Merge Sort

T#17

Merge Sort

- The key to Merge Sort is merging two sorted lists into one, such that if you have two lists X ($x_1 \leq x_2 \leq \cdots \leq x_m$) and Y($y_1 \leq y_2 \leq \cdots \leq y_n$) the resulting list is Z($z_1 \leq z_2 \leq \cdots \leq z_{m+n}$)

- Recursive in structure
  - ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - ***Combine*** the solutions to create a solution to the original problem

***<u>Sorting Problem</u>*: Sort a sequence of *n* elements into non-decreasing order.

- ***Divide***: Divide the *n*-element sequence to be sorted into two subsequences of *n/2* elements each
- ***Conquer:*** Sort the two subsequences recursively using merge sort.
- ***Combine***: Merge the two sorted subsequences to produce the sorted answer.

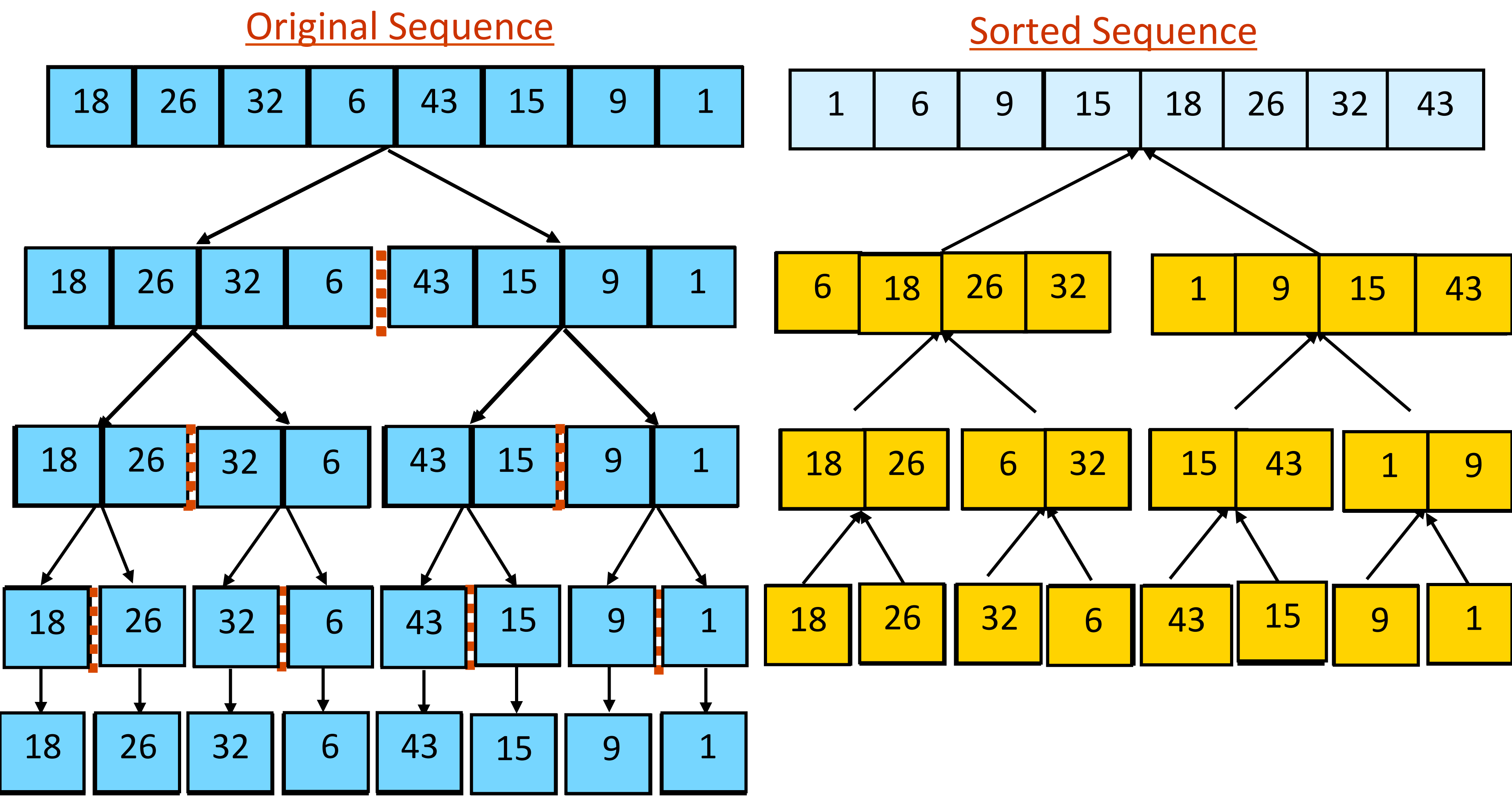
Merge Sort

```
void mergeSort(int [] A, int [] B)
  n = A.length
  if n == 1 then
    B[0] = A[0]
  else
    n1 = ⌈n/2⌉
    n2 = n − n1 {so that n2 = ⌊n/2⌋}
    Set A1 to be A[0], ..., A[n1 − 1] {length n1}
    Set A2 to be A[n1], ..., A[n − 1] {length n2}
    mergeSort(A1, B1)
    mergeSort(A2, B2)
    merge(B1, B2, B)
  end if
```

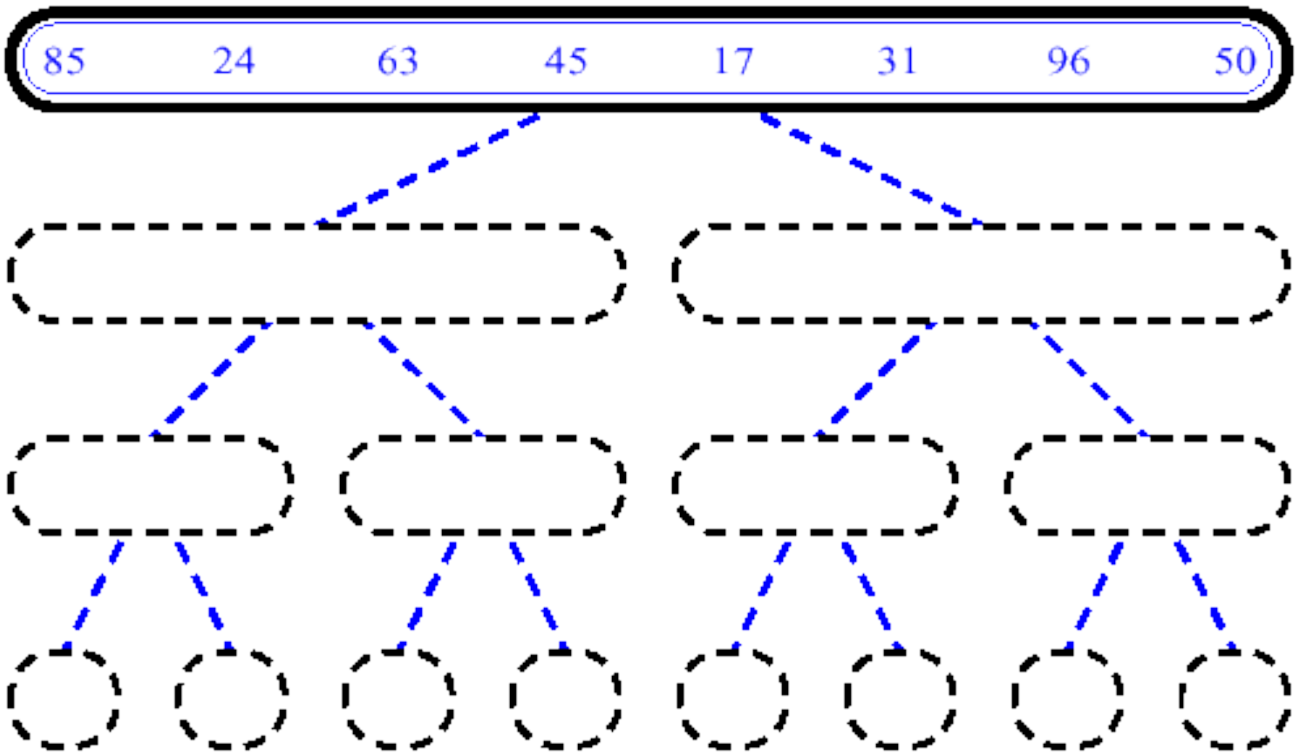
```
void merge(int [] A1, int [] A2, int [] B)
  n1 = length(A1); n2 = length(A2)
  Declare B to be an array of length n1 + n2
  i1 = 0; i2 = 0; j = 0
  while (i1 < n1) and (i2 < n2) do
    if A1[i1] ≤ A2[i2] then
      B[j] = A1[i1]; i1 = i1 + 1
    else
      B[j] = A2[i2]; i2 = i2 + 1
    end if
    j = j + 1
  end while
  {Copy remainder of A1 (if any)}
  while i1 < n1 do
    B[j] = A1[i1]; i1 = i1 + 1; j = j + 1
  end while

  {Otherwise copy remainder of A2}
  while i2 < n2 do
    B[j] = A2[i2]; i2 = i2 + 1; j = j + 1
  end while
```

