
cmpt-225

Sorting

Sorting

- Fundamental problem in computing science
 - putting a collection of items in order
 - Often used as part of another algorithm
 - e.g. sort a list, then do many binary searches
 - e.g. looking for identical items in an array:
1, 5, 3, 1, 4, 3, 2, 1, 4, 5
 - unsorted: do $O(n^2)$ comparisons
 - 1, 1, 1, 2, 3, 3, 4, 4, 5, 5
 - sort $O(??)$, then do $O(n)$ comparisons
-

Sorting Example

12, 2, 23, -3, 21, 14

Easy....

but think about a systematic approach....

Sorting Example

4, 3, 5435, 23, -324, 432, 23, 22, 29,
11, 31, 21, 21, 17, -5, -79, -19, 312,
213, 432, 321, 11, 1243, 12, 15, 1, -1,
214, 342, 76, 78, 765, 756, -465, -2, 453,
534, 45265, 65, 23, 89, 87684, 2, 234,
6657, 7, 65, -42, 432, 876, 97, 0, -11, -65,
-87, 645, 74, 645

How well does your intuition generalize to big examples?

The General Sorting Problem

Given:

- A sequence.
 - Items are all of the same type.
 - There are no other restrictions on the number or values of items in the sequence.
- A comparison function.
 - Given two sequence items, determine which is first.
 - This function is the **only** way we can compare.

Return:

- A sorted sequence with the same items as original.
-

Sorting

- There are many algorithms for sorting
 - Each has different properties:
 - easy/hard to understand
 - fast/slow for large lists
 - fast/slow for short lists
 - fast in all cases/on average
-

Selection Sort

- Find the smallest item in the list
 - Switch it with the first position
 - Find the next smallest item
 - Switch it with the second position
 - Repeat until you reach the last element
-

Selection Sort: Example

Original list:

17	8	75	23	14
----	---	----	----	----

Smallest is 17:

8	14	17	23	75
---	----	----	----	----

Smallest is 8:

8	17	75	23	14
---	----	----	----	----

Smallest is 23:

8	14	17	23	75
---	----	----	----	----

Smallest is 14:

8	14	75	23	17
---	----	----	----	----

DONE!

Selection Search: Running Time

- Scan entire list (n steps)
- Scan rest of the list ($n-1$ steps)....
- Total steps:

$$n + (n - 1) + (n - 2) + \dots + 1$$

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

$$= n^2/2 + n/2$$

- So, selection sort is $O(n^2)$
-

Selection Sort in Java

```
public void selectionSort (int[] arr)
{
    int i,j,min,temp;
    for(j=0; j < arr.length-1; j++)
    {
        //find the smallest from j to arr.length-1
        min=j;
        for (i=j+1; i < arr.length; i++){
            if (arr[i] < arr[min])
                min=i;
        }
        //replace the smallest with the jth element.
        temp=arr[j];
        arr[j]=arr[min];
        arr[min]=temp;
    }
}
```

More precise analysis of Selection Sort

```
public void selectionSort (int[] arr)
{
    int i,j,min,temp;
    for(j=0; j < arr.length-1; j++)
    {
        // outer for loop is evaluated n-1 times
        min=j; //n-1 times
        for (i=j+1; i < arr.length; i++){// n(n-1)/2 evaluations
            if (arr[i] < arr[min]) // n(n-1)/2 comparisons
                min=i; //(*)n(n-1)/2 worst case, 0 best case
        }
        //replace the smallest with the jth element.
        temp=arr[j];    //n-1 times
        arr[j]=arr[min]; //n-1 times
        arr[min]=temp;  //n-1 times
    }
}
```

Selection Sort: Cost Function

- There is 1 operation needed to initializing the outer loop
 - The outer loop is evaluated $n-1$ times
 - 7 instructions (these include the outer loop comparison and increment, and the initialization of the inner loop)
 - Cost is $7(n-1)$
 - The inner loop is evaluated $n(n-1)/2$ times
 - There are 4 instructions in the inner loop, but one (*) is only evaluated sometimes
 - Worst case cost upper bound: $4(n(n-1)/2)$
 - Total cost: $1 + 7(n-1) + 4(n(n-1)/2)$ [worst case]
 - Assumption: that all instructions have the same cost
-

Selection Sort: Summary

- Number of comparisons:
 $n(n-1)/2$
 - The best case time cost:
 $1 + 7(n-1) + 3(n(n-1)/2)$ (*array was sorted*)
 - The worst case time cost (an upper bound):
 $1 + 7(n-1) + 4(n(n-1)/2)$
 - The number of swaps:
 $n-1$ [number of moves: $3(n-1)$]
-

Bubble Sort

- Bubble sort
 - Strategy
 - Compare adjacent elements and exchange them if they are out of order
 - Comparing the first two elements, the second and third elements, and so on, will move the largest (or smallest) elements to the end of the array
 - Repeating this process will eventually sort the array into ascending (or descending) order
-

Bubble Sort

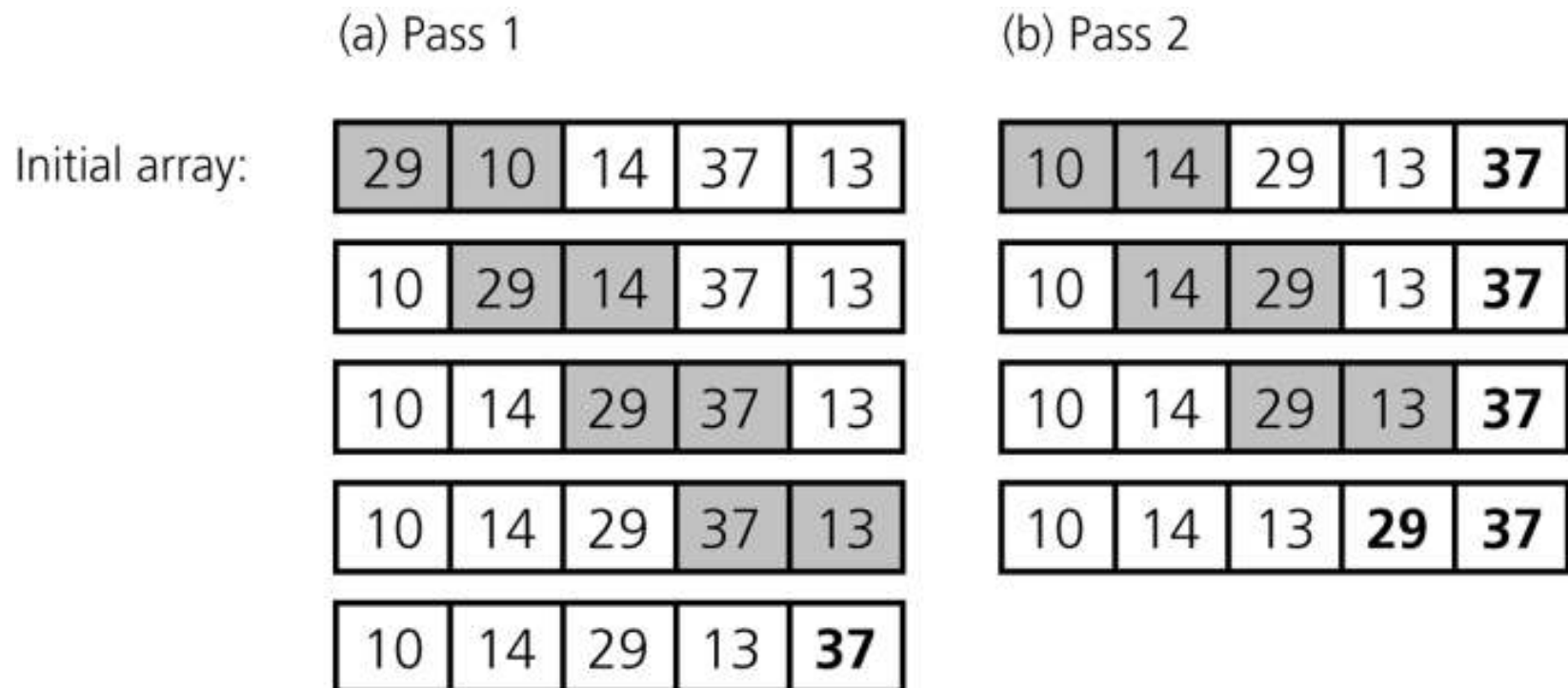
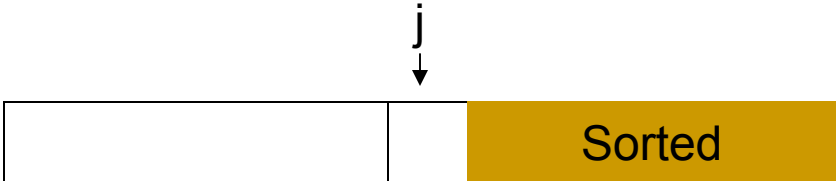


Figure 10-5

The first two passes of a bubble sort of an array of five integers: a) pass 1;
b) pass 2

Bubble Sort

```
public void bubbleSort (Comparable[] arr) {  
    for (int j = arr.length-1; j>0; j--){  
        for (int i = 0; i<j; i++){  
            if (arr[i].compareTo(arr[i+1]) > 0) {  
                Comparable tmp = arr[i];  
                arr[i] = arr[i+1];  
                arr[i+1] = tmp;  
            }  
        }  
    }  
}
```



The diagram illustrates the state of an array during the bubble sort process. The array is represented as a horizontal bar divided into three sections. The first two sections are white, and the third section is yellow and labeled "Sorted". An arrow labeled "j" points down to the boundary between the second and third sections, indicating the current position of the outer loop's index.

