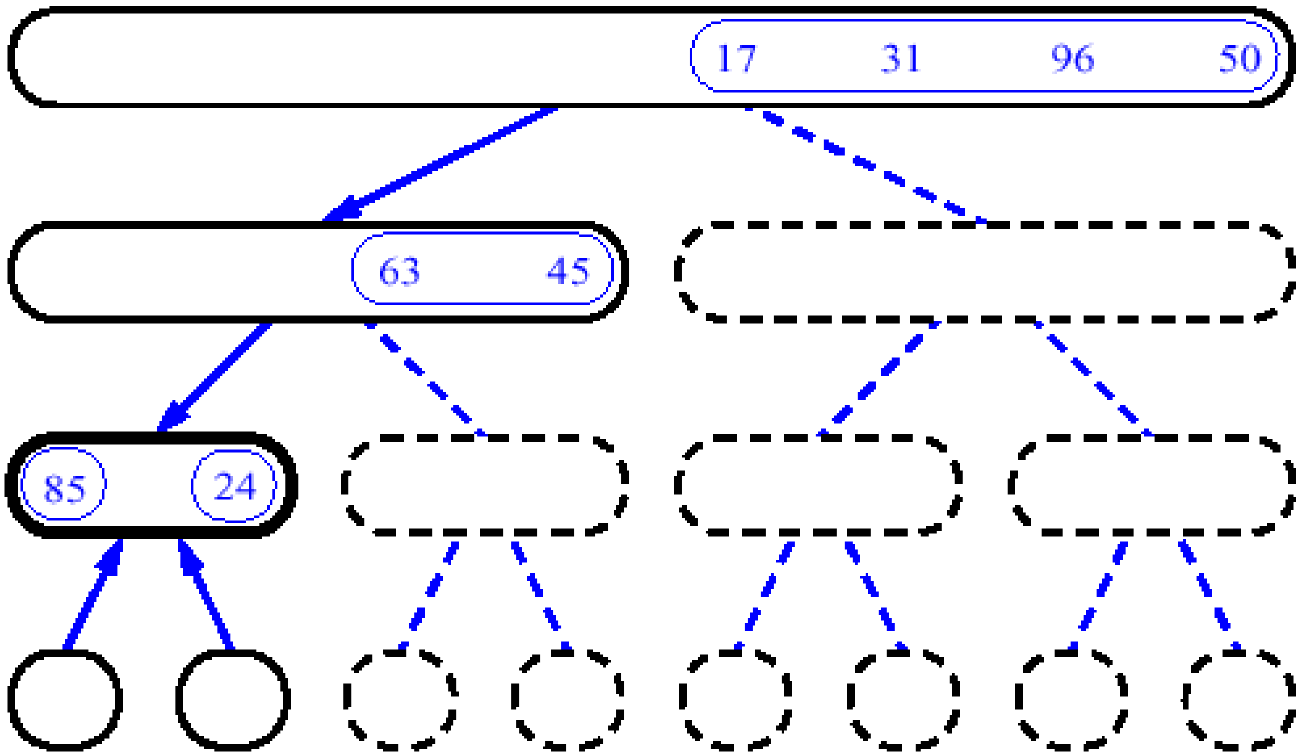
MergeSort (Example) - 5



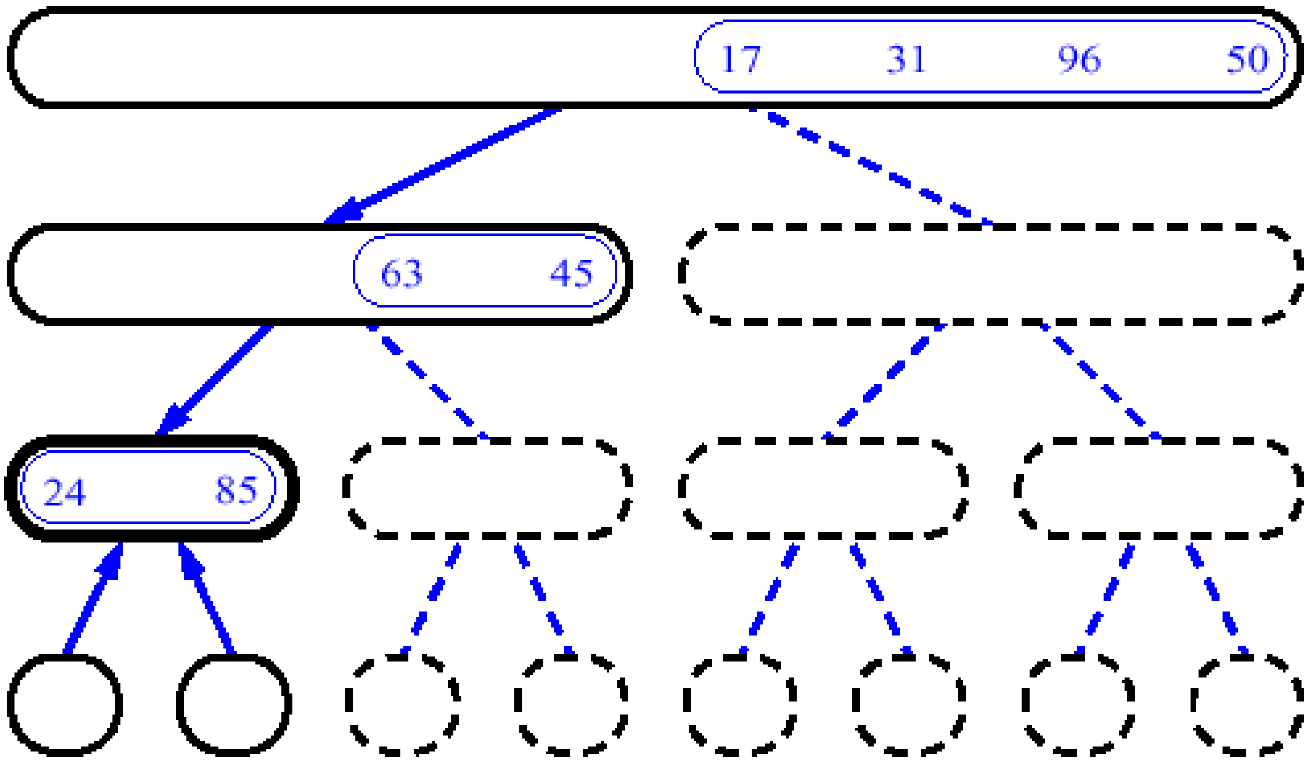
MergeSort (Example) - 6



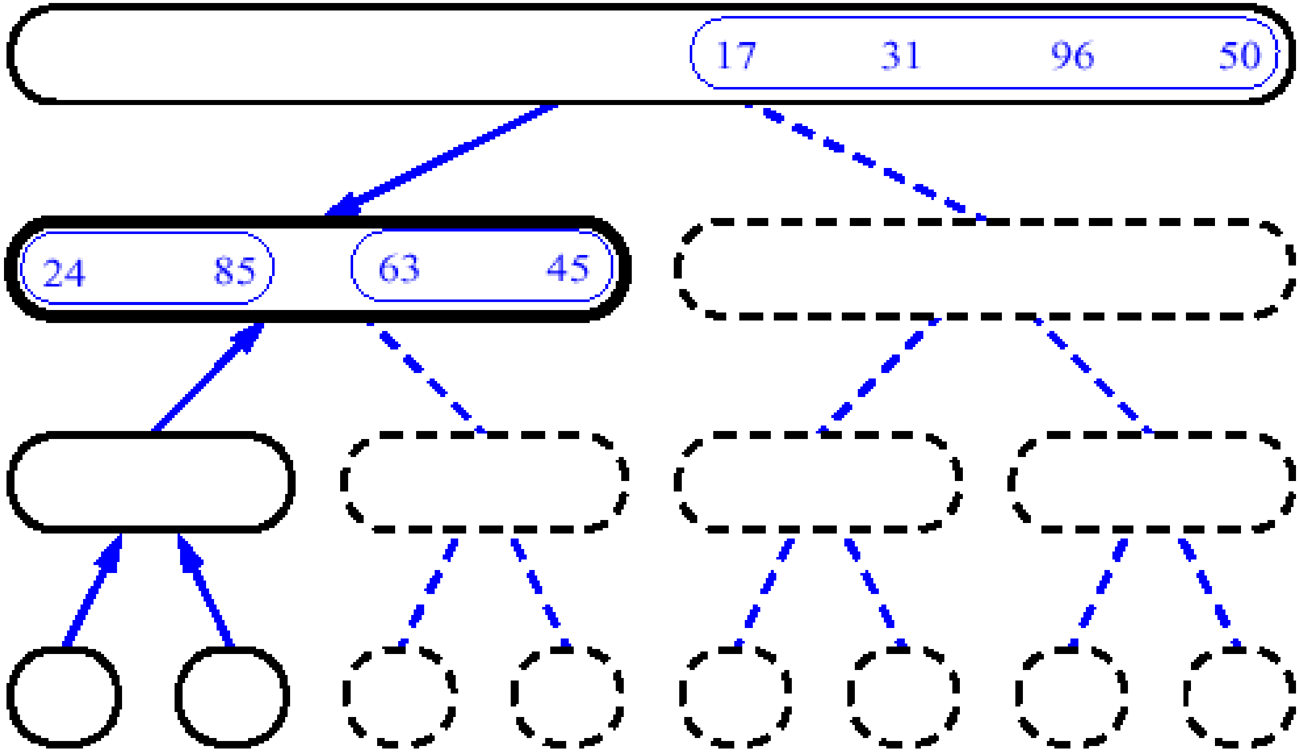
MergeSort (Example) - 7



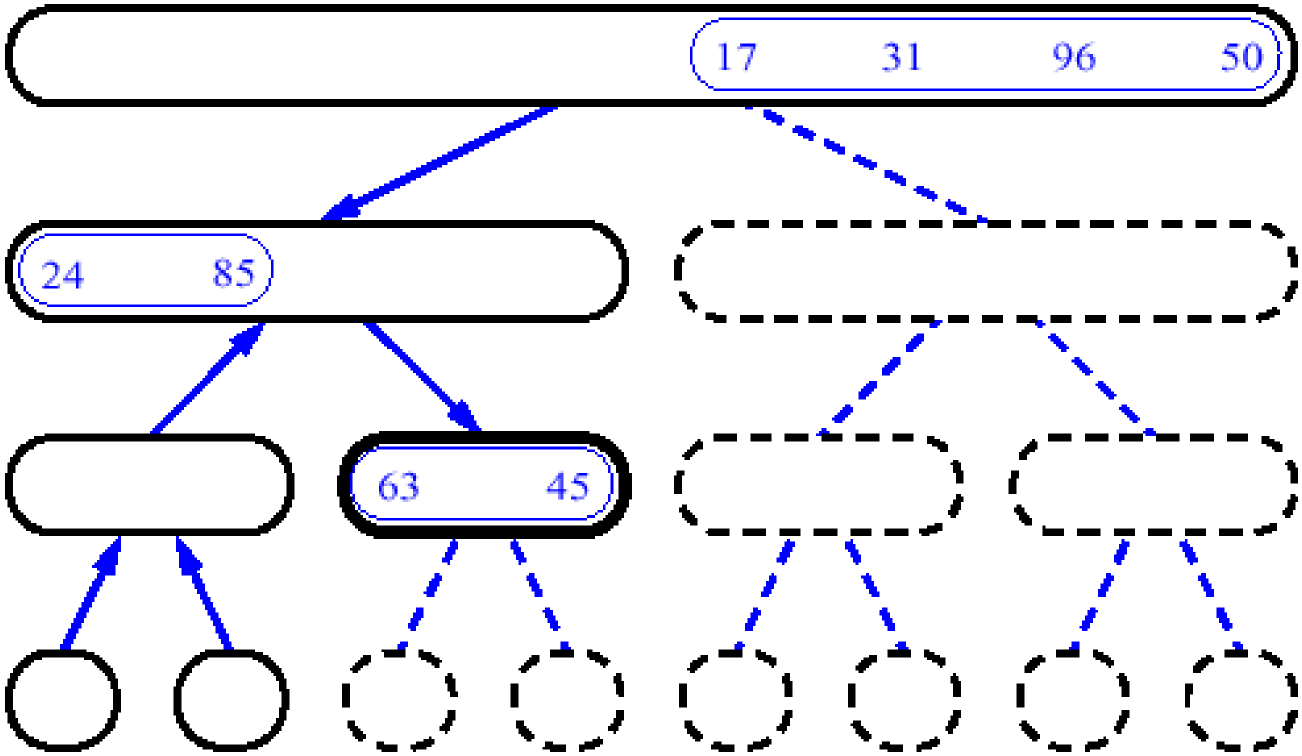
MergeSort (Example) - 8



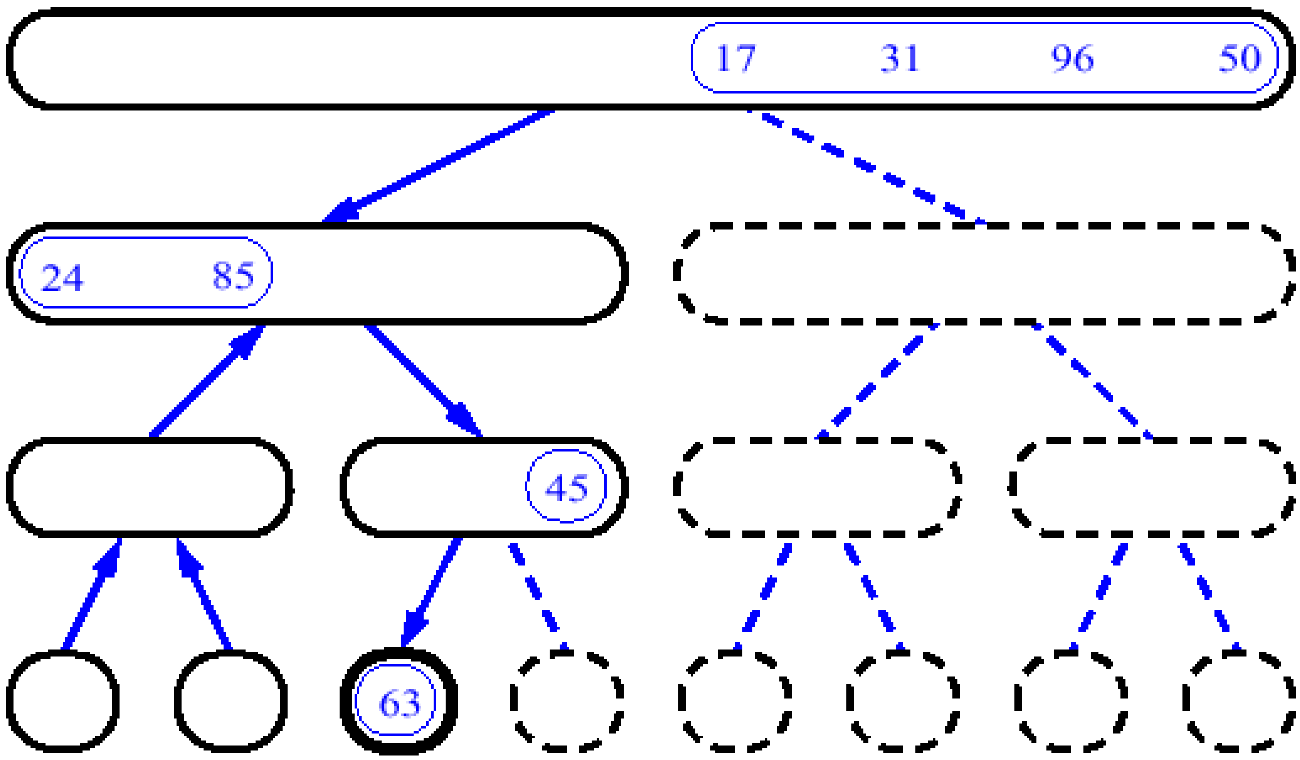
MergeSort (Example) - 9



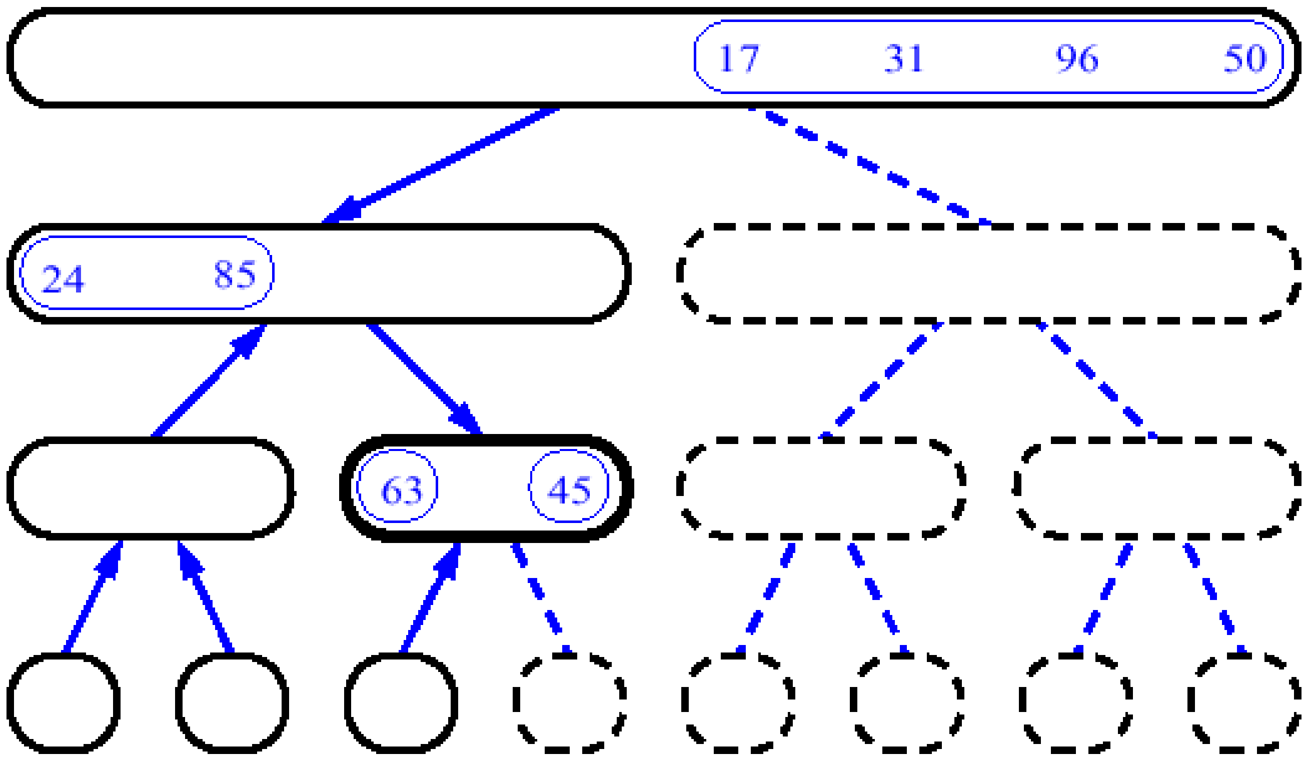