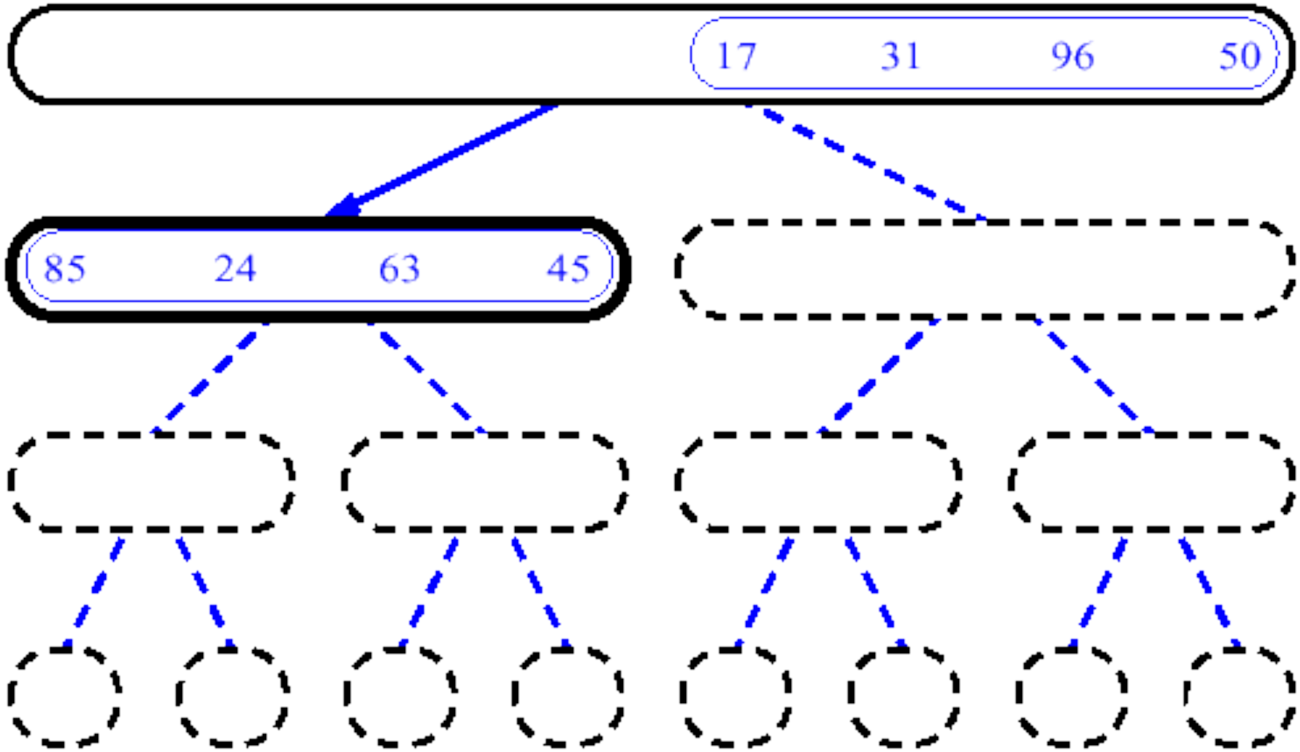
Merge Sort – Example



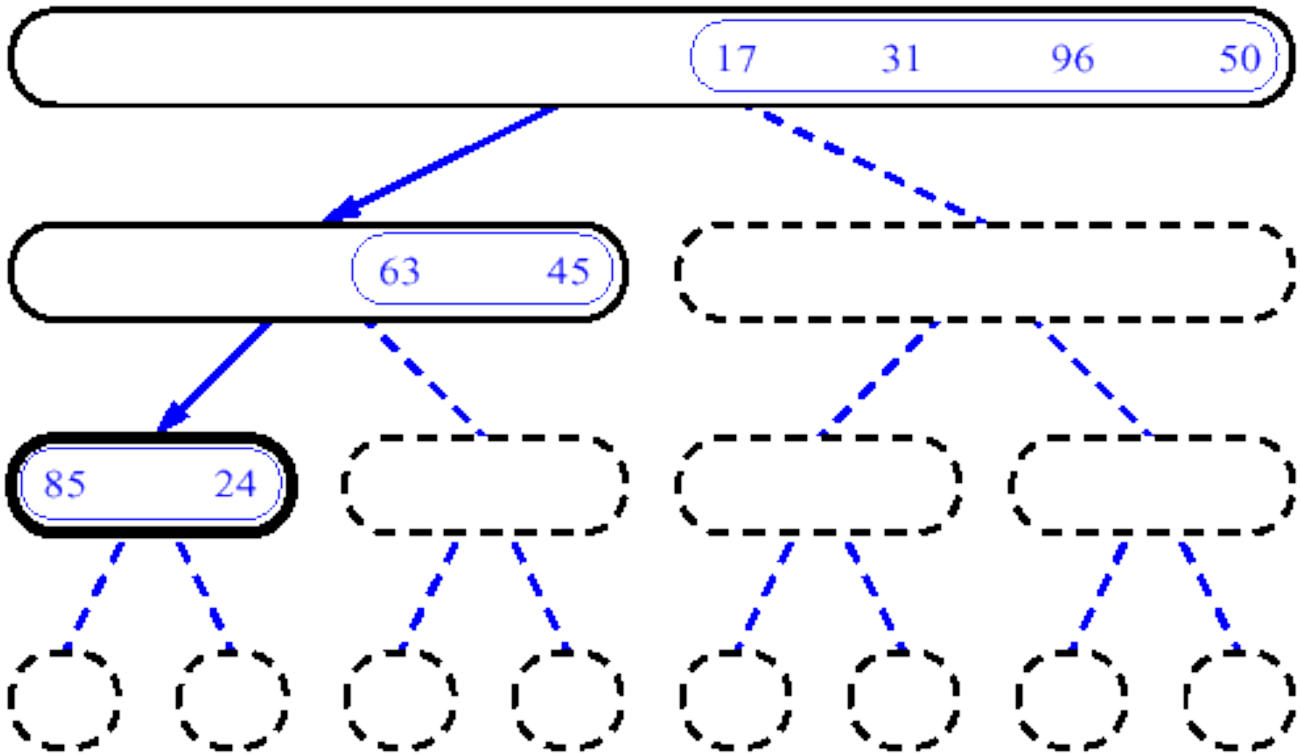
MergeSort (Example) - 1



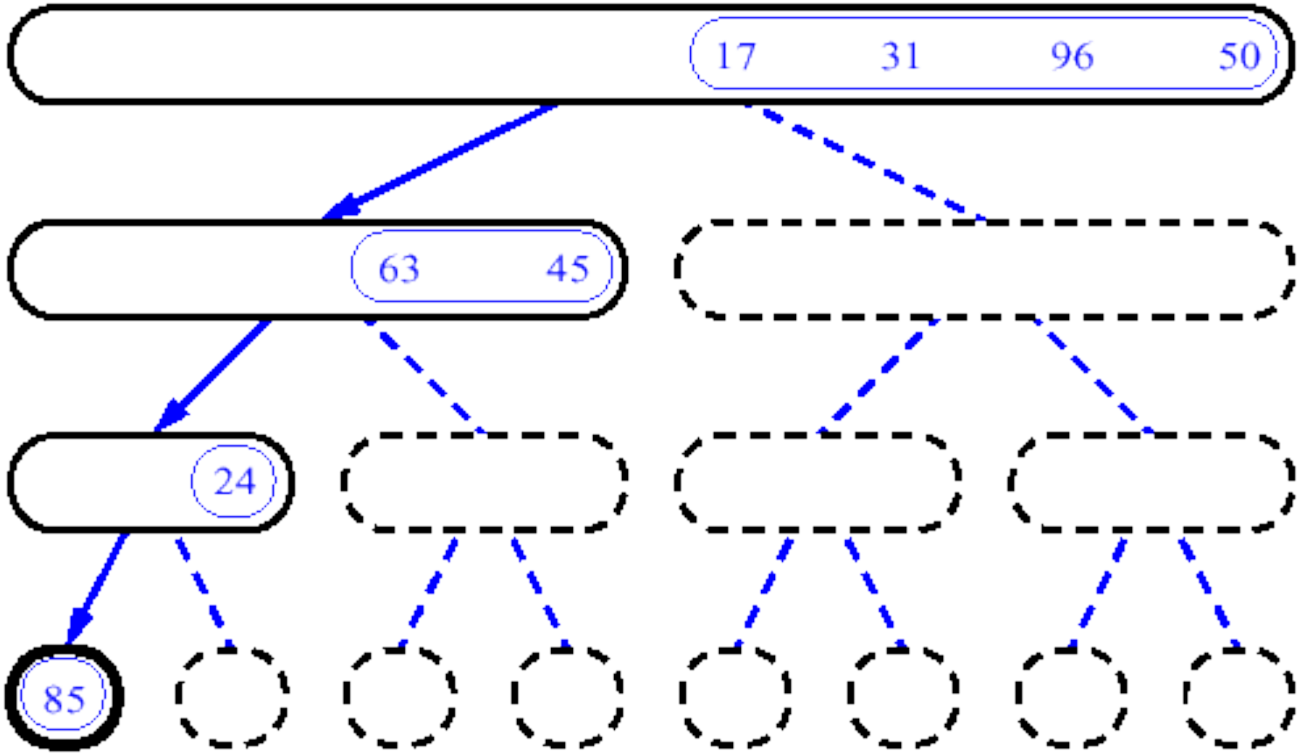
MergeSort (Example) - 2



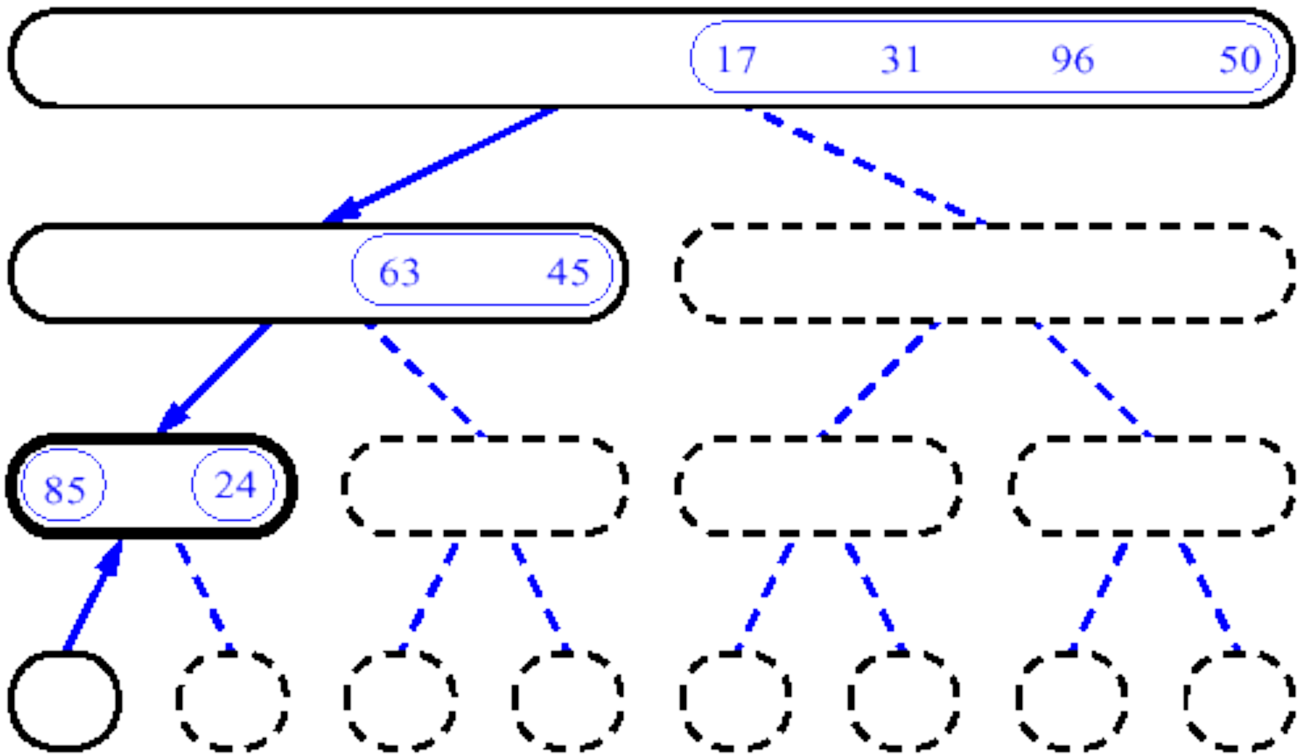
MergeSort (Example) - 3



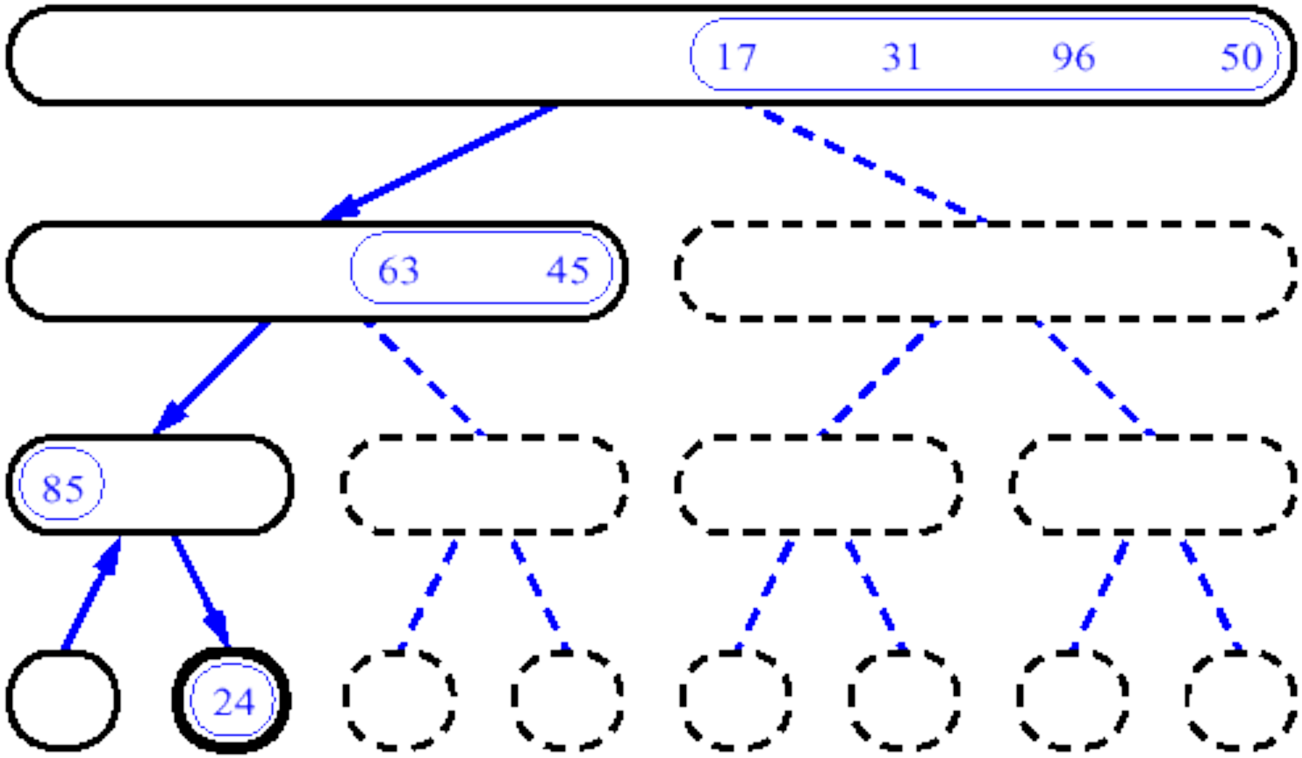
MergeSort (Example) - 4