First round

17	12	10	20	19
12	17	10	20	19
12	10	17	20	19
12	10	17	20	19
12	10	17	19	20

Second round

12	10	17	19	20
10	12	17	19	20
10	12	17	19	20
10	12	17	19	20
10	12	17	19	20

Third round

10	12	17	19	20
10	12	17	19	20
10	12	17	19	20

Forth round

10	12	17	19	20
10	12	17	19	20

☐ After the second round the list is already sorted but the algorithm continues to work

☐ A more efficient implementations stops when the list is sorted.

Bubble Sort

```
public void bubbleSort (Comparable[] arr) {  
    boolean isSorted = false;  
    for (int j = arr.length-1; !isSorted && j>0; j--){  
        isSorted = true;  
        for (int i = 0; i<j; i++){  
            if (arr[i].compareTo(arr[i+1]) > 0) {  
                isSorted = false;  
                Comparable tmp = arr[i];  
                arr[i] = arr[i+1];  
                arr[i+1] = tmp;  
            }  
        }  
    }  
}
```

Bubble Sort

- Analysis
 - Worst case: $O(n^2)$
 - Best case: $O(n)$ //the list is already sorted.
 - Beyond Big-O: bubble sort generally performs worse than the other $O(n^2)$ sorts
 - ... you generally don't see bubble sort outside a university classroom
-

Insertion Sort.

- First we consider a version that uses an extra array.
 - Start with an empty auxiliary array and insert each elements of the input array in the proper position in the auxiliary array.
 - Return the auxiliary array.
-

Example

Original

10	8	9	13	2
----	---	---	----	---

Sorted

10				
----	--	--	--	--

Example

Original	<table><tr><td>10</td><td>8</td><td>9</td><td>13</td><td>2</td></tr></table>	10	8	9	13	2	<table><tr><td>10</td><td>8</td><td>9</td><td>13</td><td>2</td></tr></table>	10	8	9	13	2
10	8	9	13	2								
10	8	9	13	2								
Sorted	<table><tr><td>10</td><td></td><td></td><td></td><td></td></tr></table>	10					<table><tr><td>8</td><td>10</td><td></td><td></td><td></td></tr></table>	8	10			
10												
8	10											

Example

Original	<table><tr><td>10</td><td>8</td><td>9</td><td>13</td><td>2</td></tr></table>	10	8	9	13	2	<table><tr><td>10</td><td>8</td><td>9</td><td>13</td><td>2</td></tr></table>	10	8	9	13	2	<table><tr><td>10</td><td>8</td><td>9</td><td>13</td><td>2</td></tr></table>	10	8	9	13	2
10	8	9	13	2														
10	8	9	13	2														
10	8	9	13	2														
Sorted	<table><tr><td>10</td><td></td><td></td><td></td><td></td></tr></table>	10					<table><tr><td>8</td><td>10</td><td></td><td></td><td></td></tr></table>	8	10				<table><tr><td>8</td><td>9</td><td>10</td><td></td><td></td></tr></table>	8	9	10		
10																		
8	10																	
8	9	10																

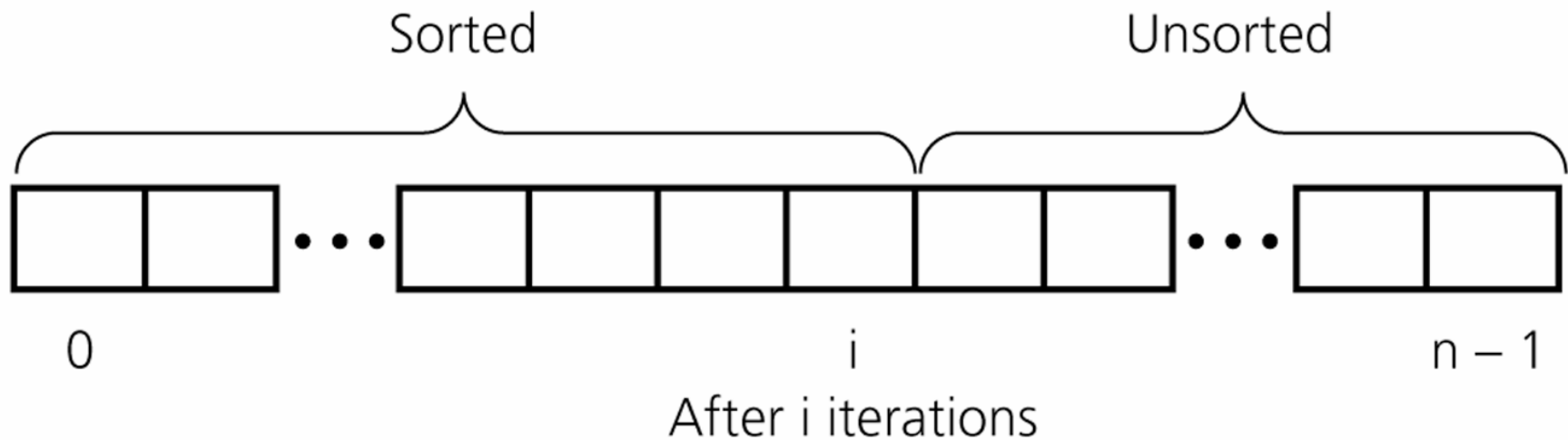
Example

Original	<table><tr><td>10</td><td>8</td><td>9</td><td>13</td><td>2</td></tr></table>	10	8	9	13	2	<table><tr><td>10</td><td>8</td><td>9</td><td>13</td><td>2</td></tr></table>	10	8	9	13	2	<table><tr><td>10</td><td>8</td><td>9</td><td>13</td><td>2</td></tr></table>	10	8	9	13	2
10	8	9	13	2														
10	8	9	13	2														
10	8	9	13	2														
Sorted	<table><tr><td>10</td><td></td><td></td><td></td><td></td></tr></table>	10					<table><tr><td>8</td><td>10</td><td></td><td></td><td></td></tr></table>	8	10				<table><tr><td>8</td><td>9</td><td>10</td><td></td><td></td></tr></table>	8	9	10		
10																		
8	10																	
8	9	10																
Original	<table><tr><td>10</td><td>8</td><td>9</td><td>13</td><td>2</td></tr></table>	10	8	9	13	2												
10	8	9	13	2														
Sorted	<table><tr><td>8</td><td>9</td><td>10</td><td>13</td><td></td></tr></table>	8	9	10	13													
8	9	10	13															

Example

Original	<table><tr><td>10</td><td>8</td><td>9</td><td>13</td><td>2</td></tr></table>	10	8	9	13	2	<table><tr><td>10</td><td>8</td><td>9</td><td>13</td><td>2</td></tr></table>	10	8	9	13	2	<table><tr><td>10</td><td>8</td><td>9</td><td>13</td><td>2</td></tr></table>	10	8	9	13	2
10	8	9	13	2														
10	8	9	13	2														
10	8	9	13	2														
Sorted	<table><tr><td>10</td><td></td><td></td><td></td><td></td></tr></table>	10					<table><tr><td>8</td><td>10</td><td></td><td></td><td></td></tr></table>	8	10				<table><tr><td>8</td><td>9</td><td>10</td><td></td><td></td></tr></table>	8	9	10		
10																		
8	10																	
8	9	10																
Original	<table><tr><td>10</td><td>8</td><td>9</td><td>13</td><td>2</td></tr></table>	10	8	9	13	2	<table><tr><td>10</td><td>8</td><td>9</td><td>13</td><td>2</td></tr></table>	10	8	9	13	2	Done!					
10	8	9	13	2														
10	8	9	13	2														
Sorted	<table><tr><td>8</td><td>9</td><td>10</td><td>13</td><td></td></tr></table>	8	9	10	13		<table><tr><td>2</td><td>8</td><td>9</td><td>10</td><td>13</td></tr></table>	2	8	9	10	13						
8	9	10	13															
2	8	9	10	13														

- We can implement the insertions in the original array avoid using the **auxiliary** array.
- An insertion sort partitions the array into two regions



Insertion Sort

- while some elements unsorted:
 - Using linear search, find the location in the sorted portion where the 1st element of the unsorted portion should be inserted
 - Move all the elements after the insertion location up one position to make space for the new element