MergeSort (Example) - 10



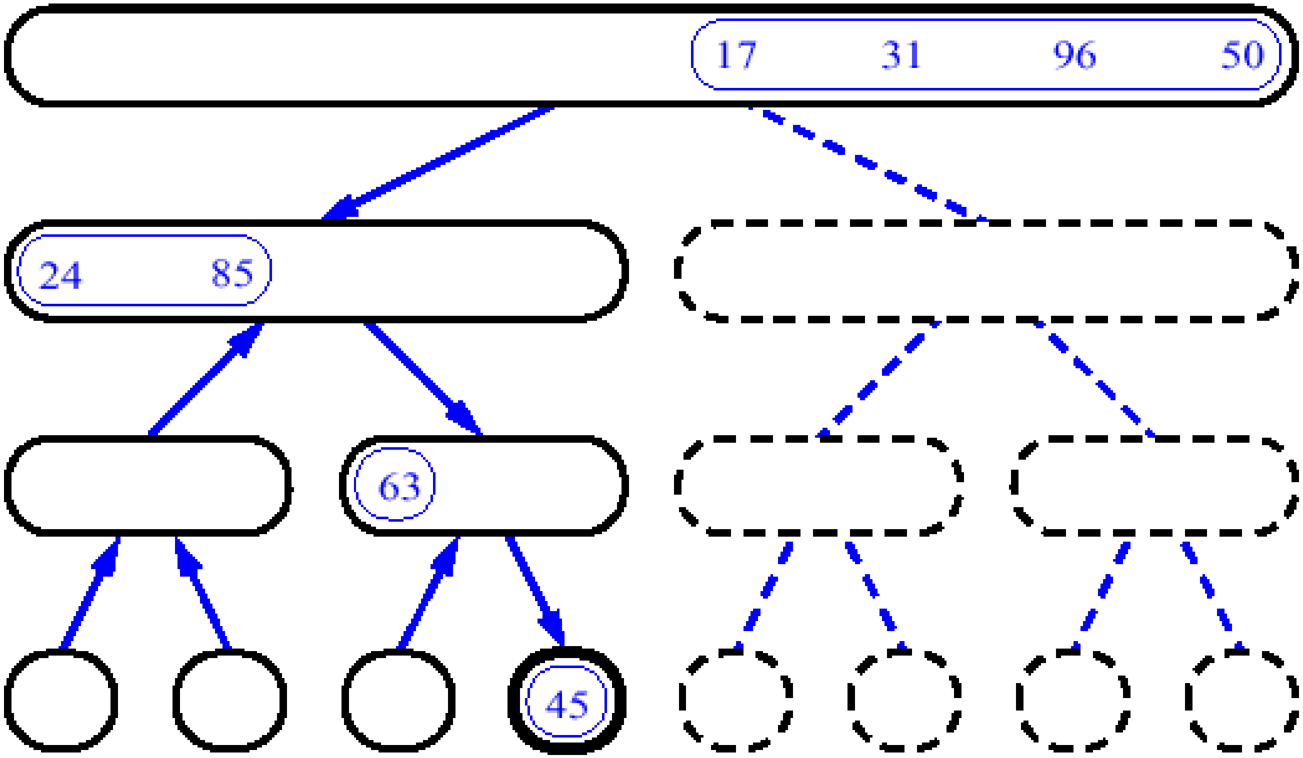
MergeSort (Example) - 11



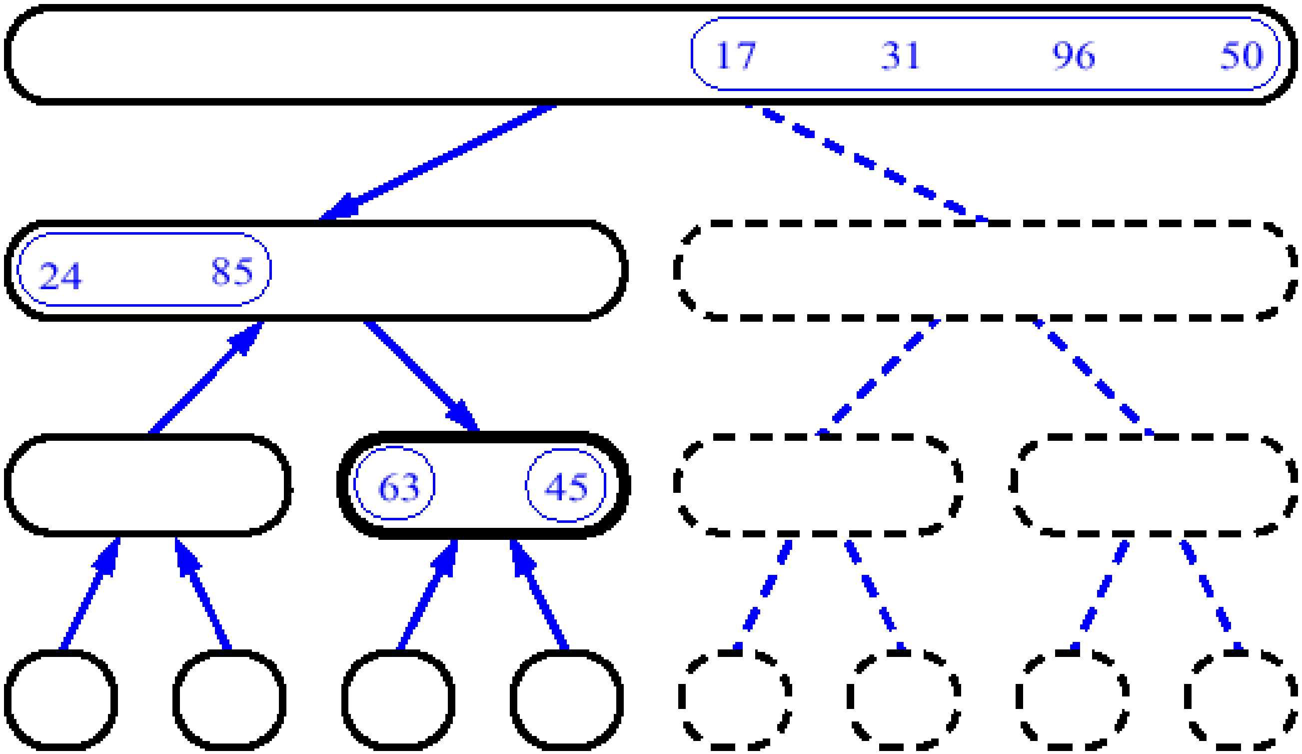
MergeSort (Example) - 12



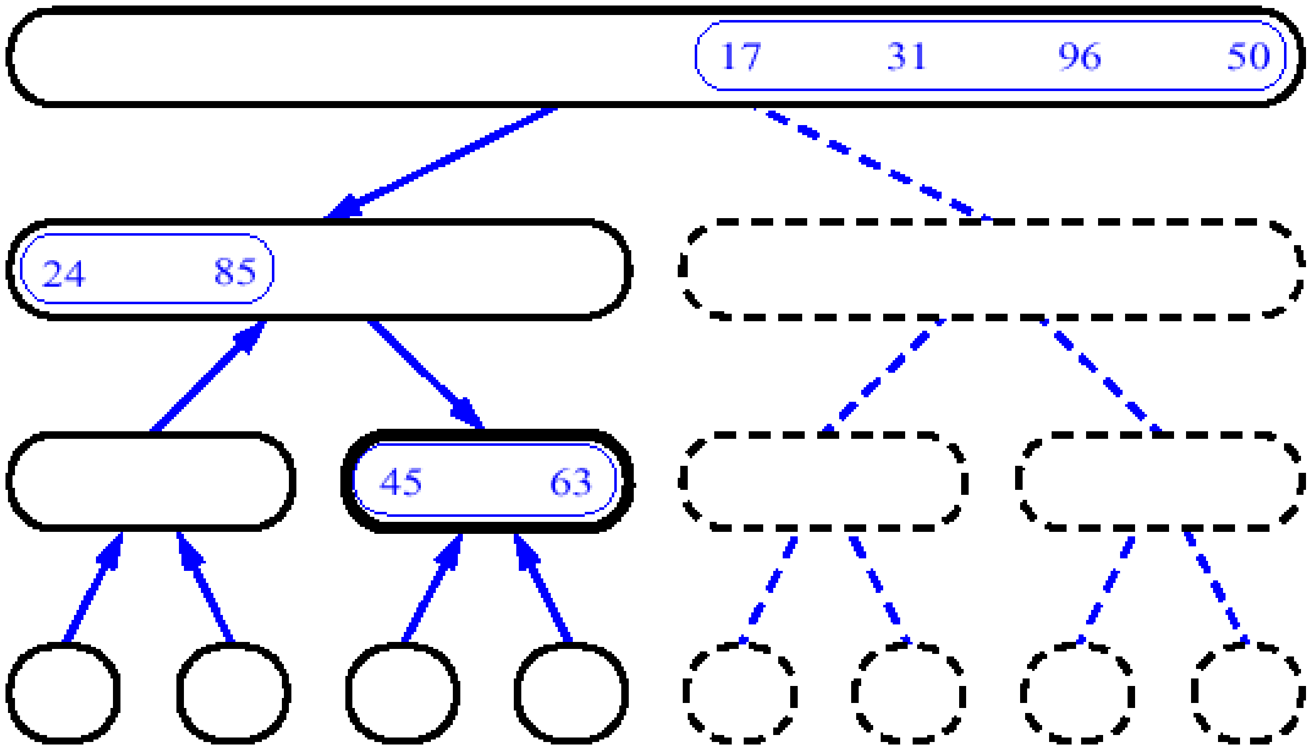
MergeSort (Example) - 13



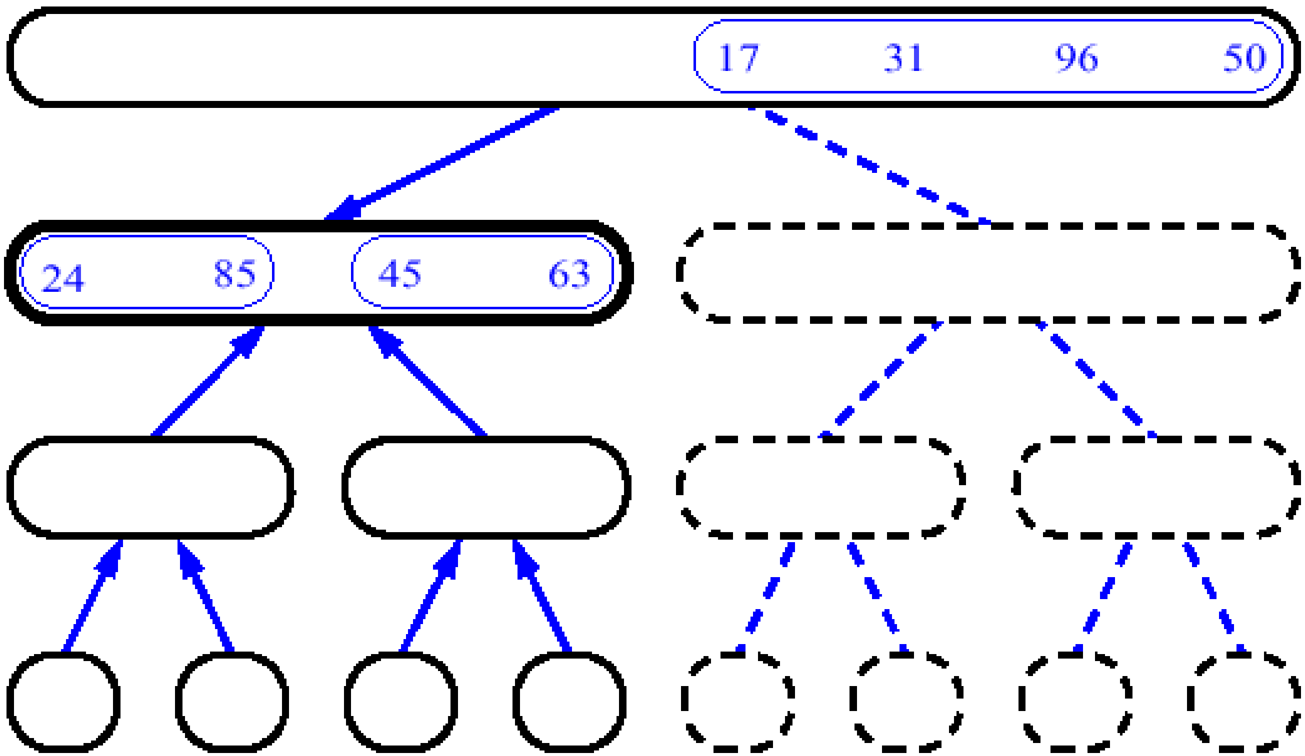
MergeSort (Example) - 14



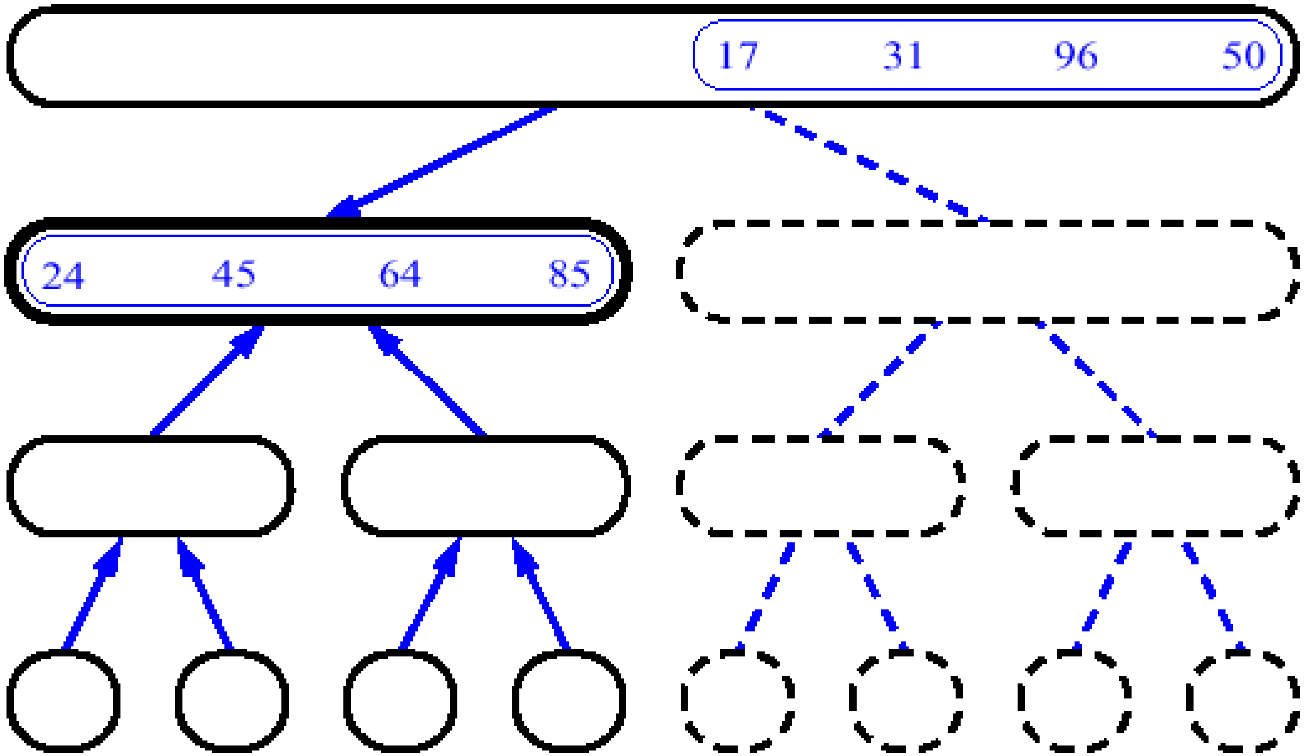
MergeSort (Example) - 15