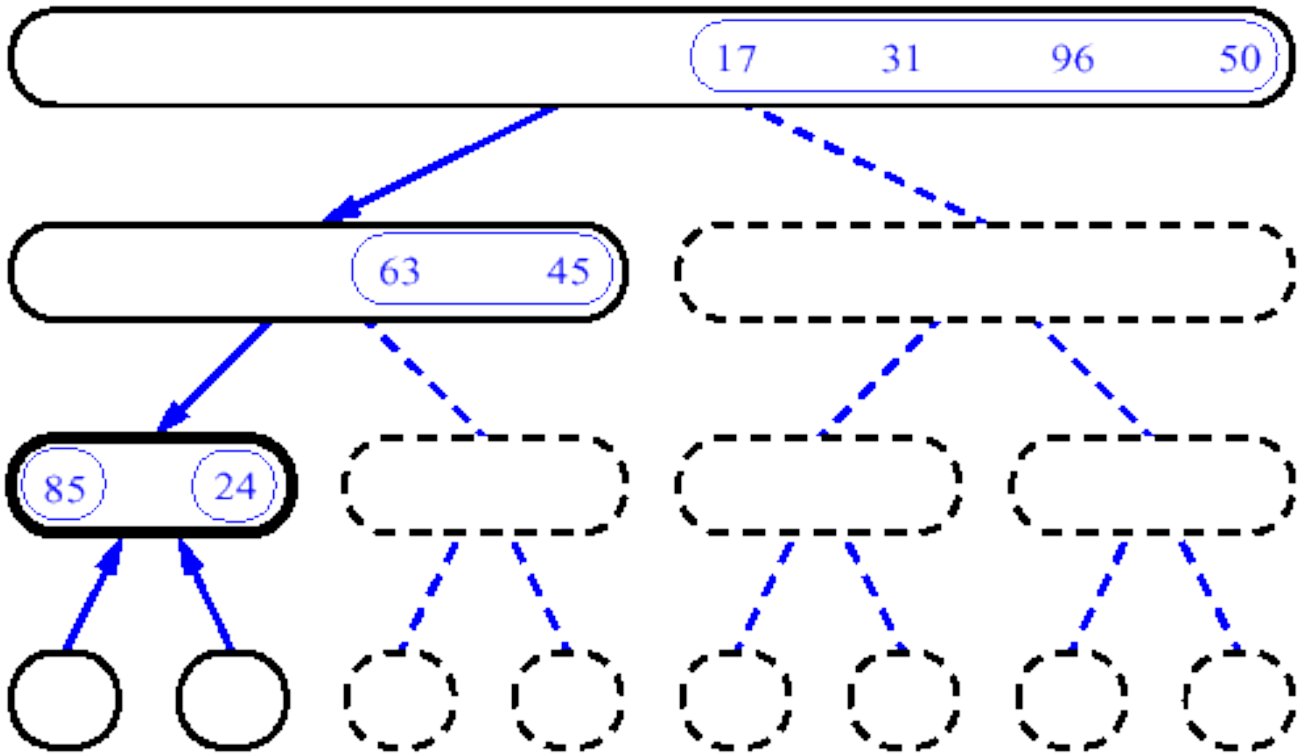
MergeSort (Example) - 5



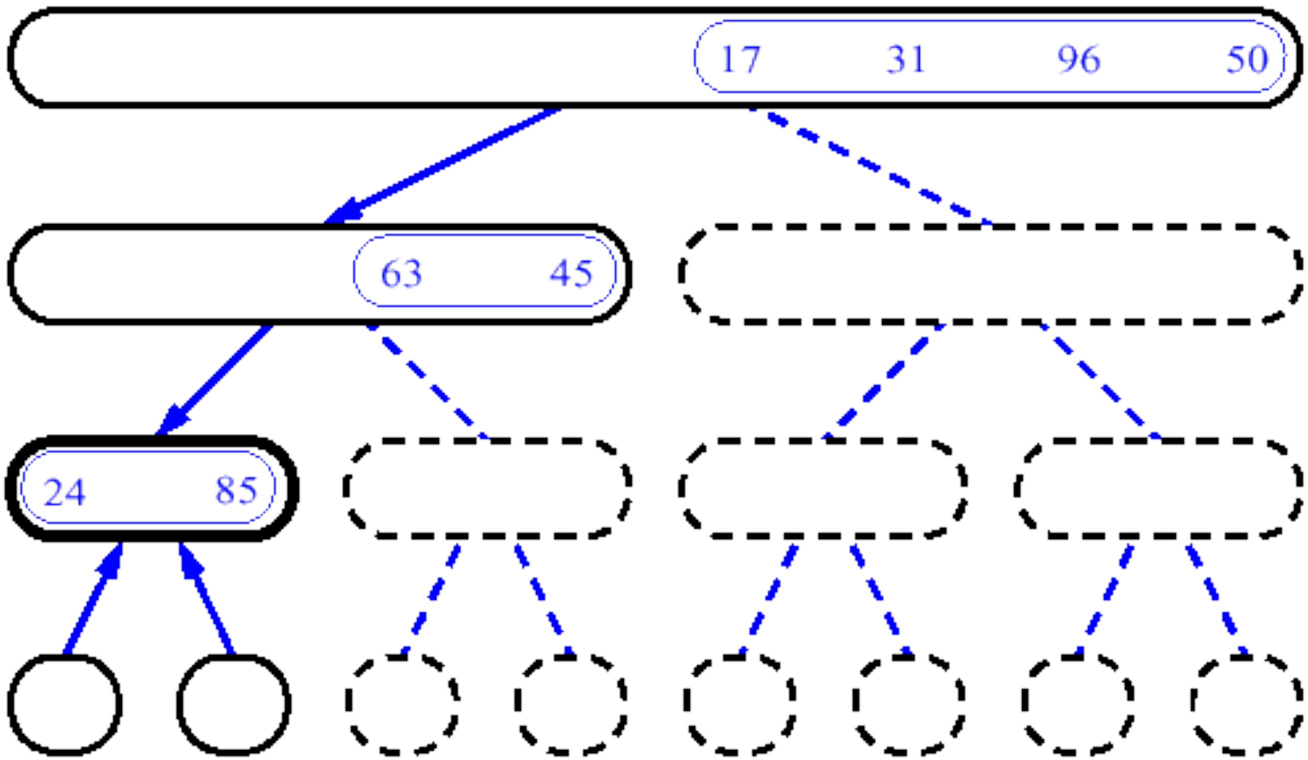
MergeSort (Example) - 6



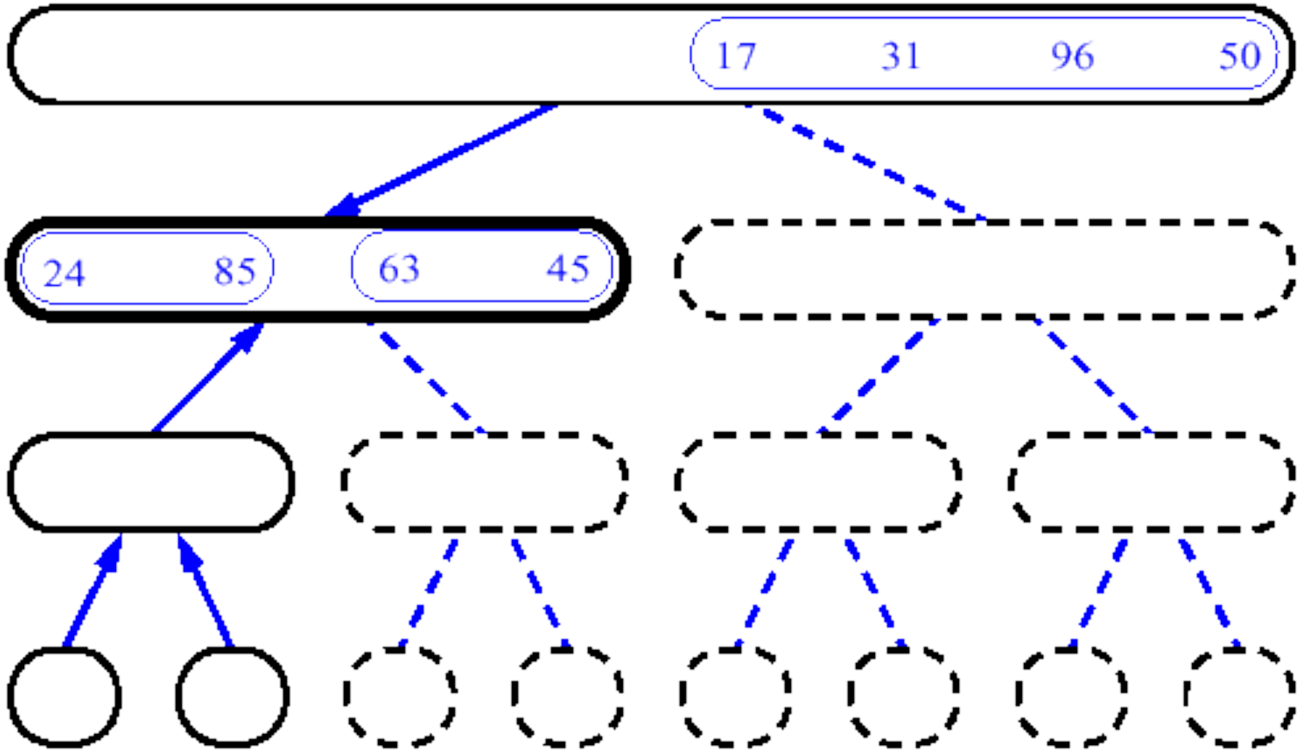
MergeSort (Example) - 7



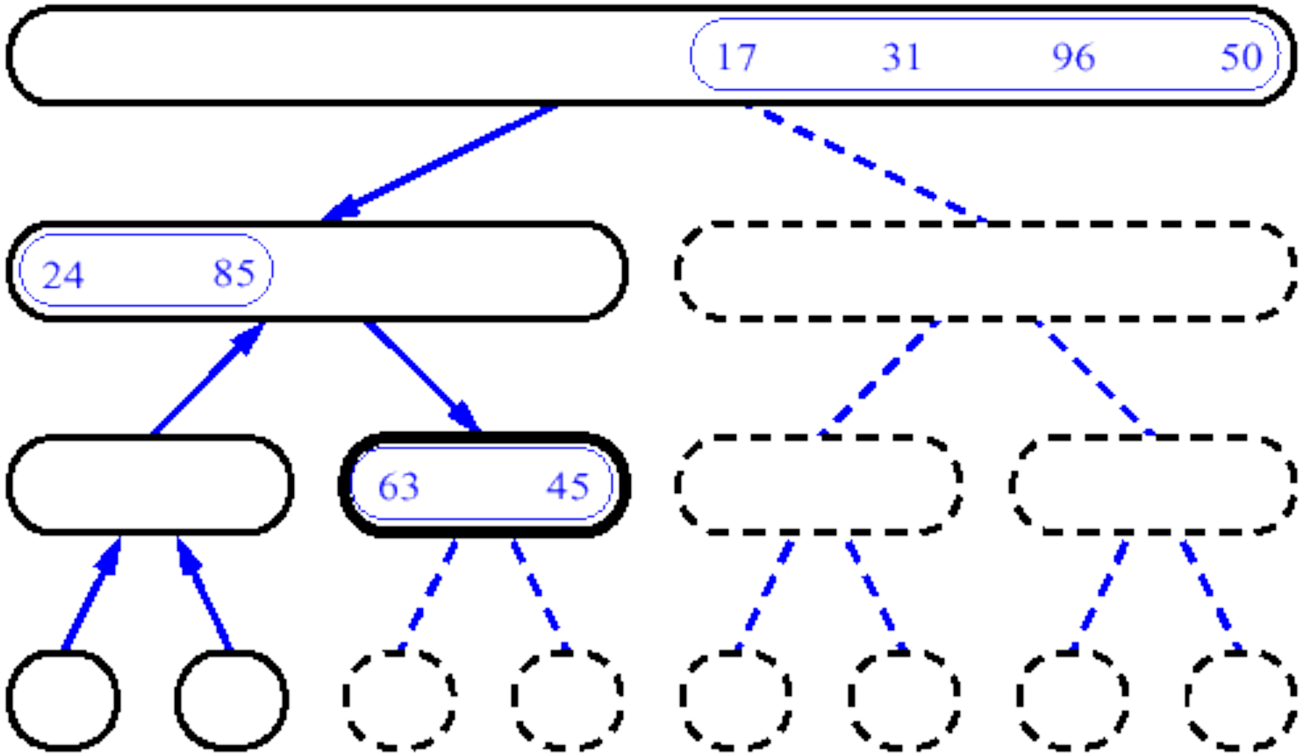
MergeSort (Example) - 8



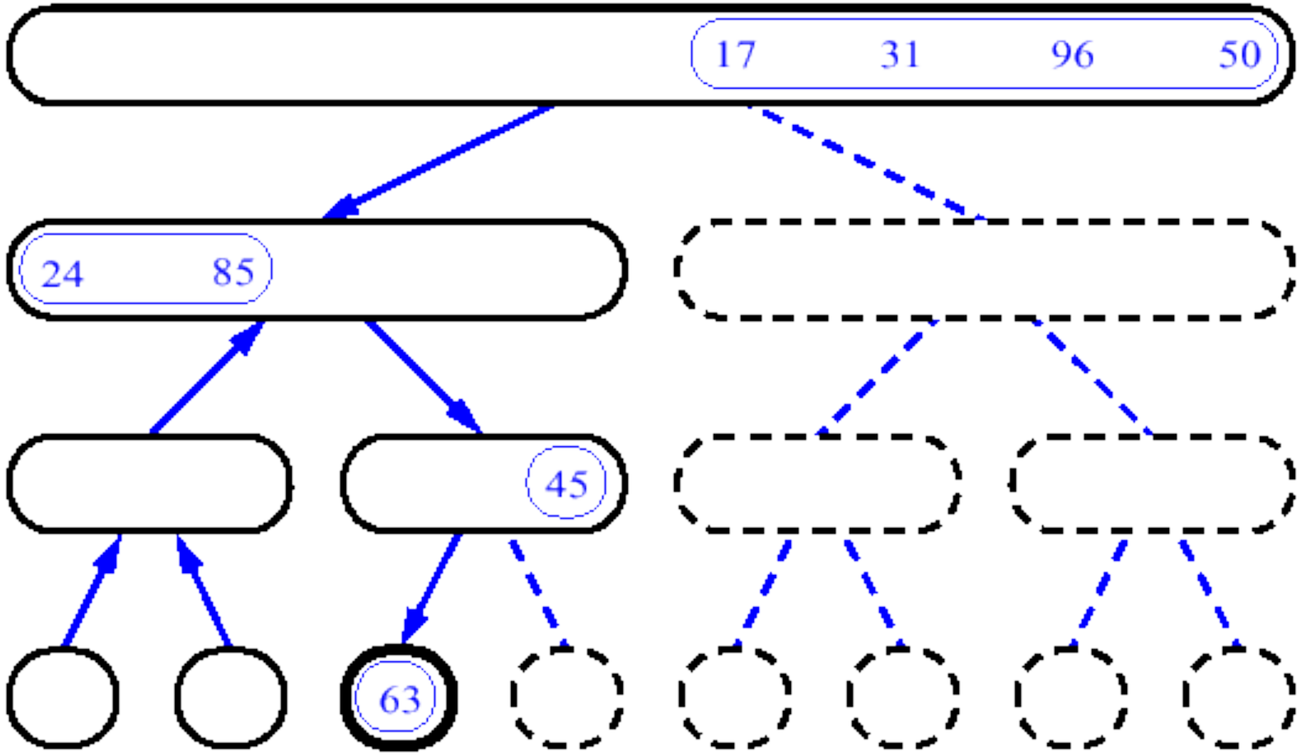
MergeSort (Example) - 9