45															
38	45	60	66	79	47	13	74	36	21	94	22	57	16	29	81

the fourth iteration of this loop is shown here

An insertion sort of an array of five integers

Initial array:

29	10	14	37	13
-----------	----	----	----	----

Copy 10

29	29	14	37	13
----	----	----	----	----

Shift 29

10	29	14	37	13
-----------	-----------	----	----	----

Insert 10; copy 14

10	29	29	37	13
----	----	----	----	----

Shift 29

10	14	29	37	13
-----------	-----------	-----------	----	----

Insert 14; copy 37, insert 37 on top of itself

10	14	29	37	13
-----------	-----------	-----------	-----------	----

Copy 13

10	14	14	29	37
----	----	----	----	----

Shift 37, 29, 14

Sorted array:

10	13	14	29	37
-----------	-----------	-----------	-----------	-----------

Insert 13

Insertion Sort Algorithm

```
public void insertionSort(Comparable[] arr) {  
    for (int i = 1; i < arr.length; ++i) {  
        Comparable temp = arr[i];  
        int pos = i;  
        // Shuffle up all sorted items > arr[i]  
        while (pos > 0 &&  
            arr[pos-1].compareTo(temp) > 0) {  
            arr[pos] = arr[pos-1];  
            pos--;  
        } // end while  
        // Insert the current item  
        arr[pos] = temp;  
    }  
}
```

Insertion Sort: Number of Comparisons

# of Sorted Elements	Best case	Worst case
0	0	0
1	1	1
2	1	2
...	...	...
n-1	1	n-1
	n-1	$n(n-1)/2$

Remark: we only count comparisons of elements in the array.

More efficient sorting algorithms

Merge Sort

- Strategy

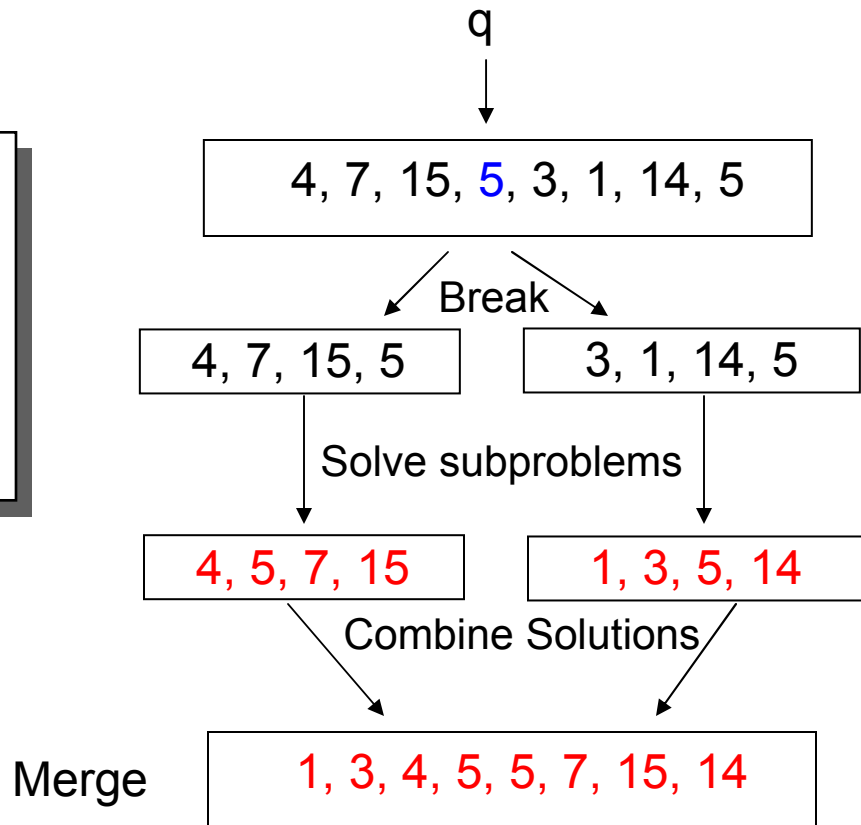
- break problem into smaller subproblems
- recursively solve subproblems
- combine solutions to answer

- Called "***divide-and-conquer***"

- we used the divide&conquer strategy in the binary search algorithm
-

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 two sorted list.

- Problem: given two sorted list A and B, create a sorted list C, that contains the elements of the two input lists.
 - Requirement: solve this problem in linear time (i.e $O(n)$ where n is the total number of elements in A and B).
-

-
- Strategy: *Take the smallest of the two frontmost elements of the list A and B, put it into C and advance to the next element of the list from which the current element was taken. Repeat this, until both sequences are empty.*
-