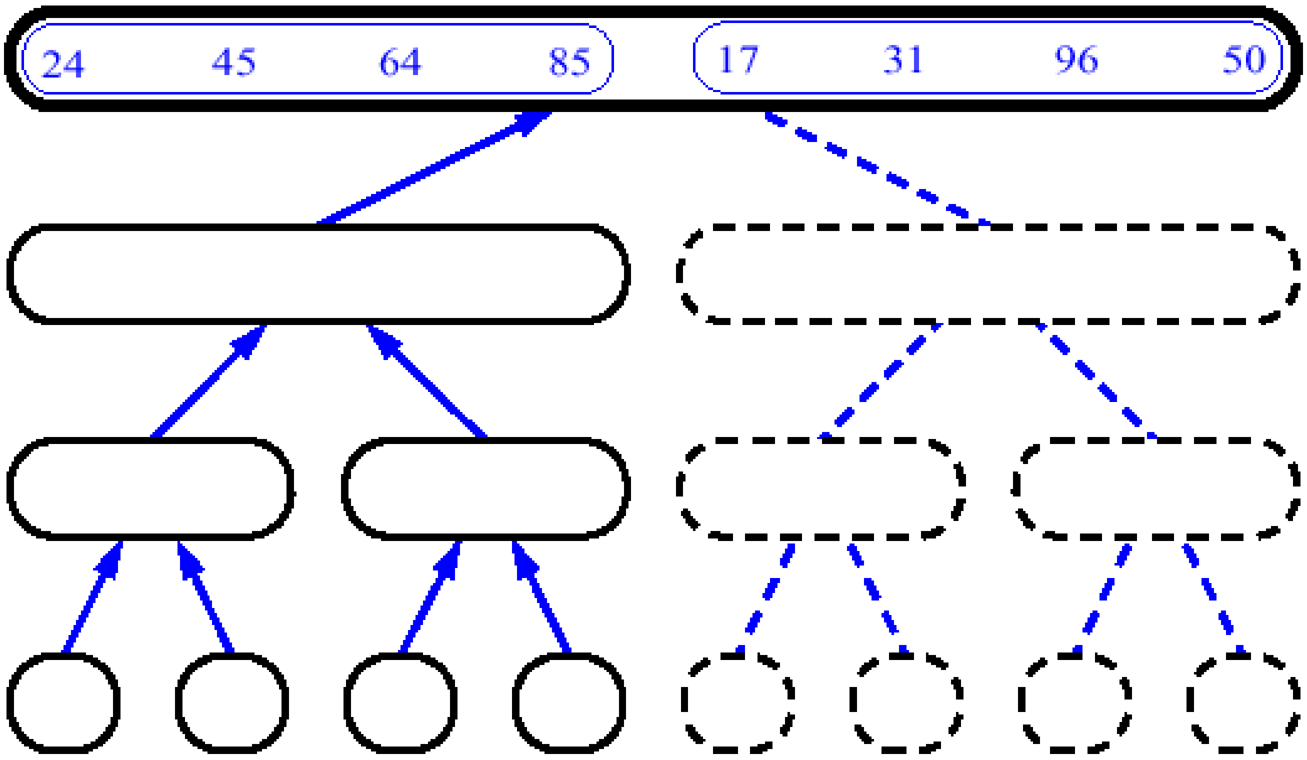
MergeSort (Example) - 16



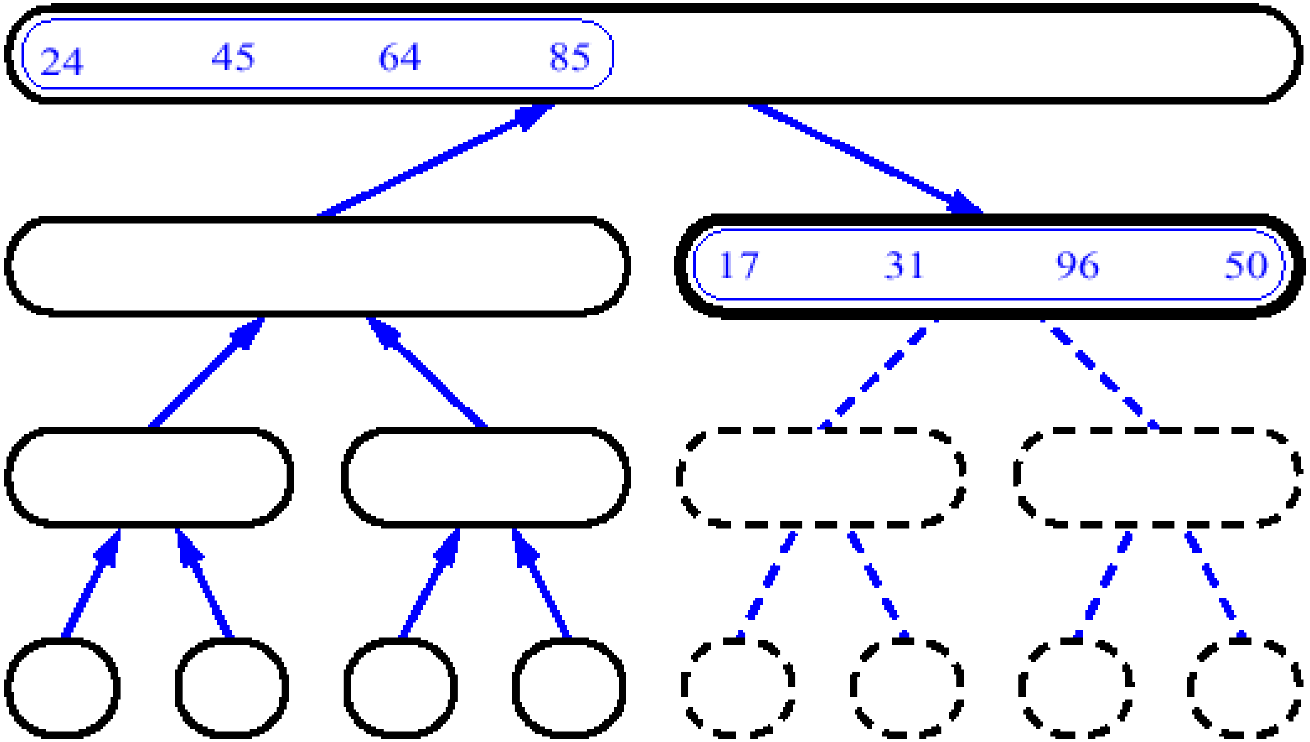
MergeSort (Example) - 17



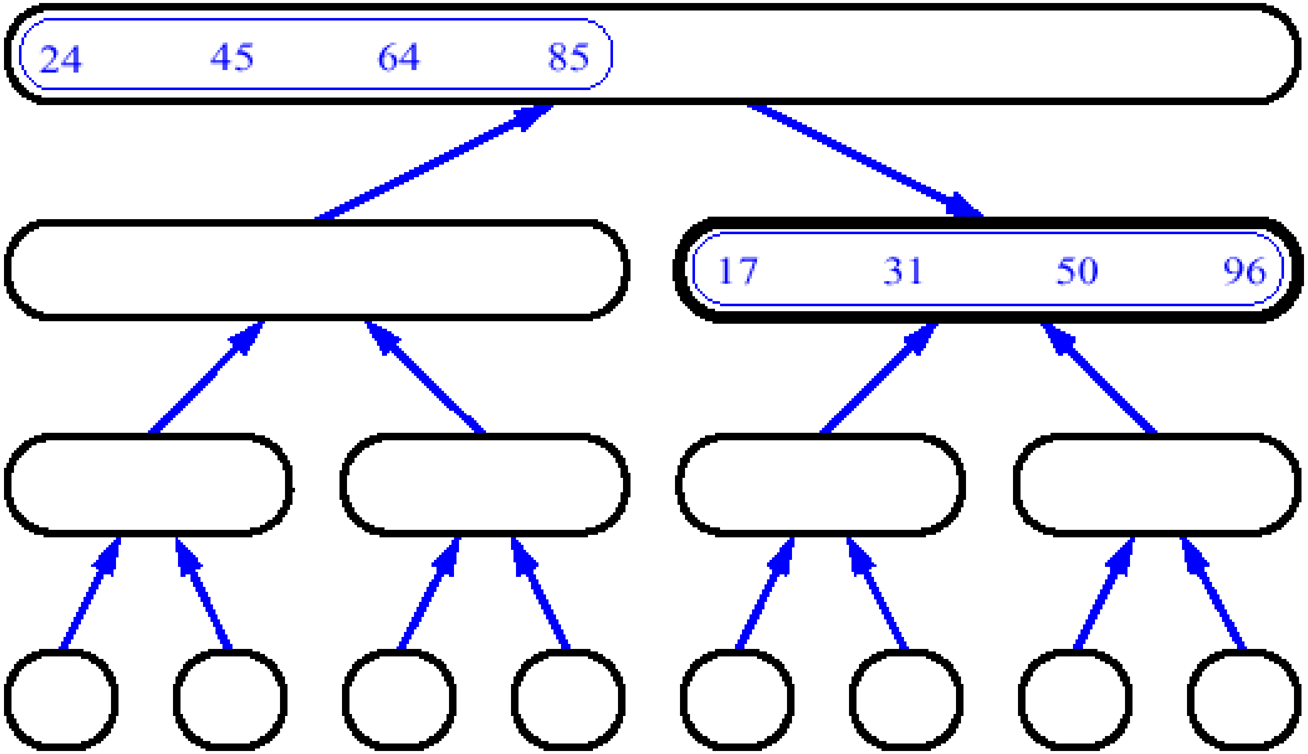
MergeSort (Example) - 18



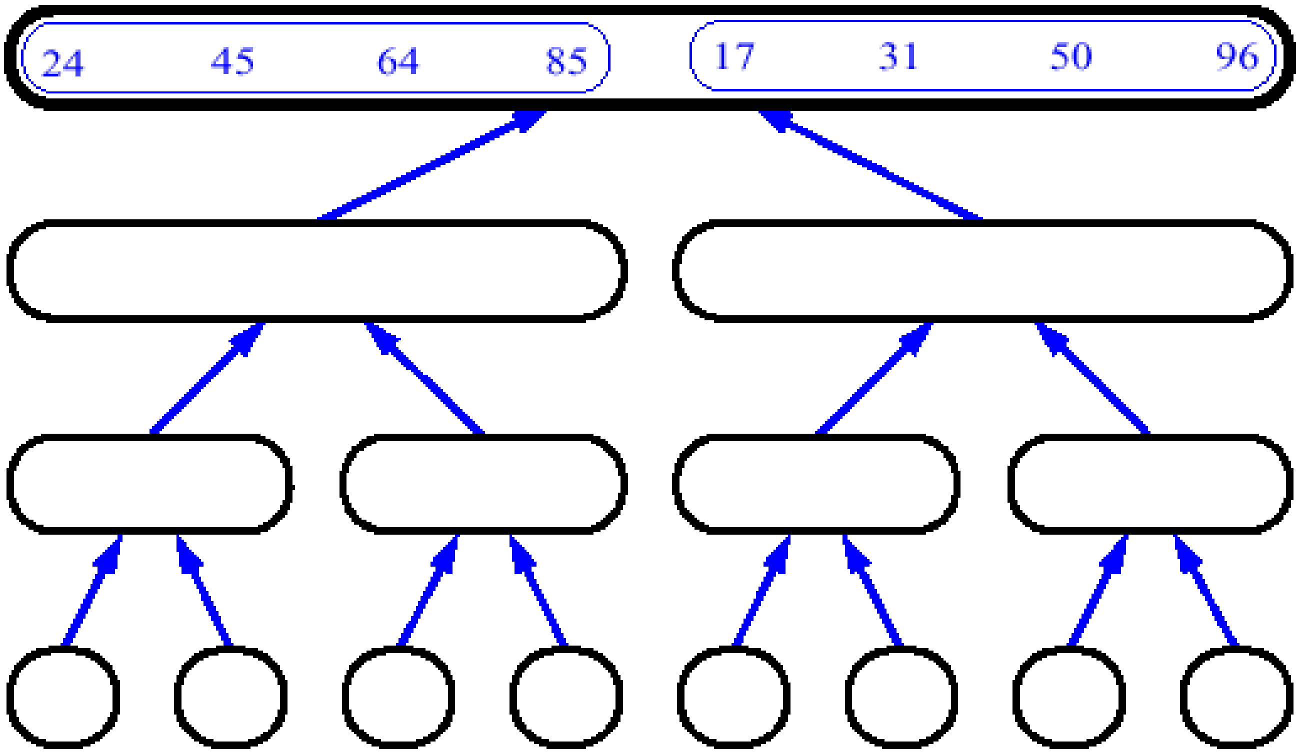
MergeSort (Example) - 19



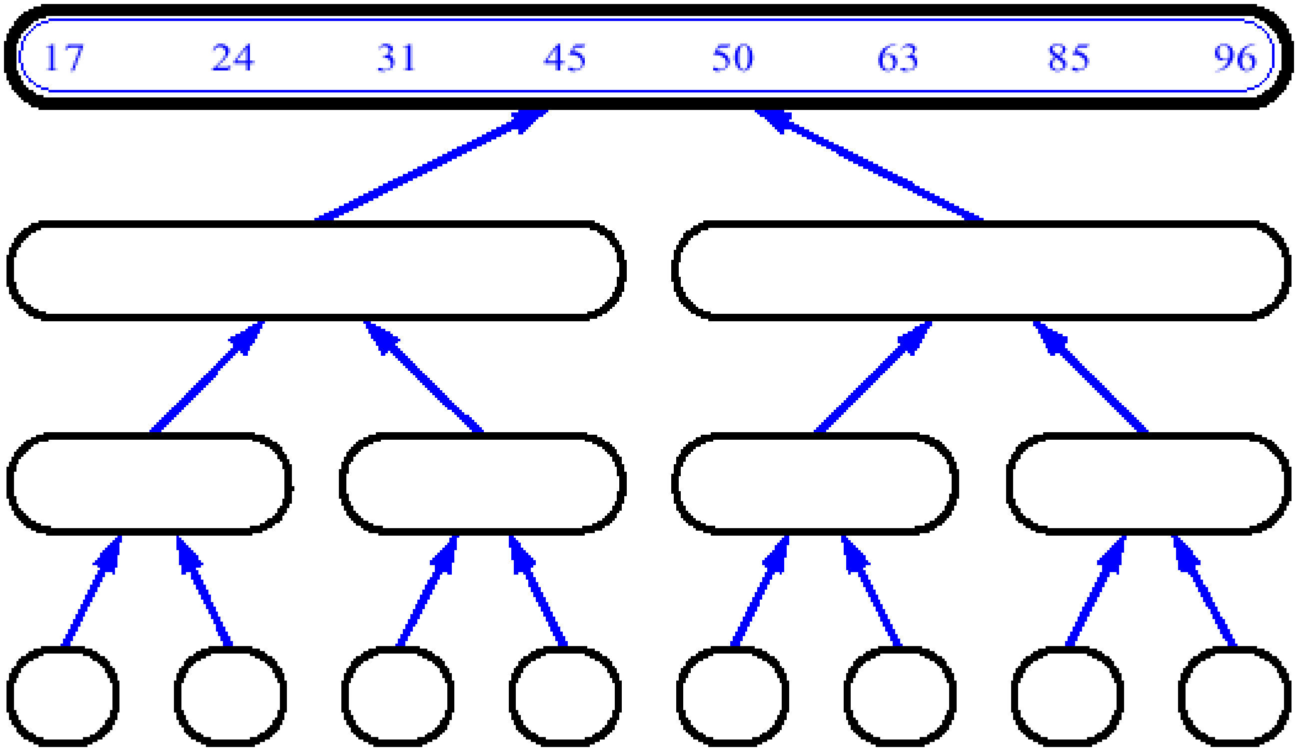
MergeSort (Example) - 20



MergeSort (Example) - 21

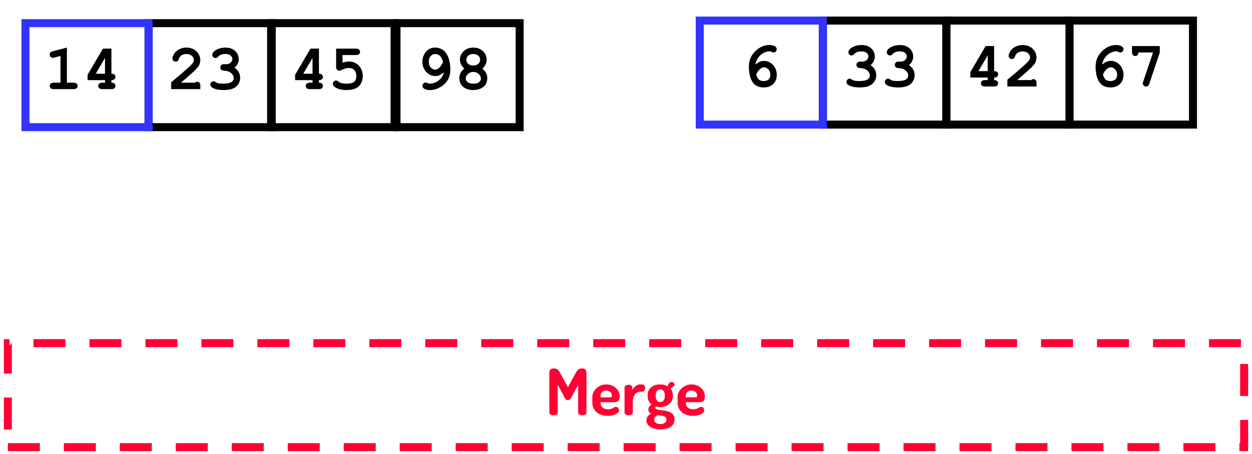