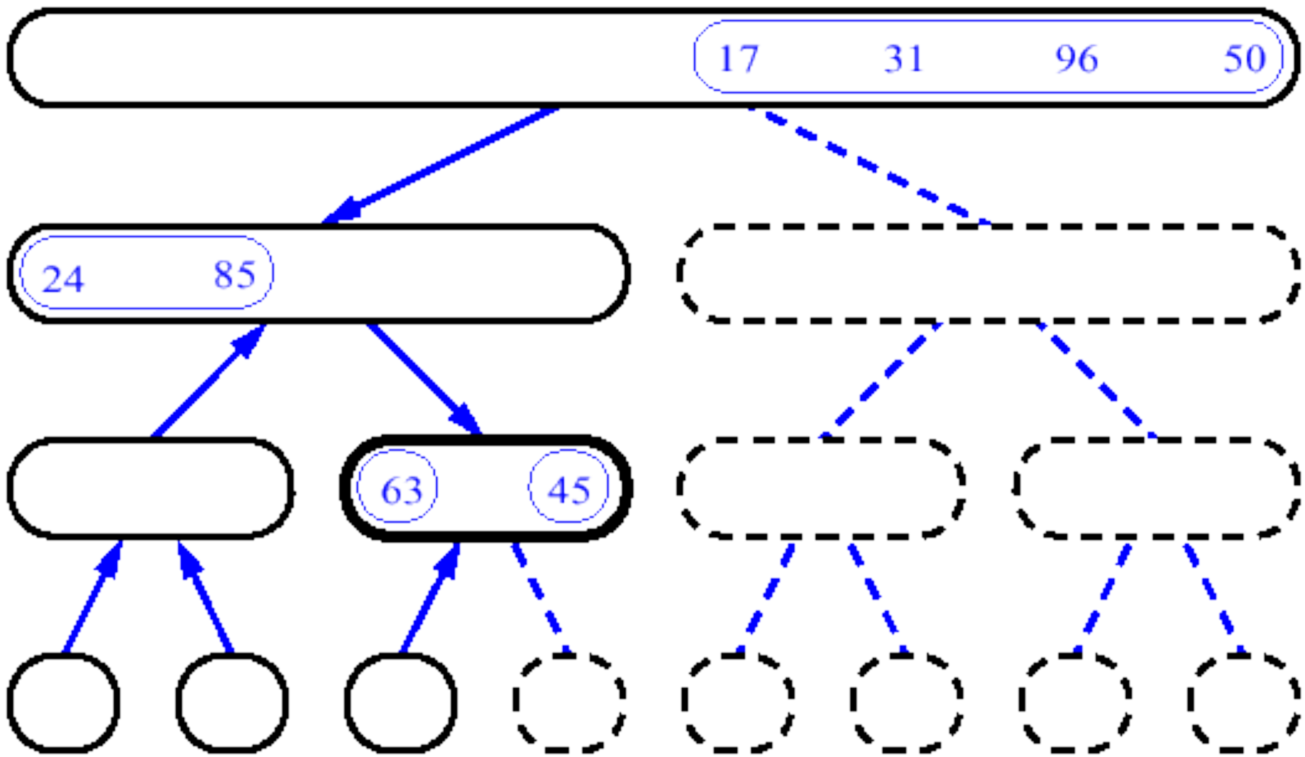
MergeSort (Example) - 10



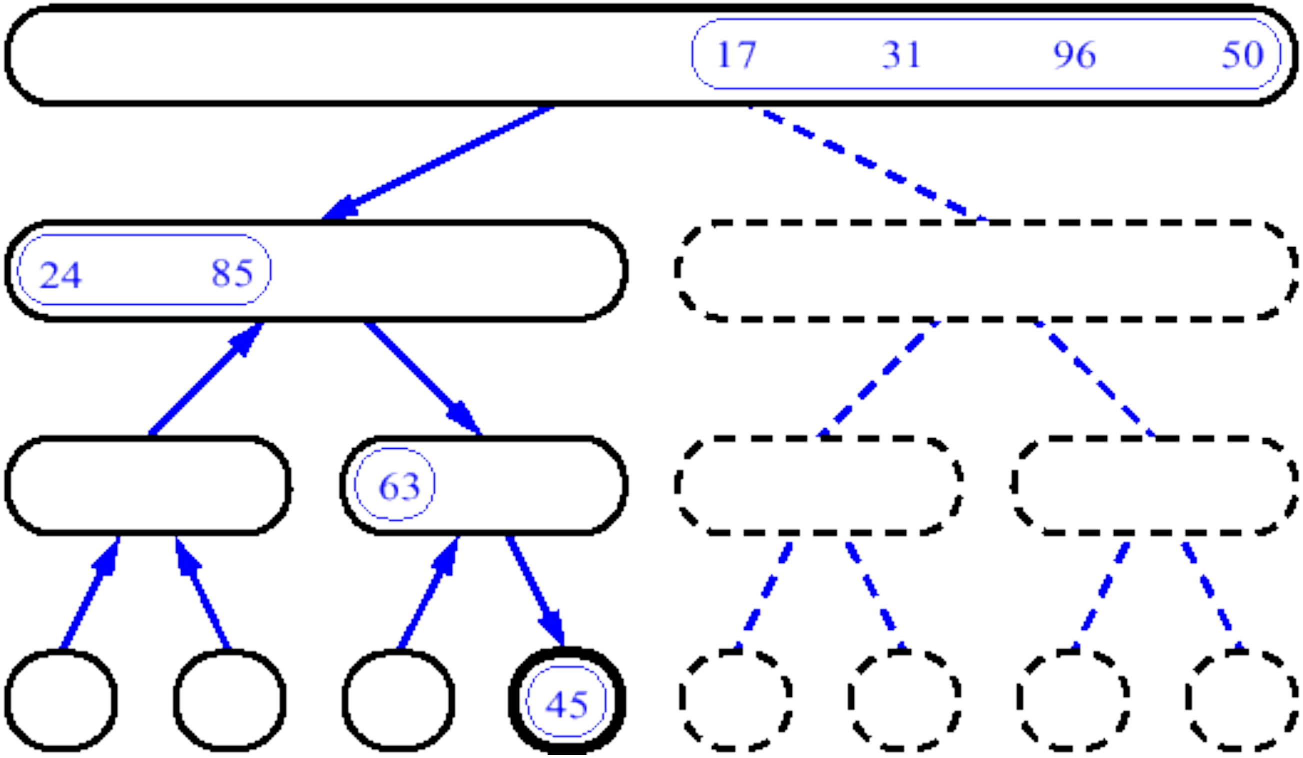
MergeSort (Example) - 11



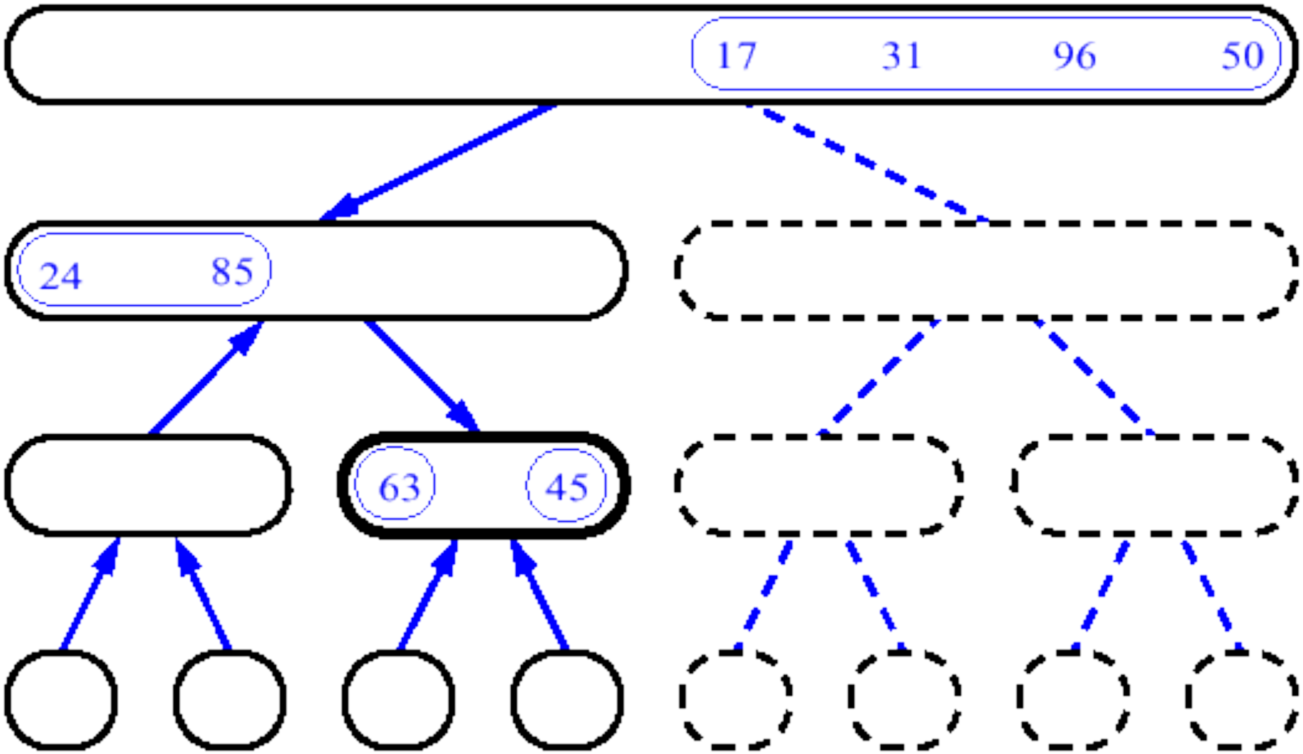
MergeSort (Example) - 12



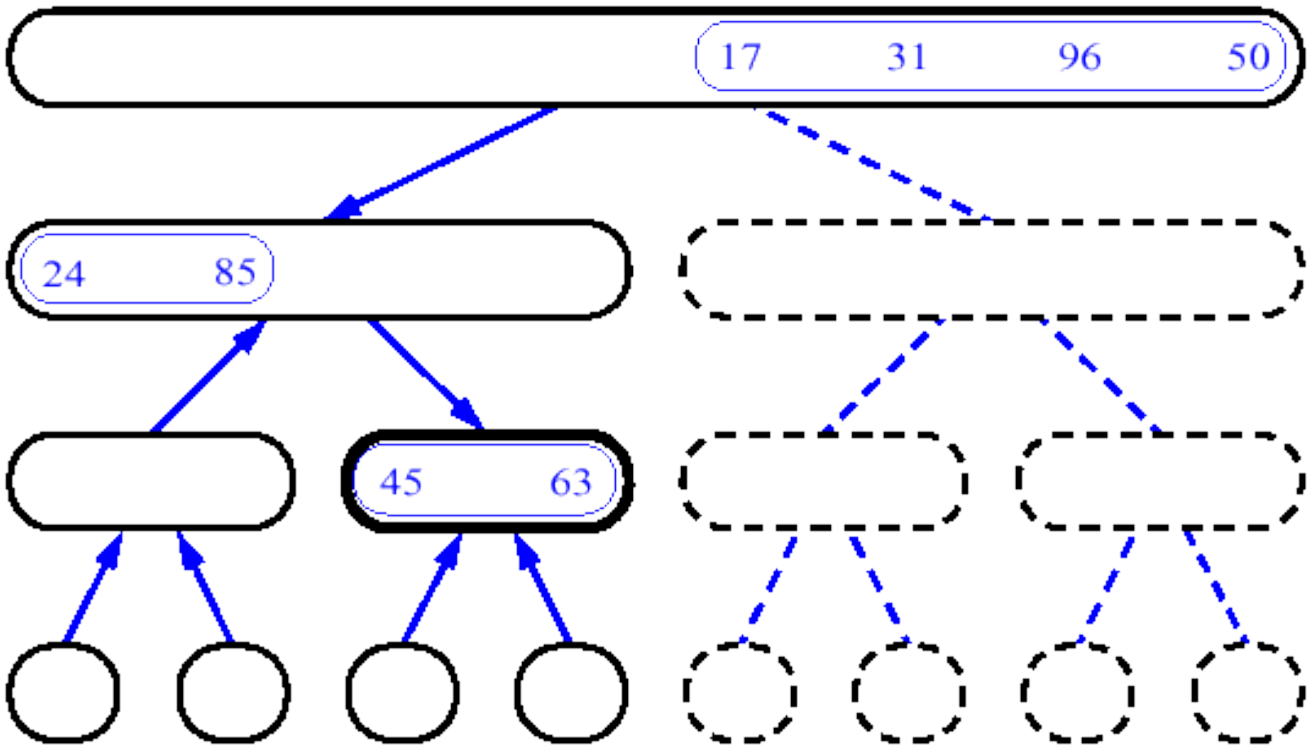
MergeSort (Example) - 13



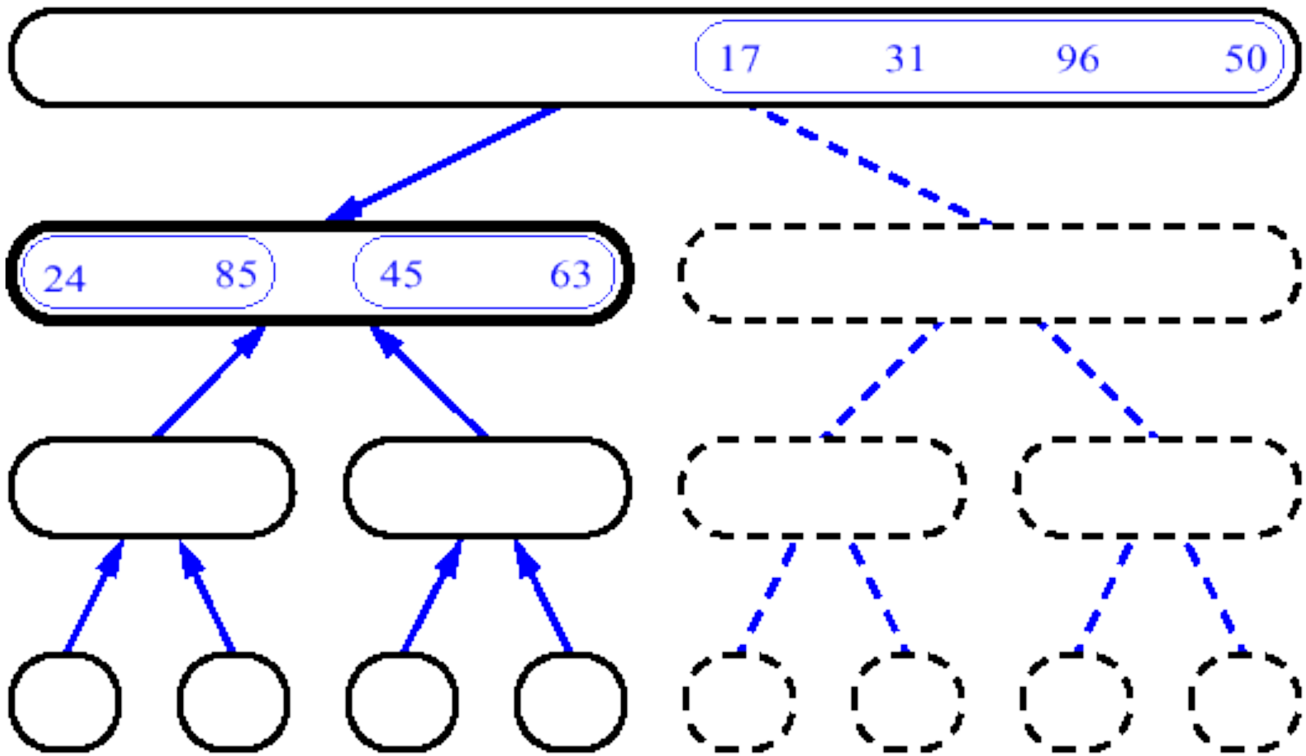
MergeSort (Example) - 14