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

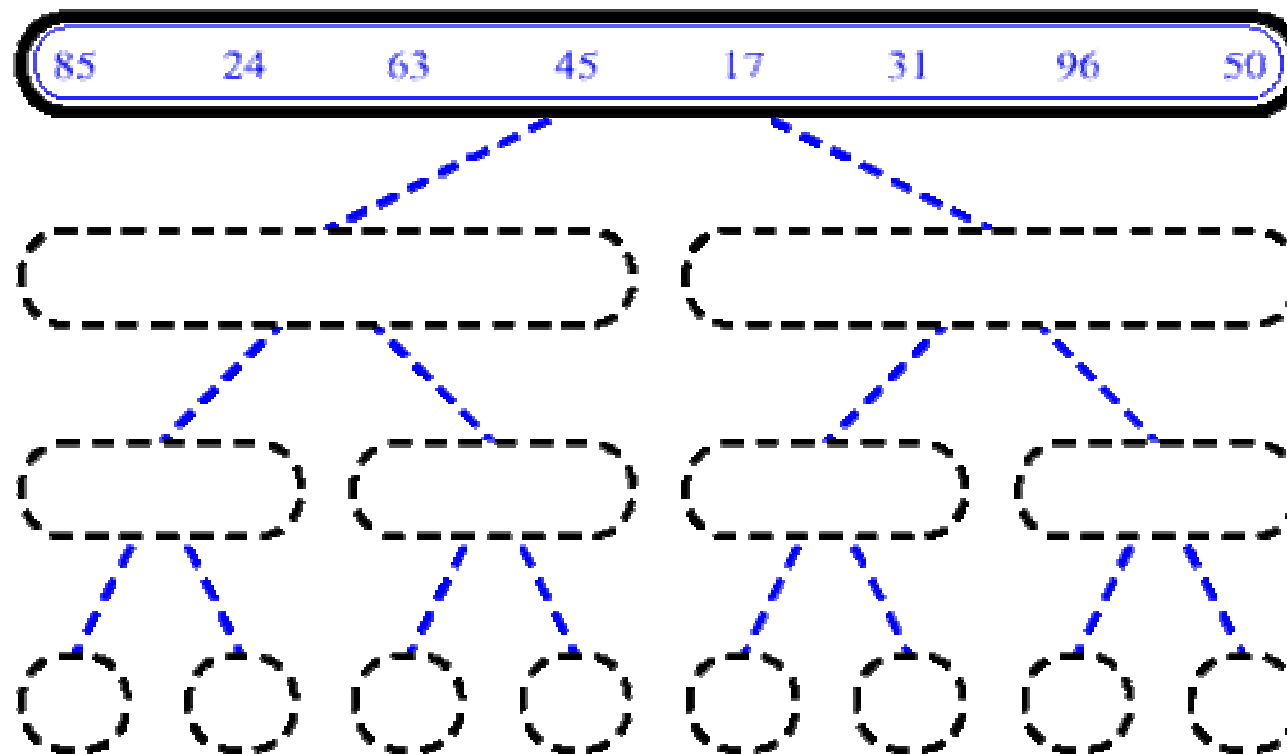
14	23	45	98
----	----	----	----

6	33	42	67
---	----	----	----

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