MergeSort (Example) - 22

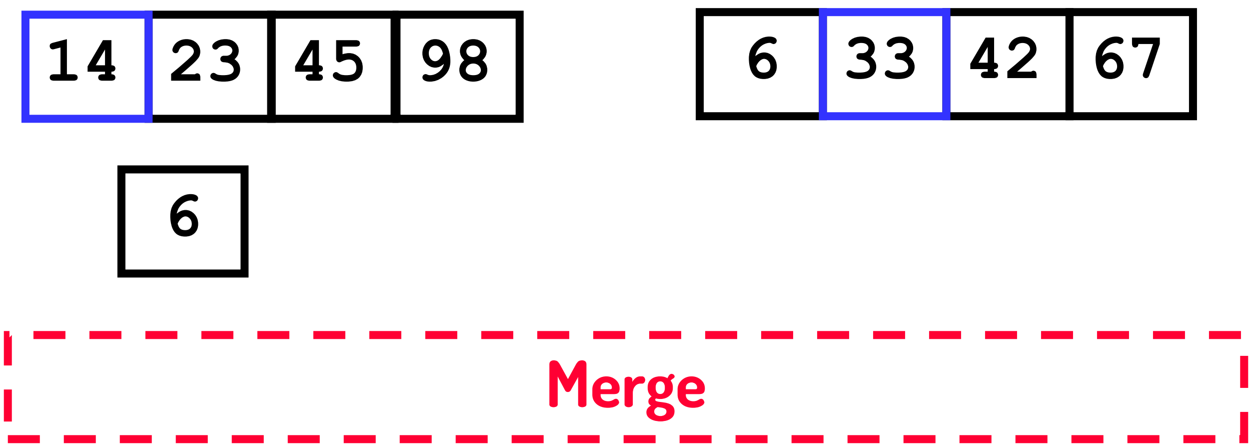


14	23	45	98

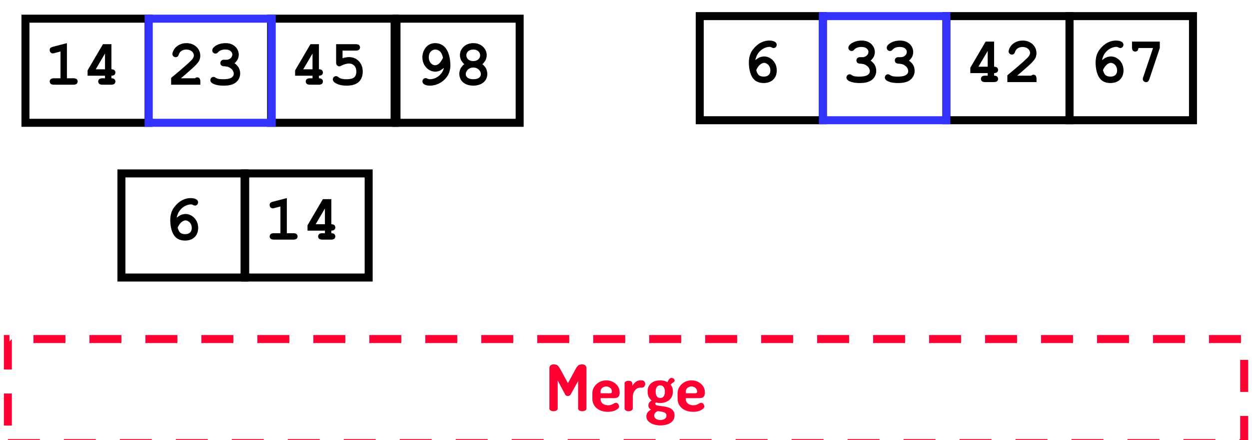
6	33	42	67

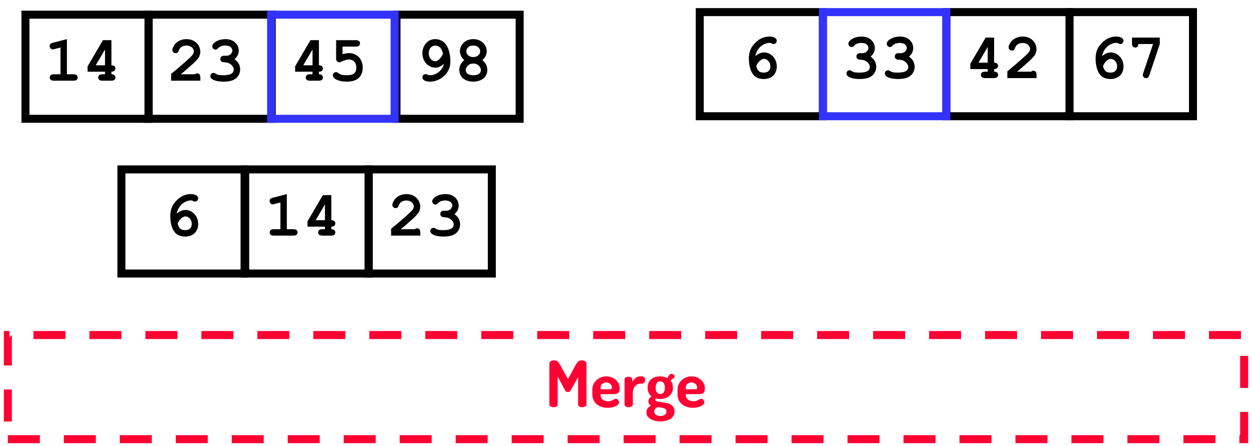


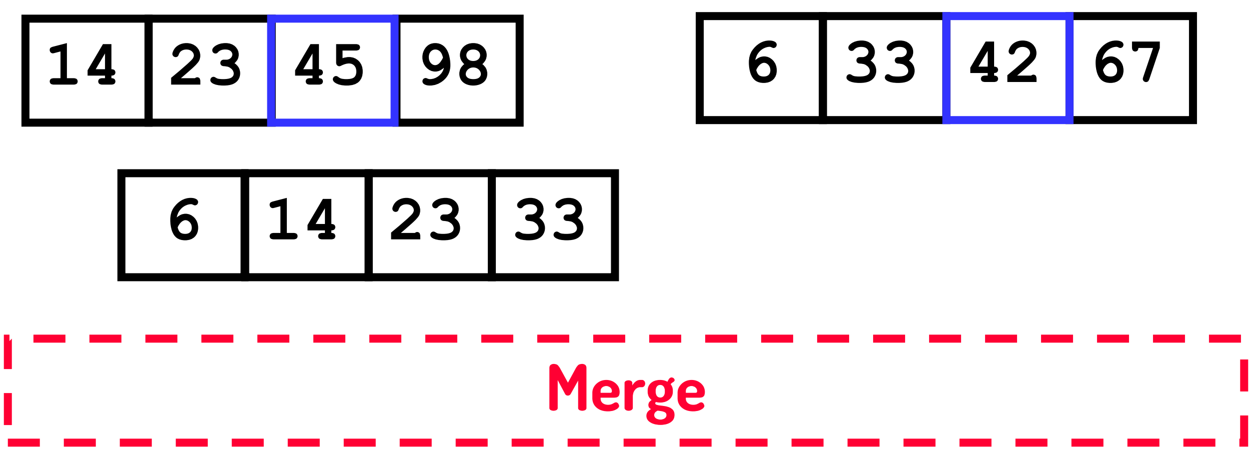
14 23 45 98
6 33 42 67
Merge

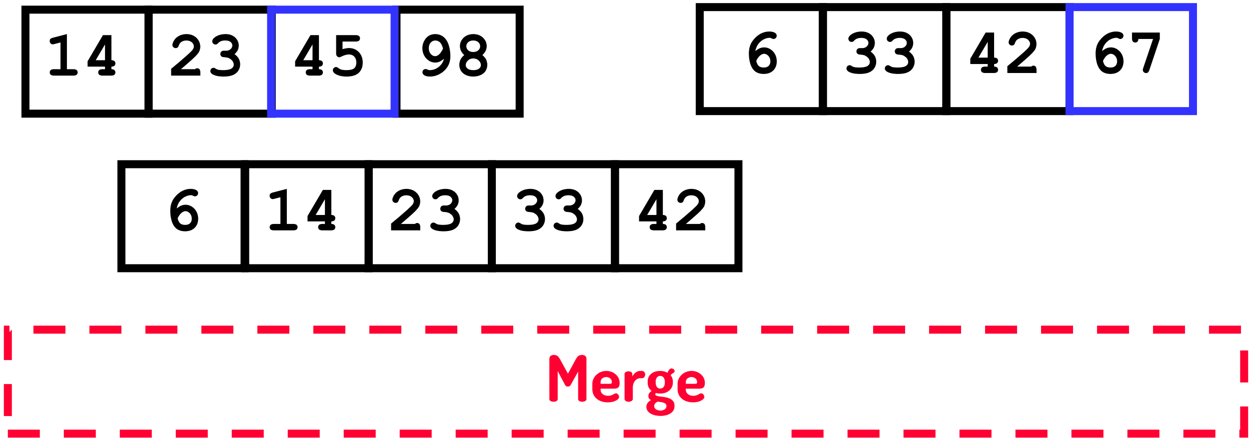


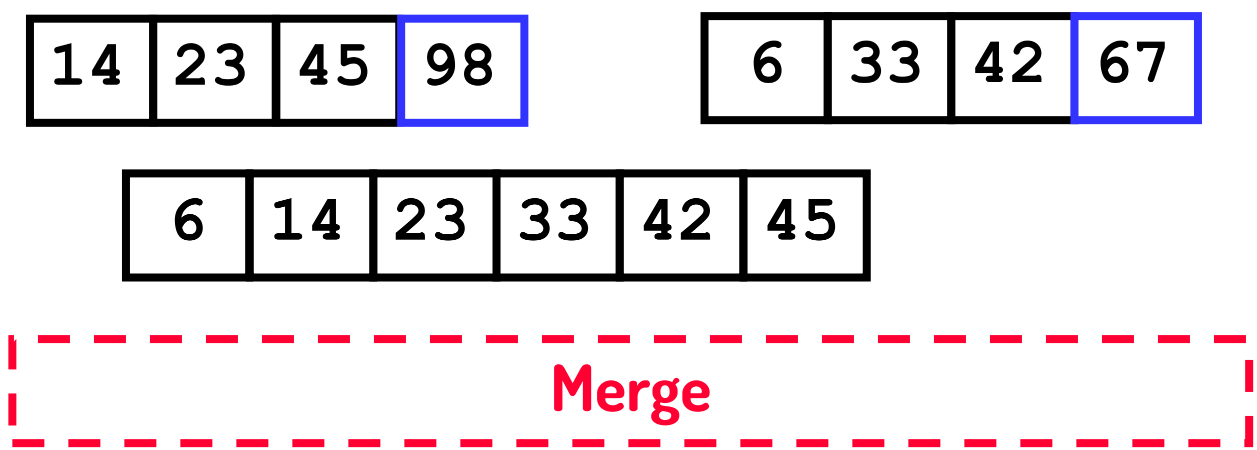
14
23
45
98
6
33
42
67
6
Merge