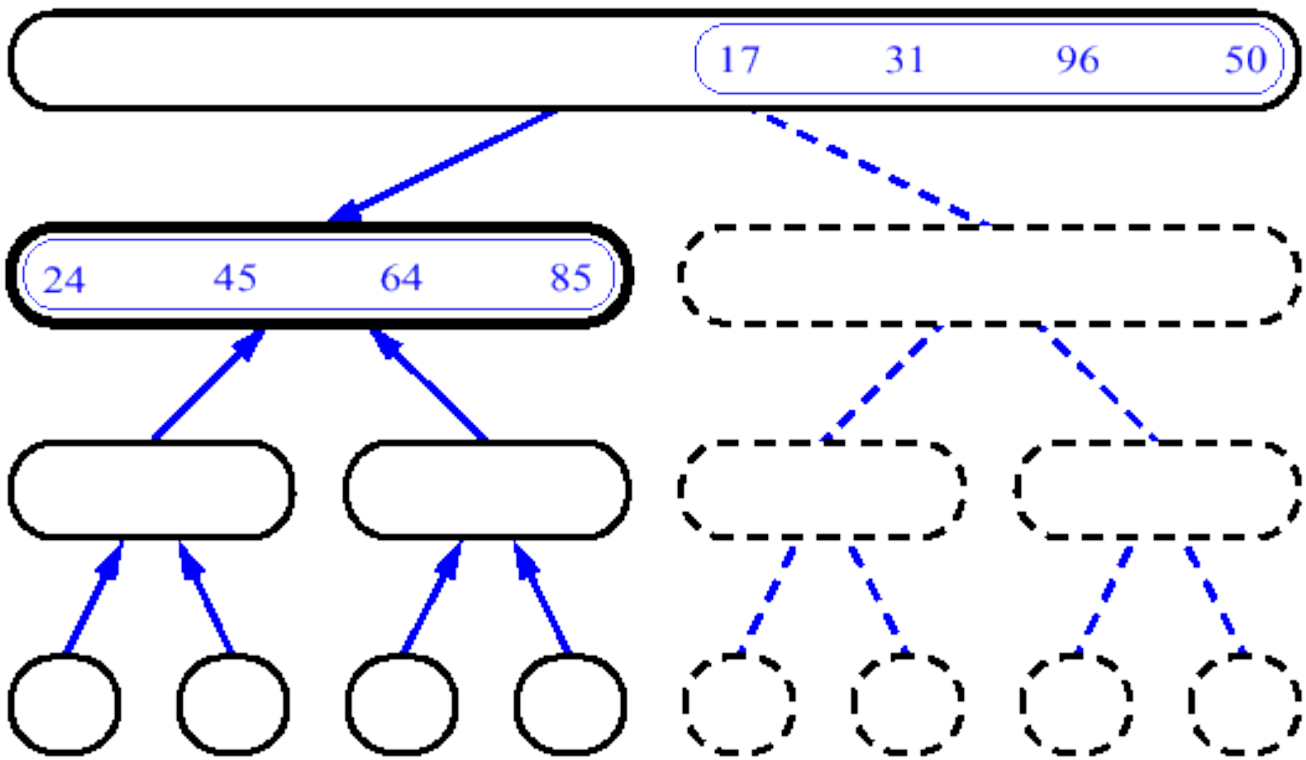
MergeSort (Example) - 15



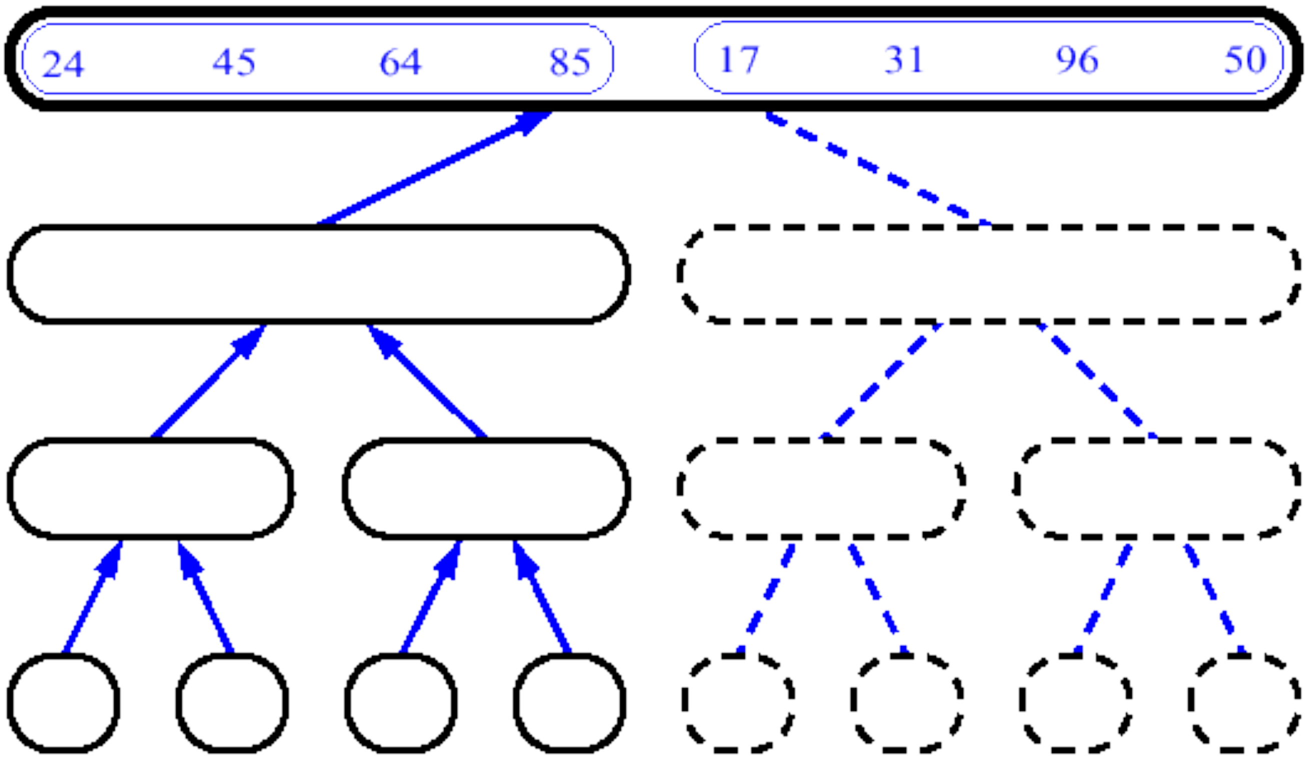
MergeSort (Example) - 16



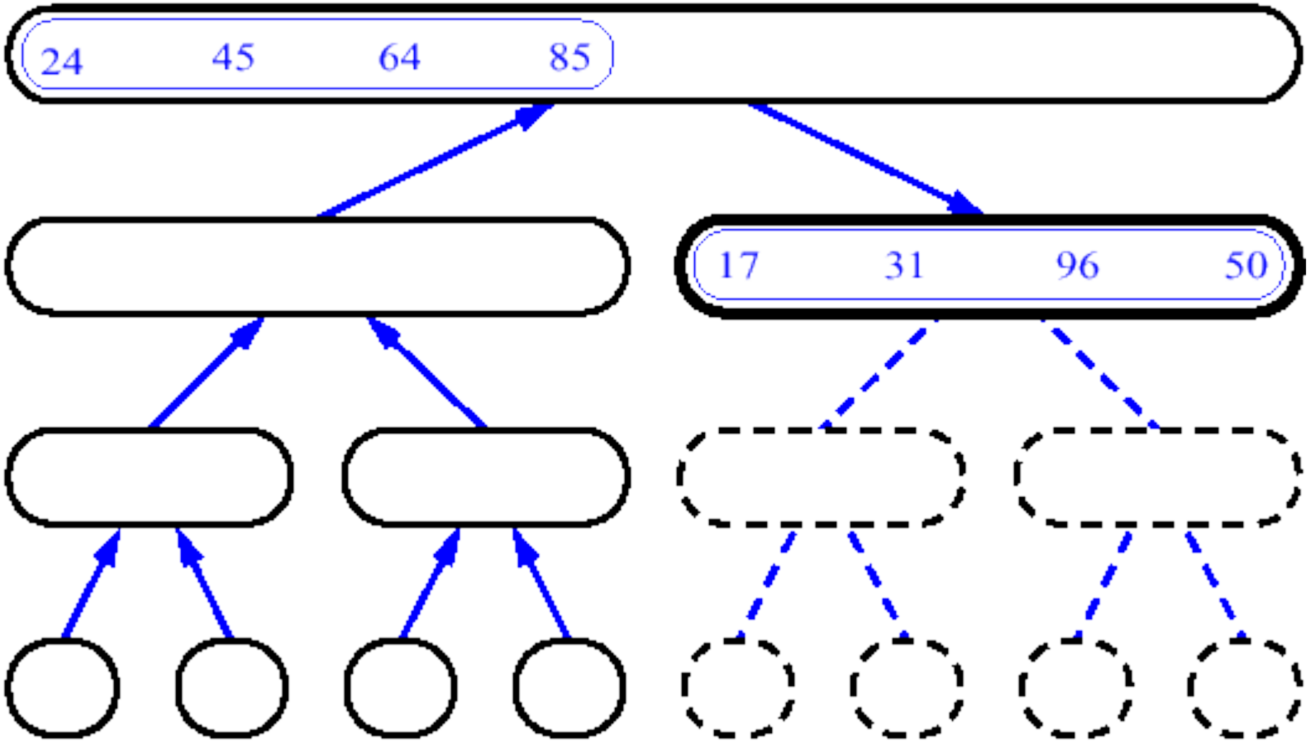
MergeSort (Example) - 17



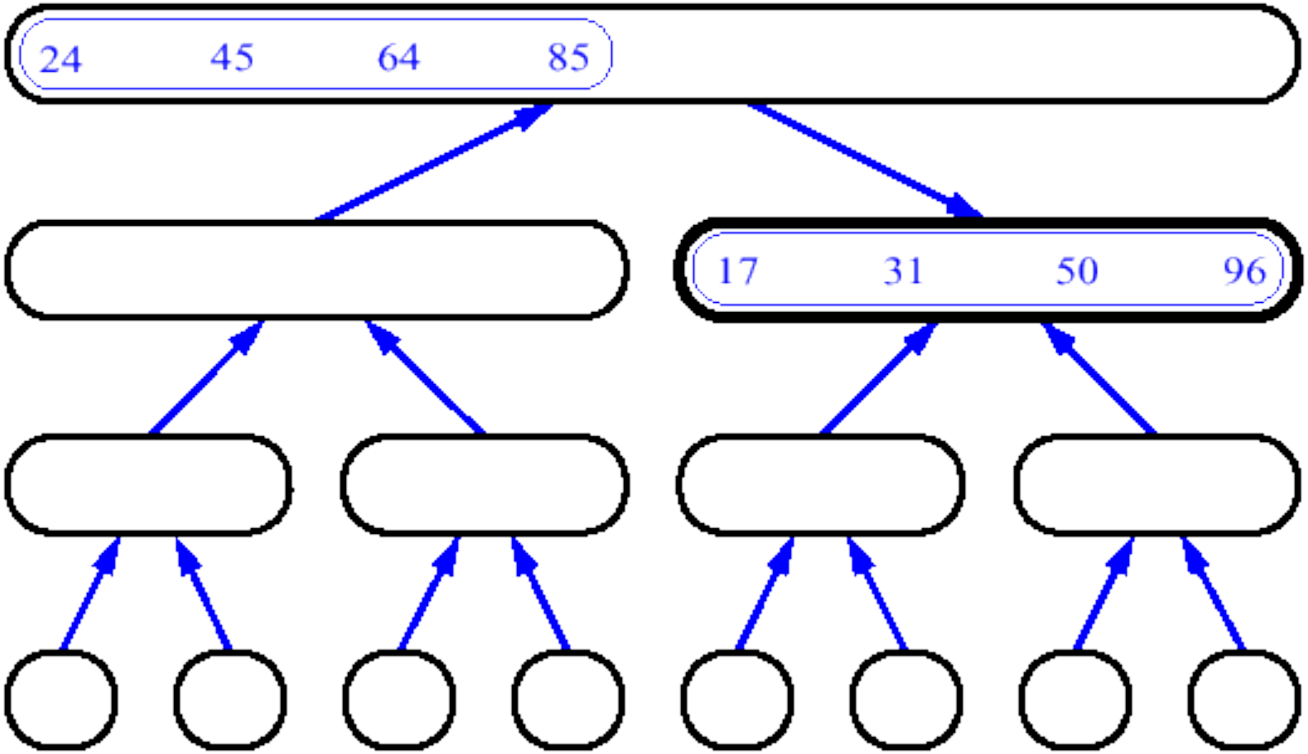
MergeSort (Example) - 18



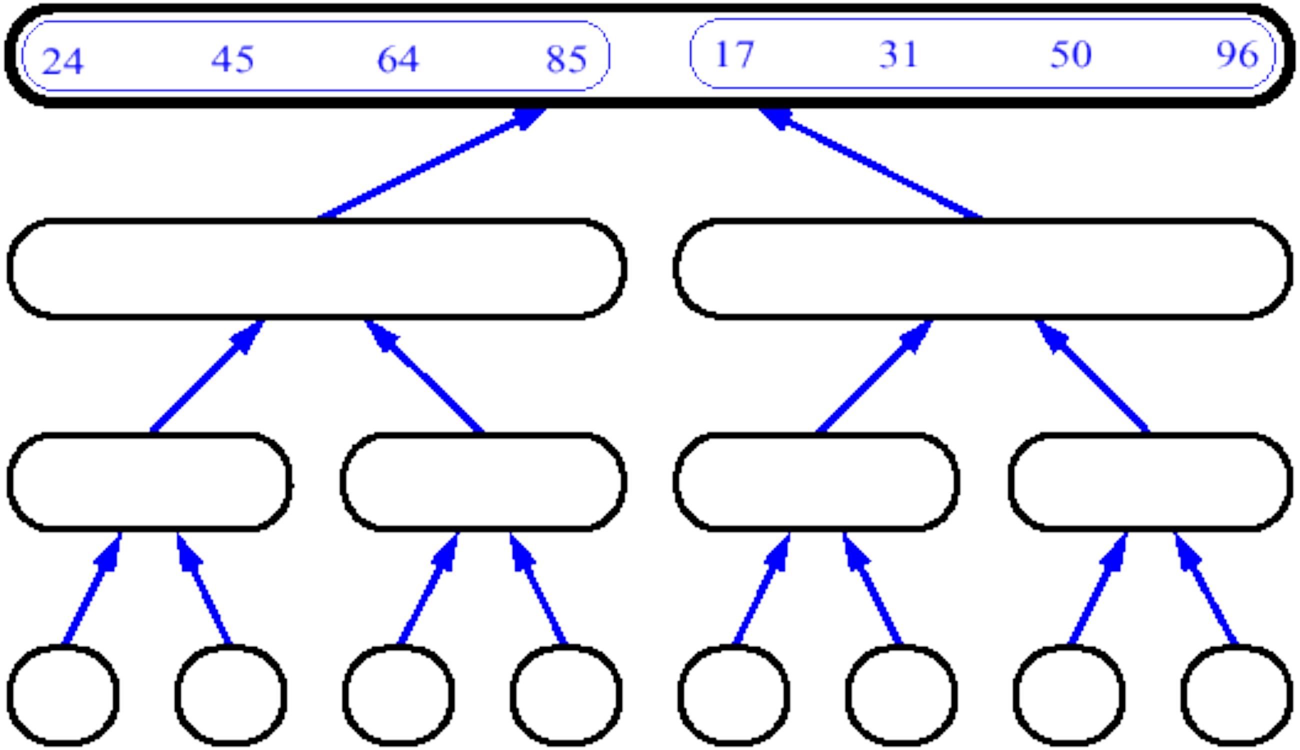
MergeSort (Example) - 19