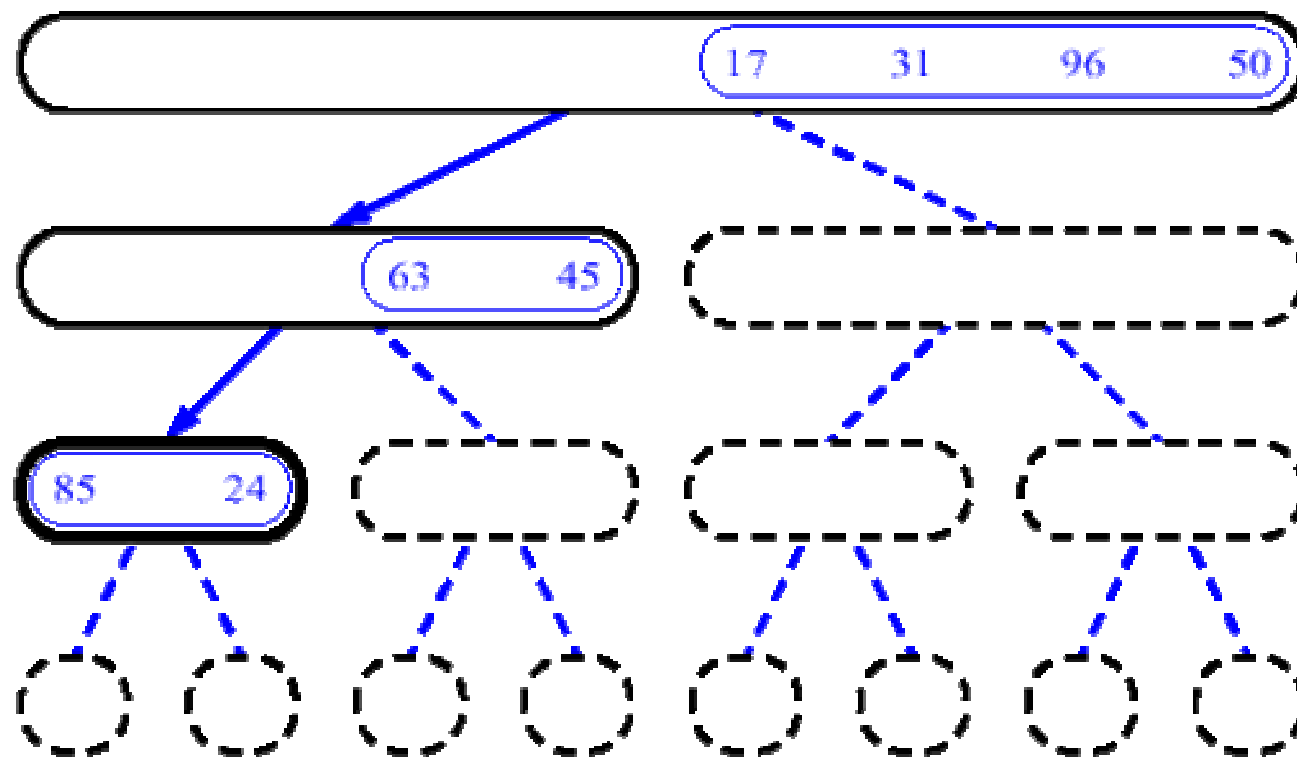
Merge

MergeSort (Example) - 1

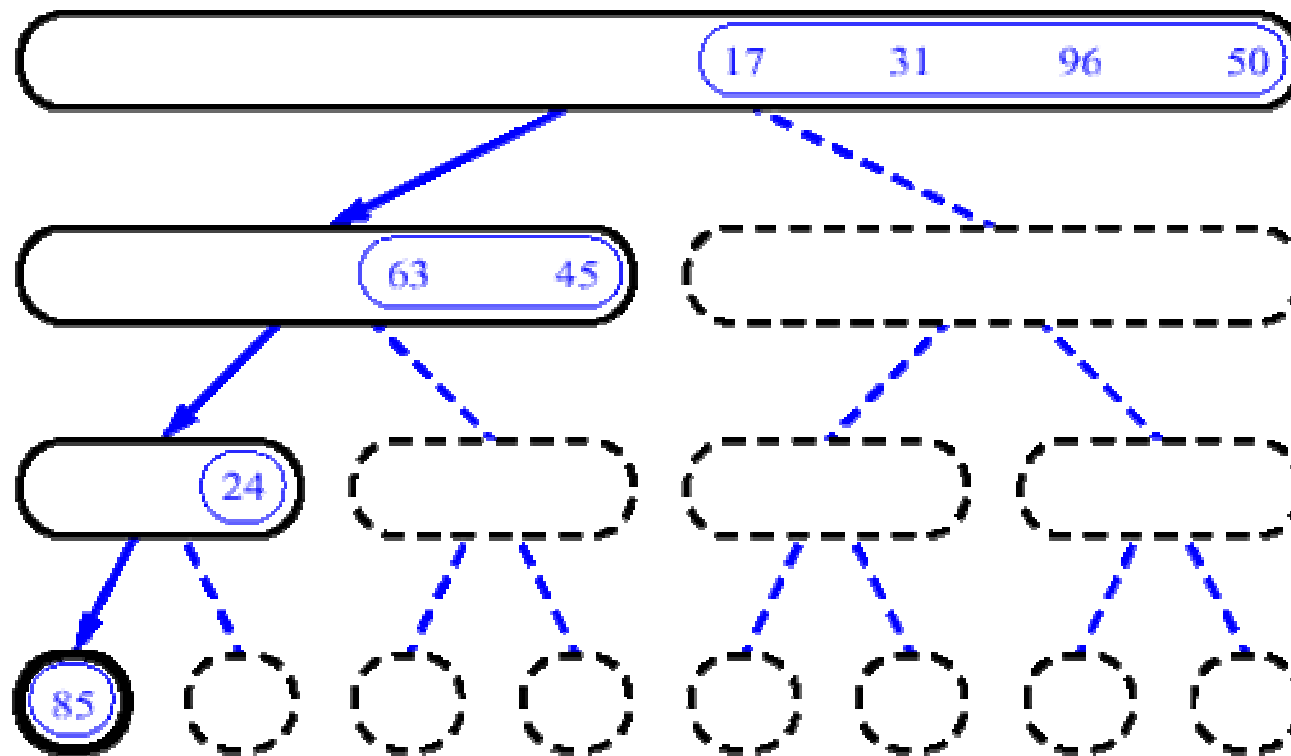


MergeSort (Example) - 2

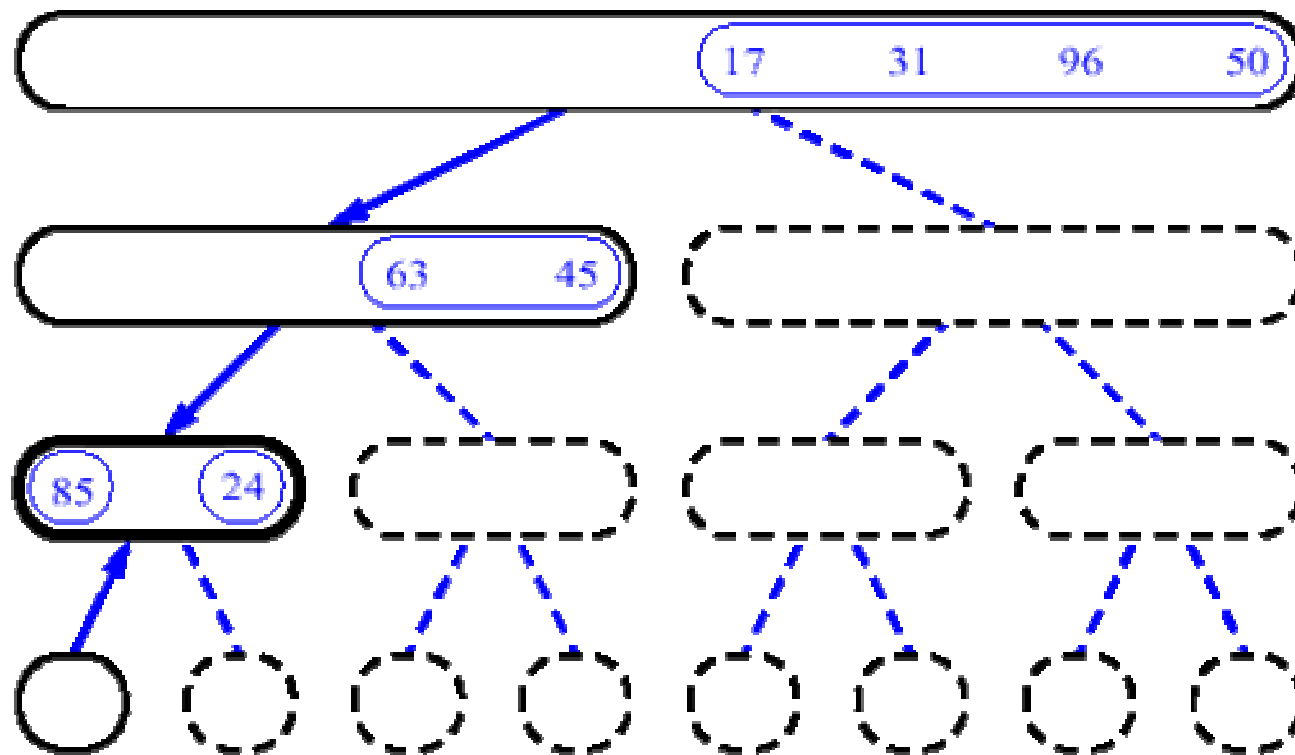
MergeSort (Example) - 3



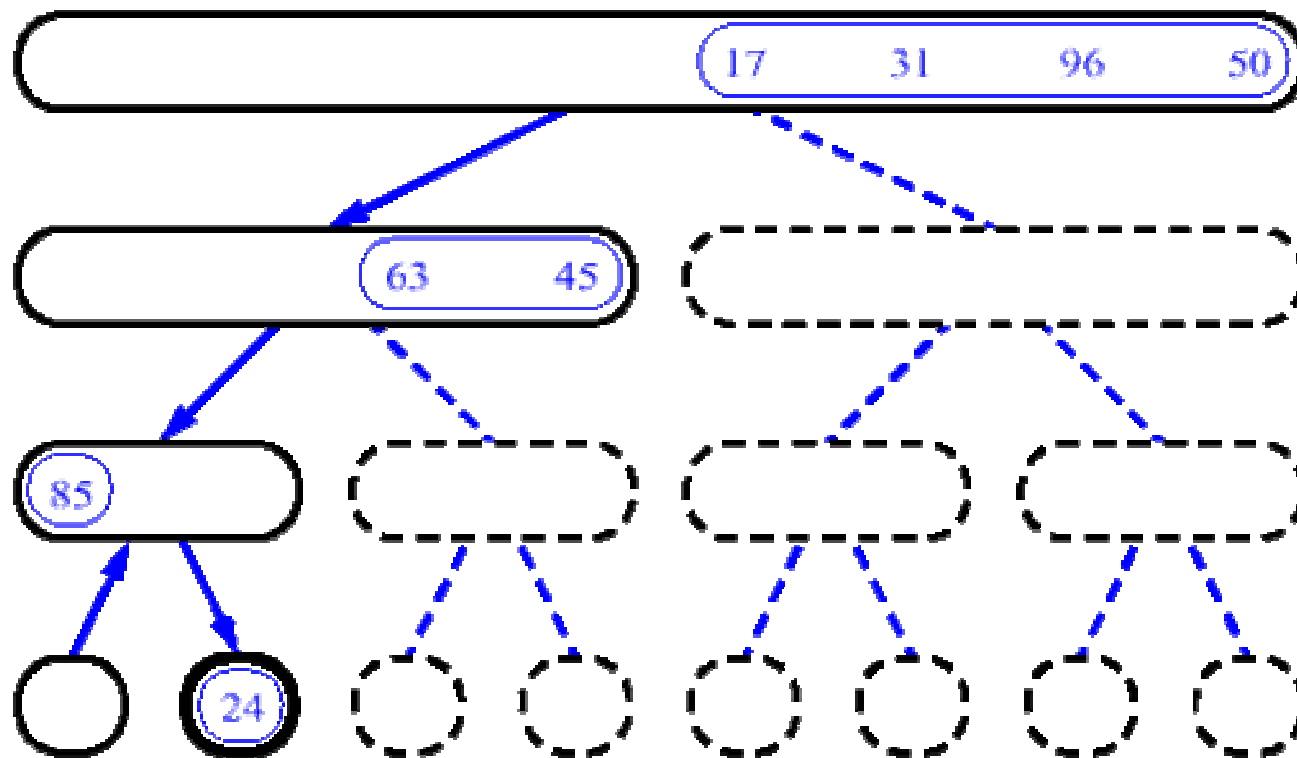