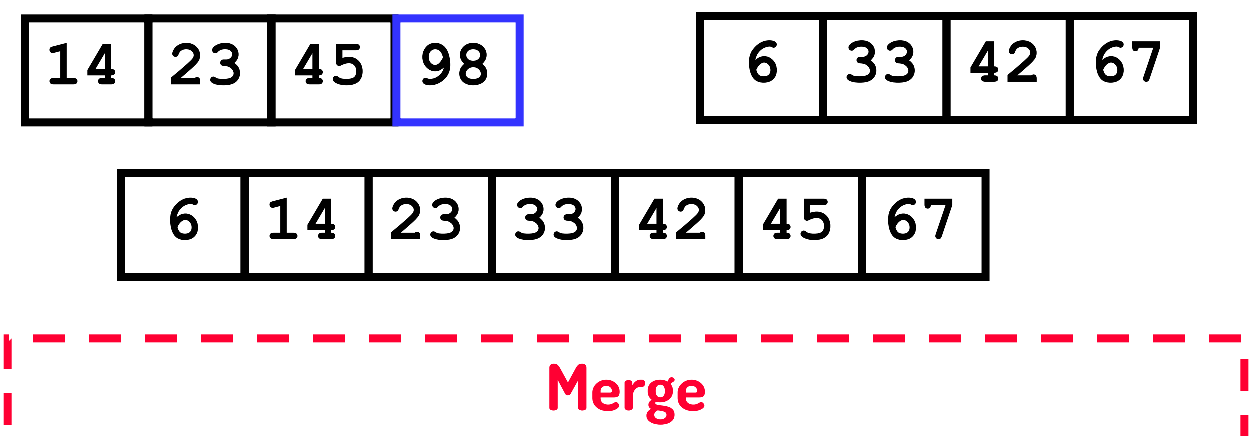

14 23 45 98
6 33 42 67
6 14
Merge

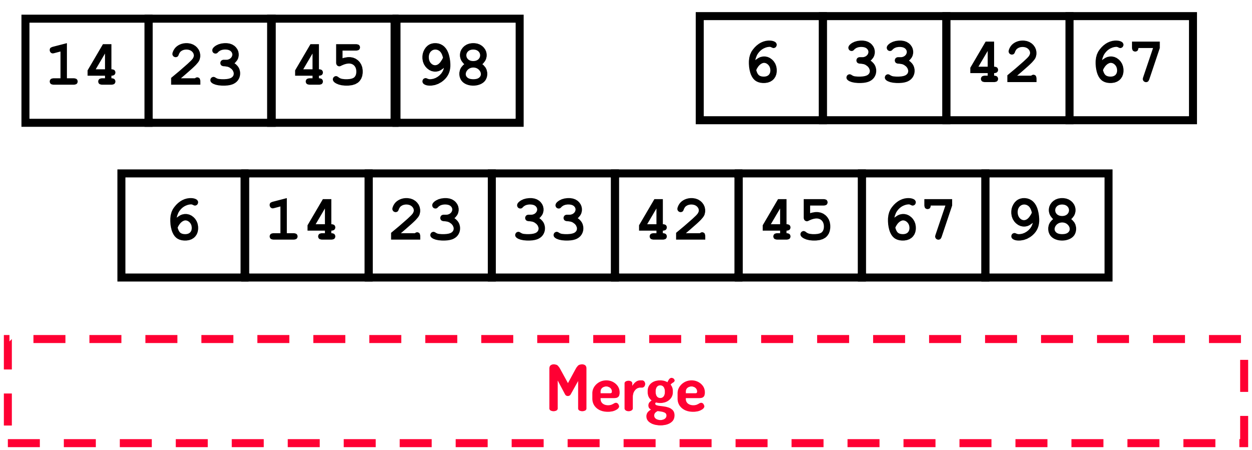

14
23
45
98
6
33
42
67
6
14
23
Merge


14 23 45 98
6 33 42 67
6 14 23 33
Merge


14
23
45
98
6
33
42
67
6
14
23
33
42
Merge


14 23 45 98
6 33 42 67
6 14 23 33 42 45
Merge


14 23 45 98
6 33 42 67
6 14 23 33 42 45 67
Merge



14 23 45 98
6 33 42 67
6 14 23 33 42 45 67 98