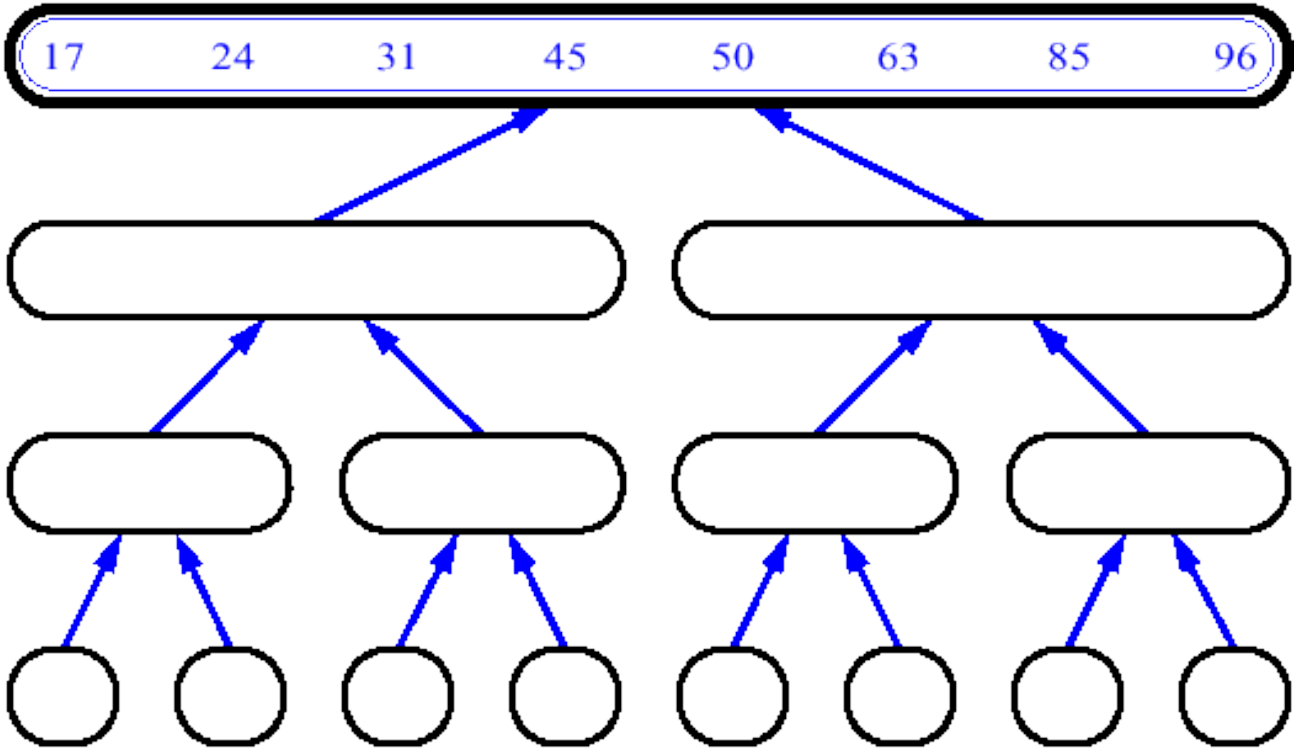
MergeSort (Example) - 20



MergeSort (Example) - 21



MergeSort (Example) - 22



Merge Sort Complexity

- Mergesort always partitions the array equally.

Thus, the recursive depth is always $\mathrm{O}(\log n)$

- The amount of work done at each level is $\mathrm{O}(n)$

Intuitively, the complexity should be $\mathrm{O}(n \lg n)$

We have,

– $T(n) = 2T(n/2) + c*n$ *for* $n>1$, $T(1)=0 \Rightarrow \Theta(n \lg n)$

Merge Sort Physical Complexity

- Work on it for 5 minutes!

Merge Sort Space Complexity

- The **Mergesort** algorithm is recursive, so it requires O(log n) stack **space** But the array case also allocates an additional O(n) **space**, which dominates the O(log n) **space** required for the stack. So the array version space complexity is O(n).

Parallelizing

- - in a perfect world you'd have to do log n merges of size n, n/2, n/4 ... (or better said 1, 2, 3 ... n/4, n/2, n - they can't be parallelized), which gives O(n). It still is O(n log n).