MergeSort (Example) - 4



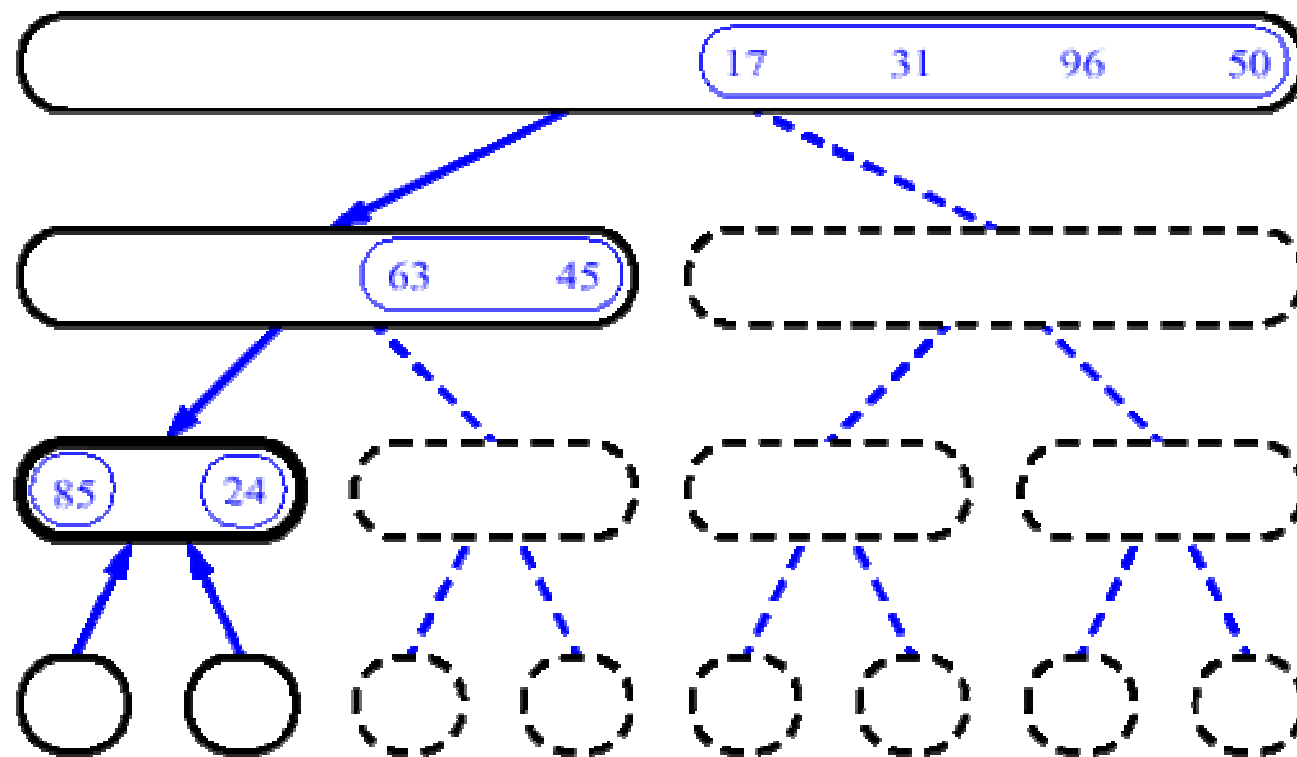
MergeSort (Example) - 5



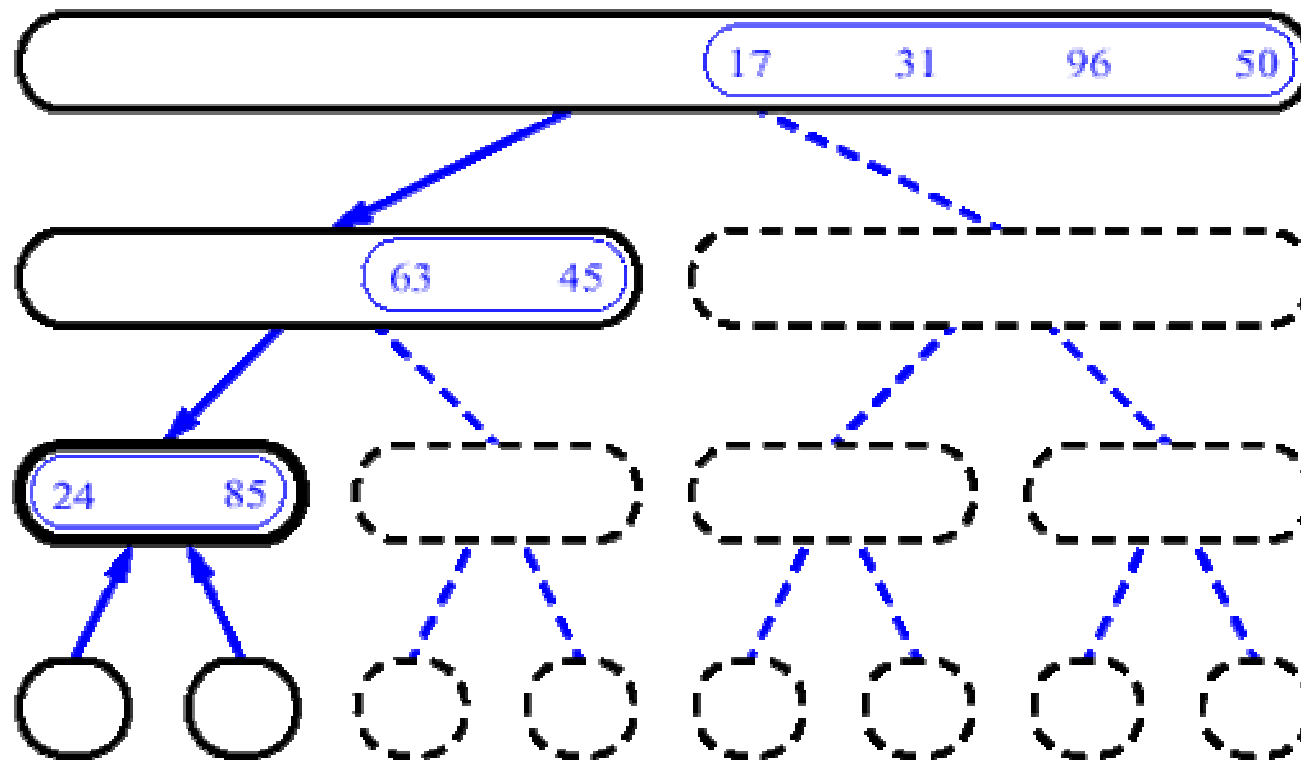
MergeSort (Example) - 6



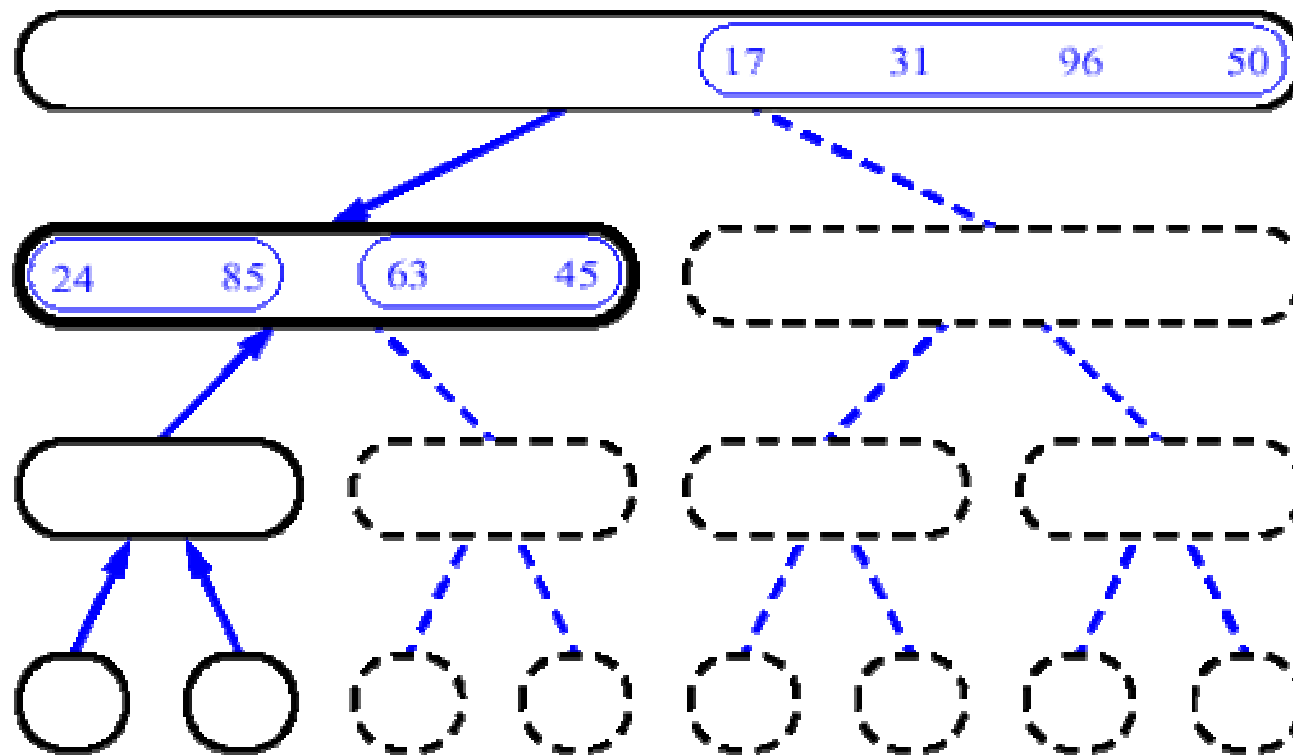
MergeSort (Example) - 7