Merge

A Useful Recurrence Relation

Def. $T(n)$ = number of comparisons of merge sort for an input of size n.

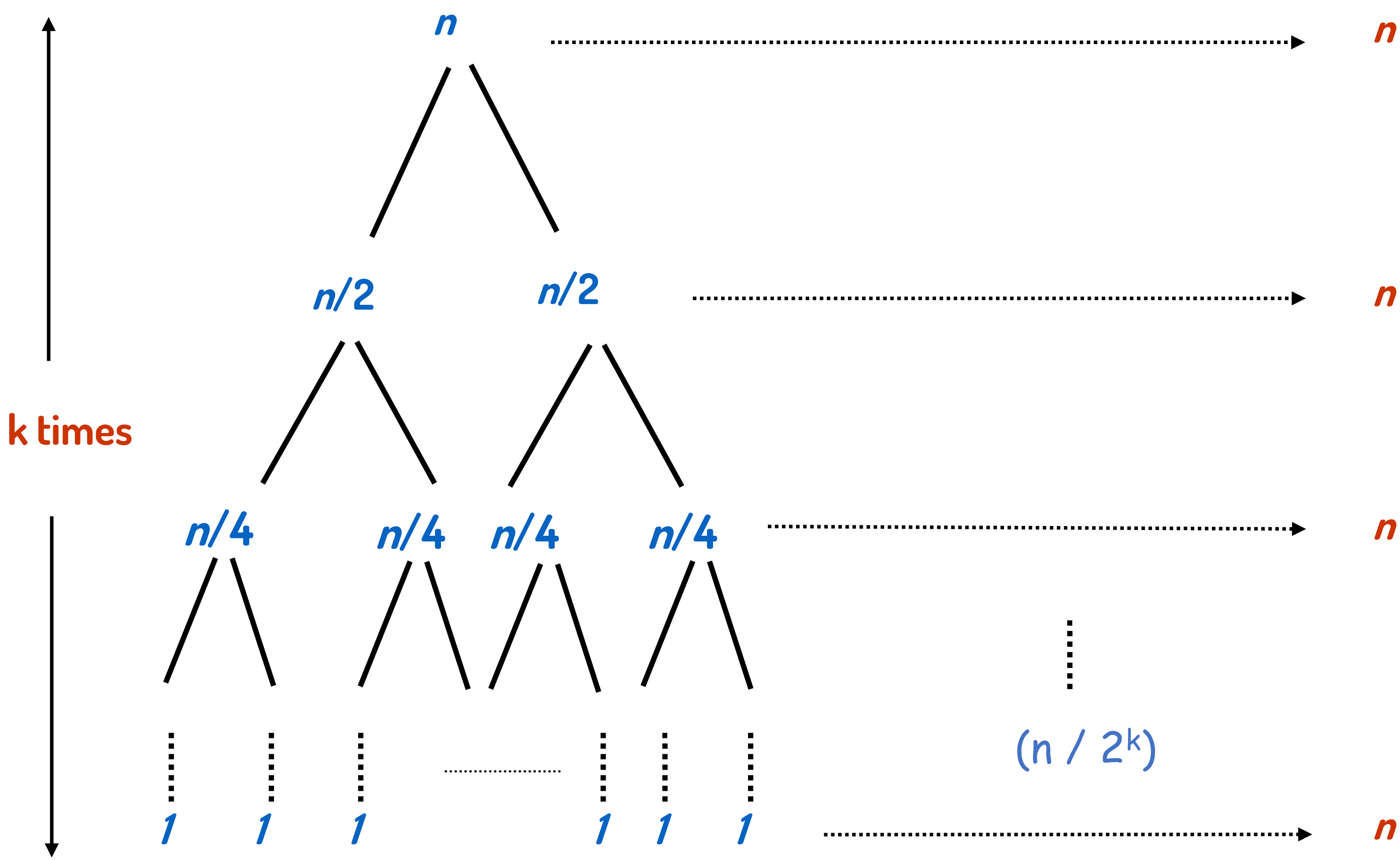
Merge sort recurrence.

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

Solution. $T(n) = O(n \log_2 n)$.

Proof by Recursion Tree

Continue expanding until the problem size reduces to 1.



Total time taken : nk

Proof by Recursion Tree Contd.

Merge sort recurrence.

$$T(n) = 1 \qquad \text{if } n = 1$$
$$T(n) = 2T(n/2) + n \qquad \text{if } n > 1$$

- So, the total time the algorithm takes = nk
- We, assume
    - $n / 2^k = 1$
    - $n = 2^k$
    - $\log_2 n = \log_2 2^k = k\log_2 2$
    - $k = \log_2 n$
- So, The time taken = nk = n $\log_2 n$

Proof by Induction

Claim. If T(n) satisfies this recurrence, then $T(n) = n \log_2 n$.

Pf. (by induction on n)

- Base case: n = 1.
- Inductive hypothesis: $T(n) = n \log_2 n$.
- Goal: show that $T(2n) = 2n \log_2 (2n)$.

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n\log_2 n + 2n \\ &= 2n\left(\log_2(2n) - 1\right) + 2n \\ &= 2n\log_2(2n) \end{aligned}$$

$$= 2n(log_2^2 + log_2^n - 1) + 2n$$

Be ready for Part-II
!!!!