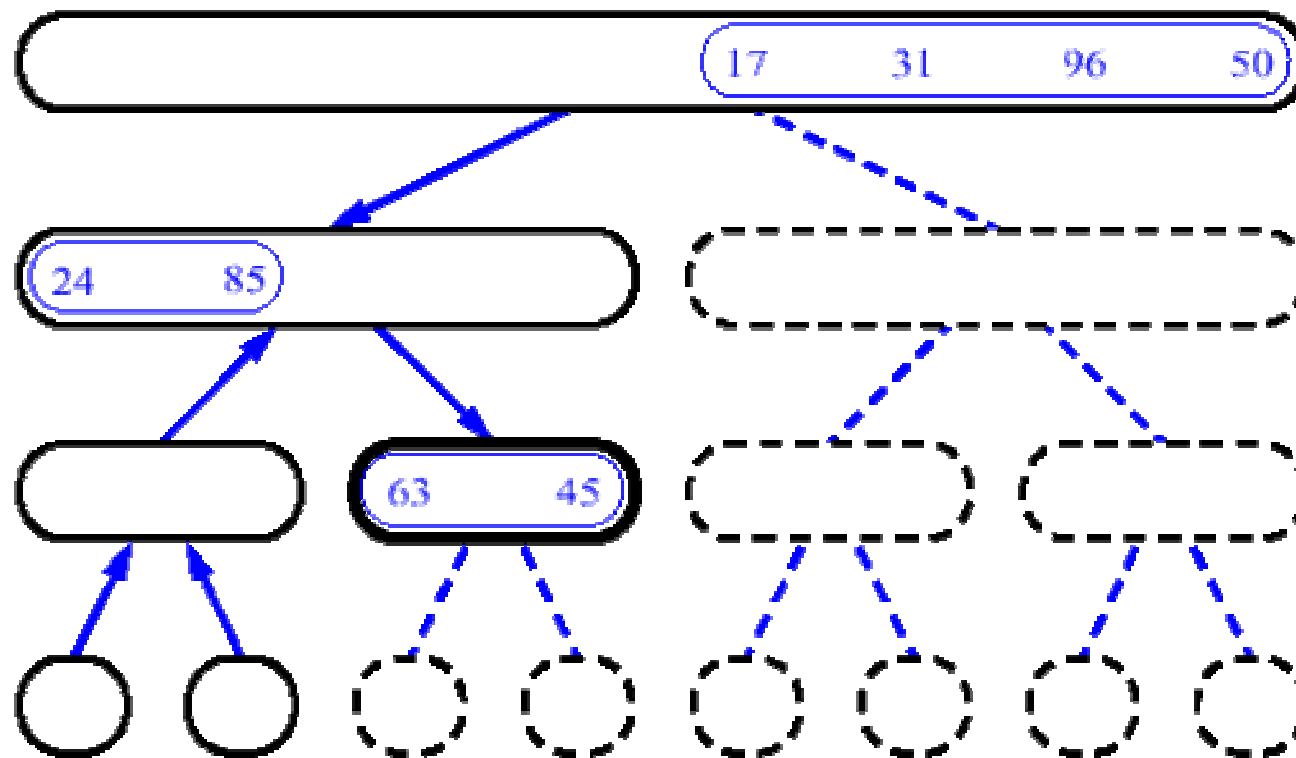
MergeSort (Example) - 8



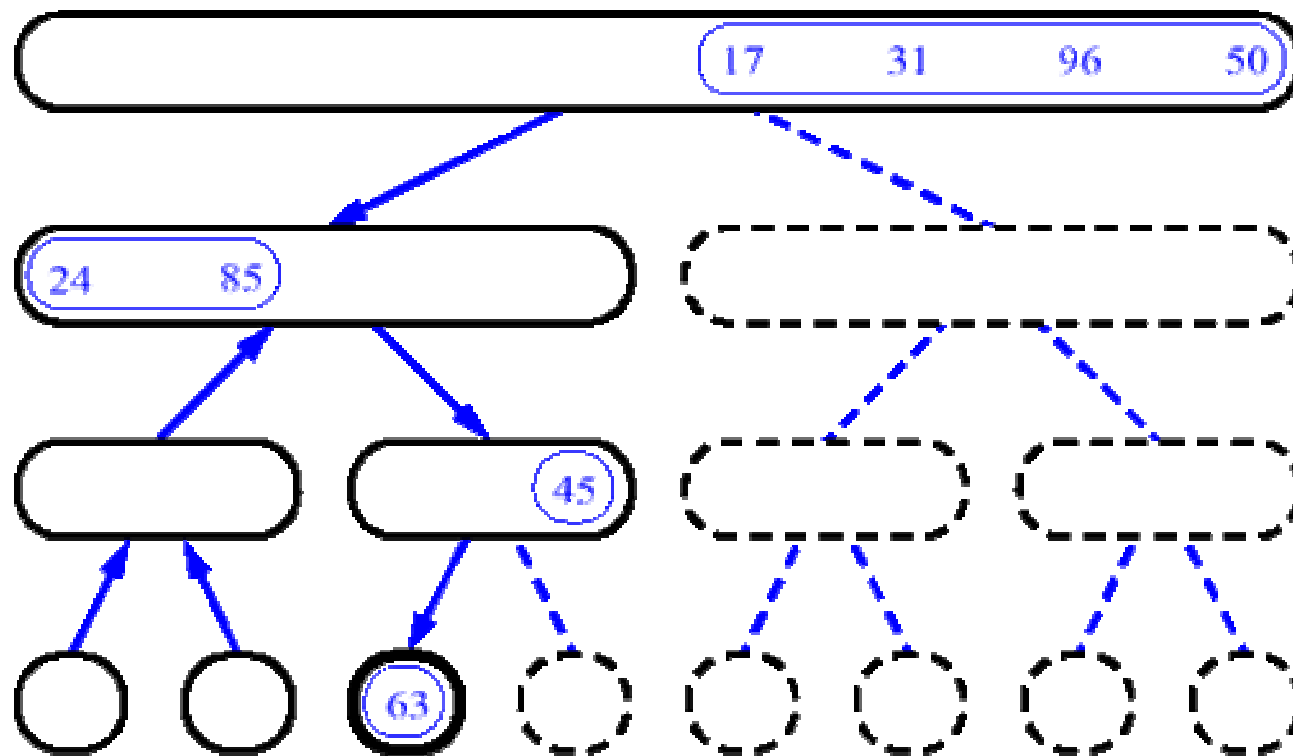
MergeSort (Example) - 9



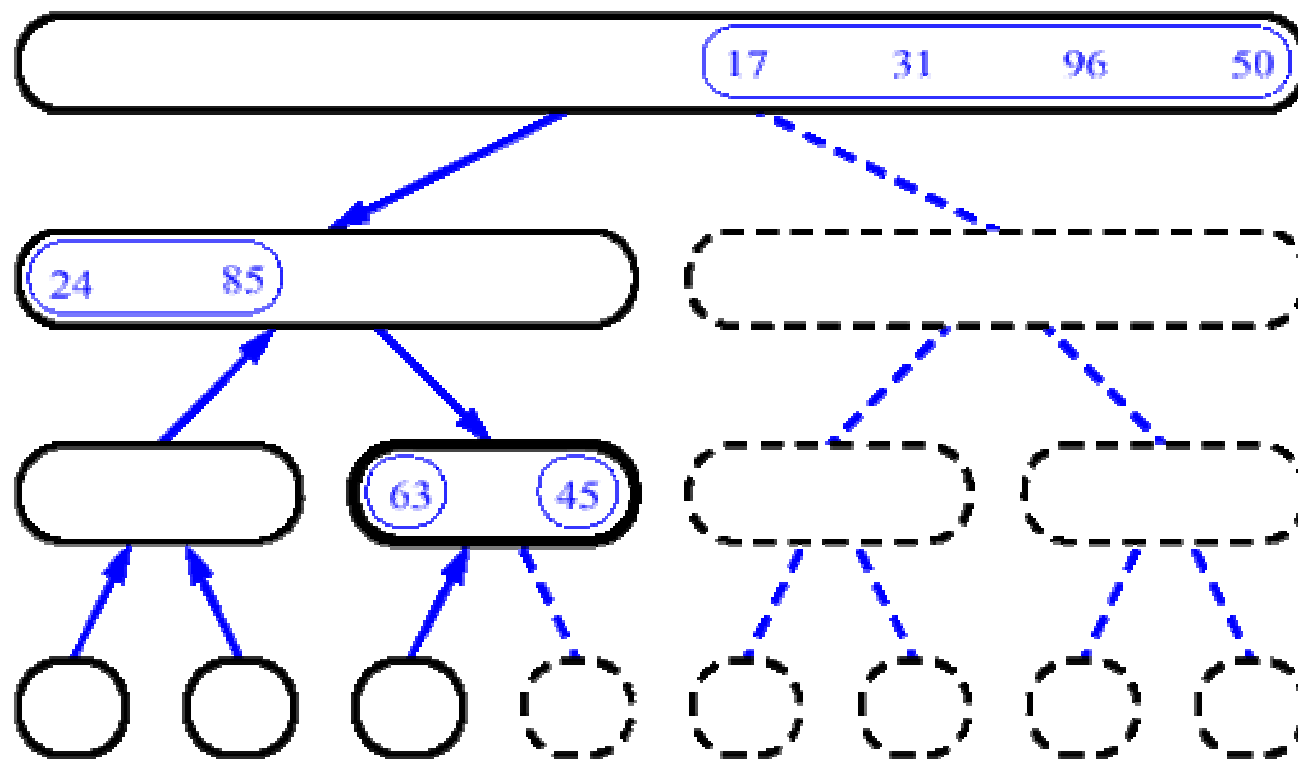
MergeSort (Example) - 10



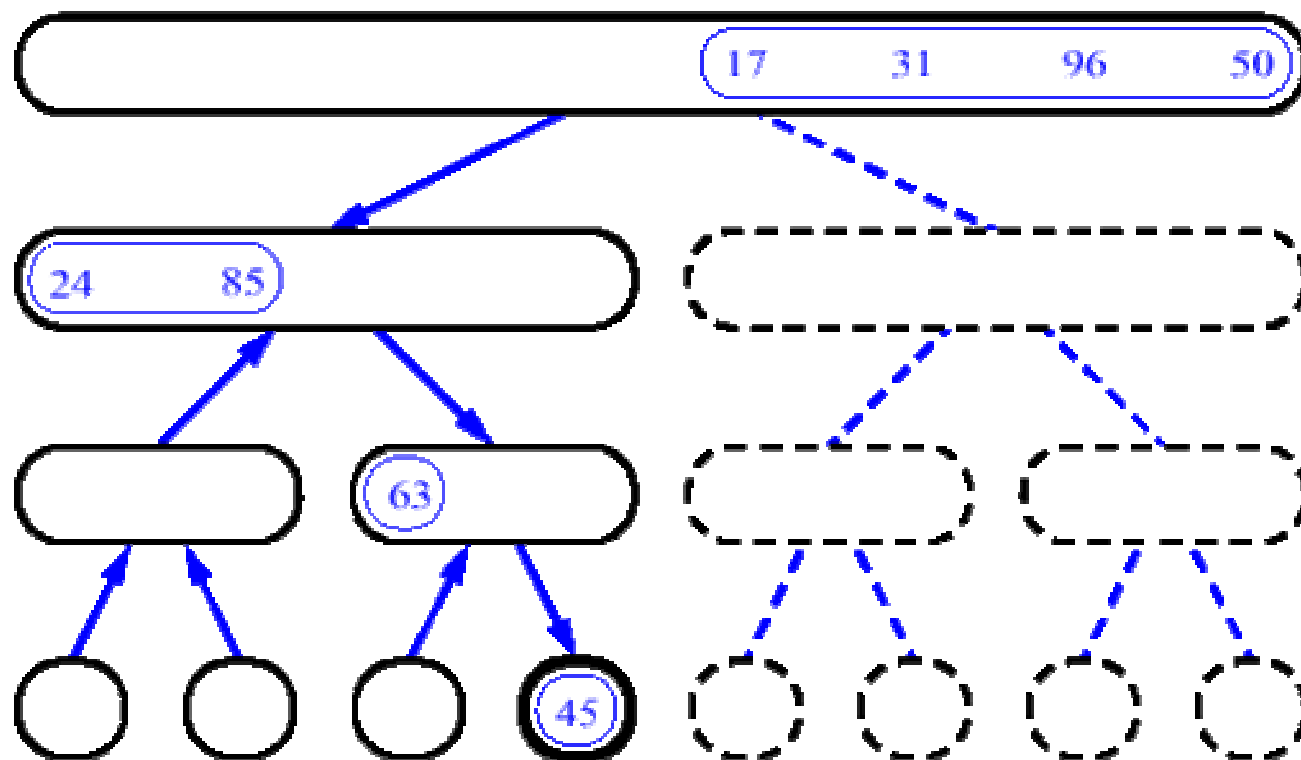
MergeSort (Example) - 11



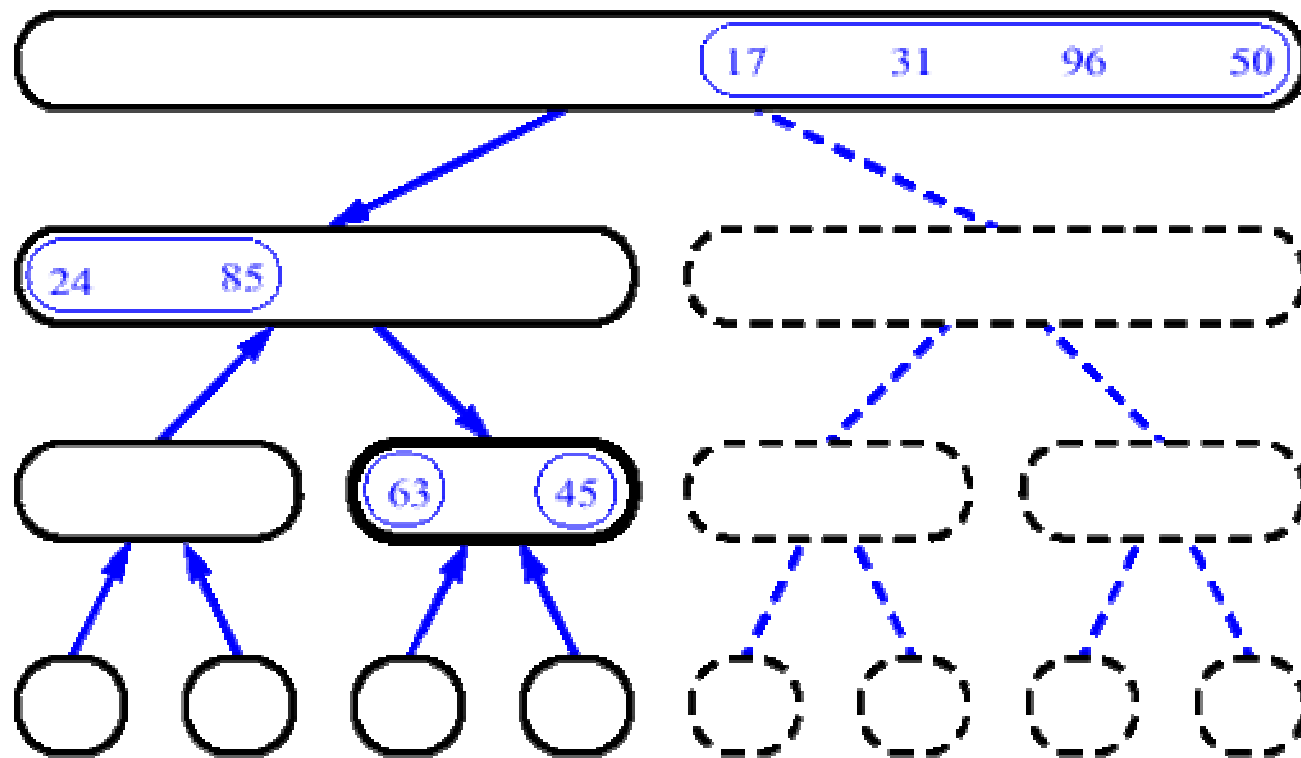
MergeSort (Example) - 12



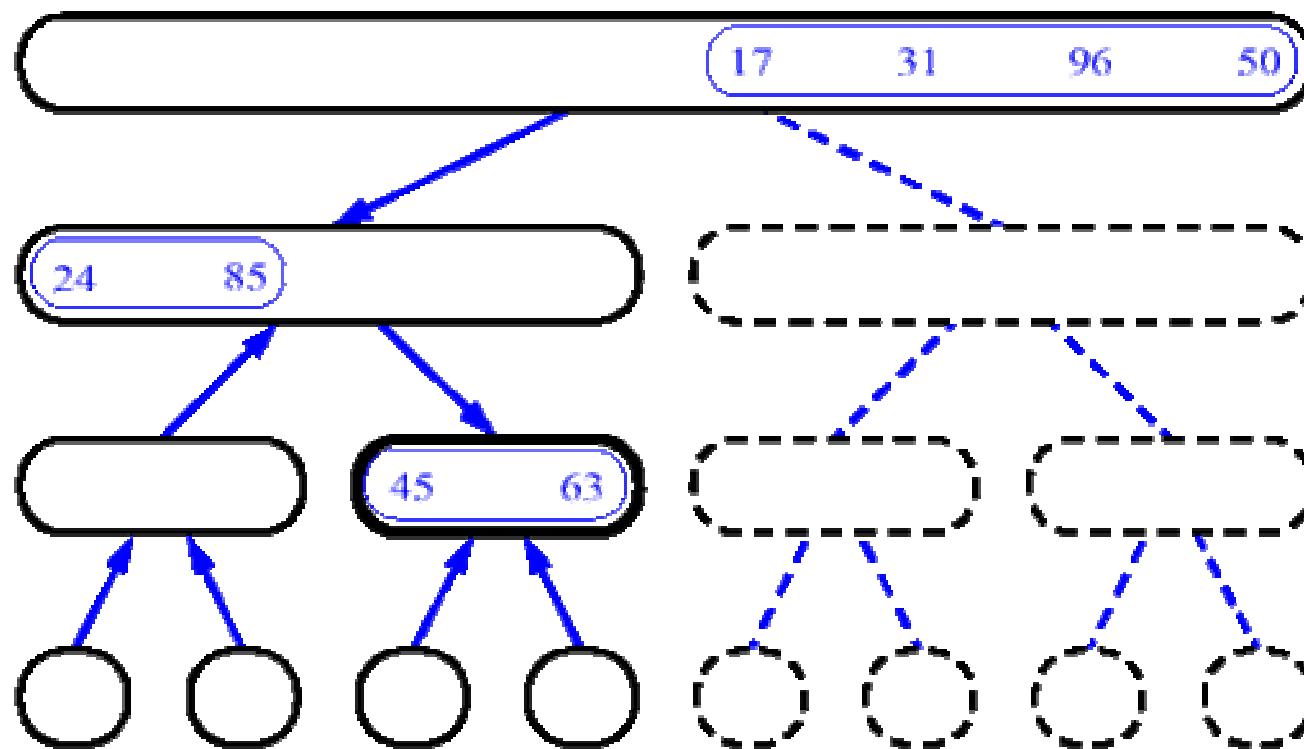
MergeSort (Example) - 13



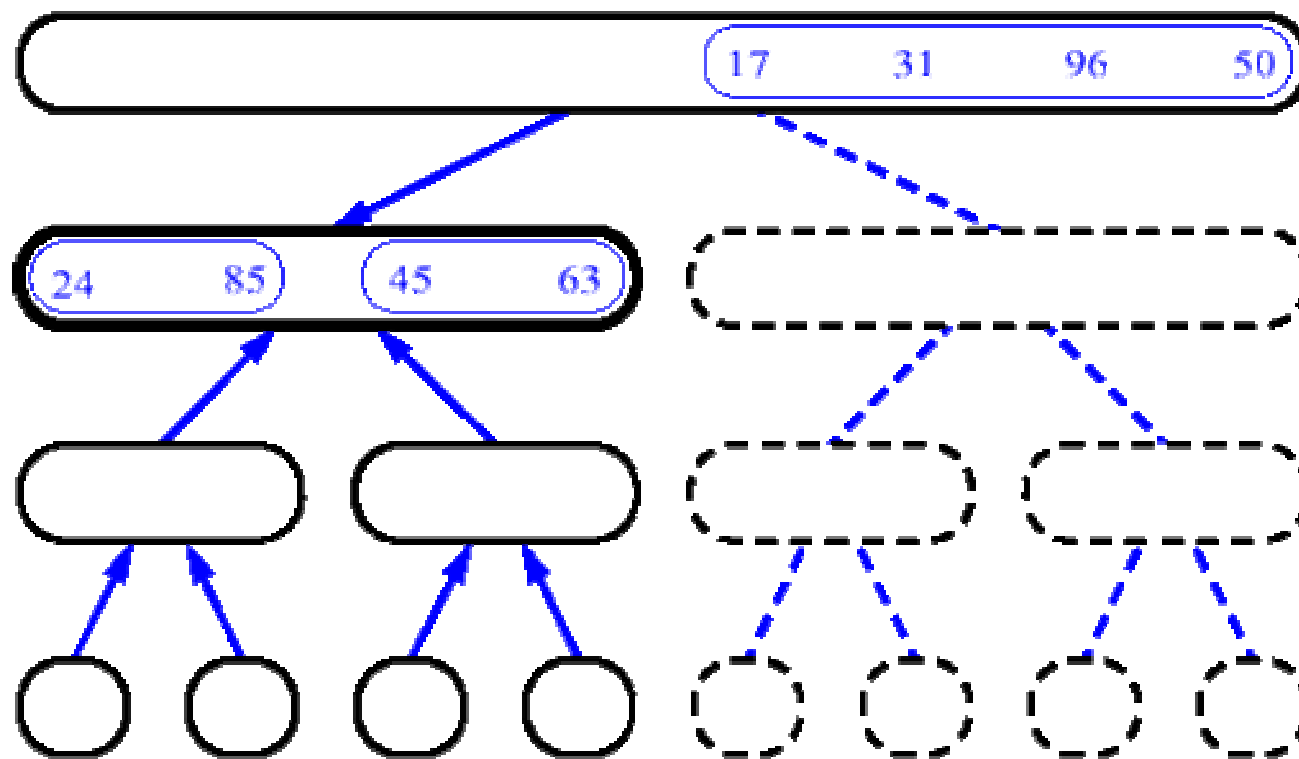
MergeSort (Example) - 14



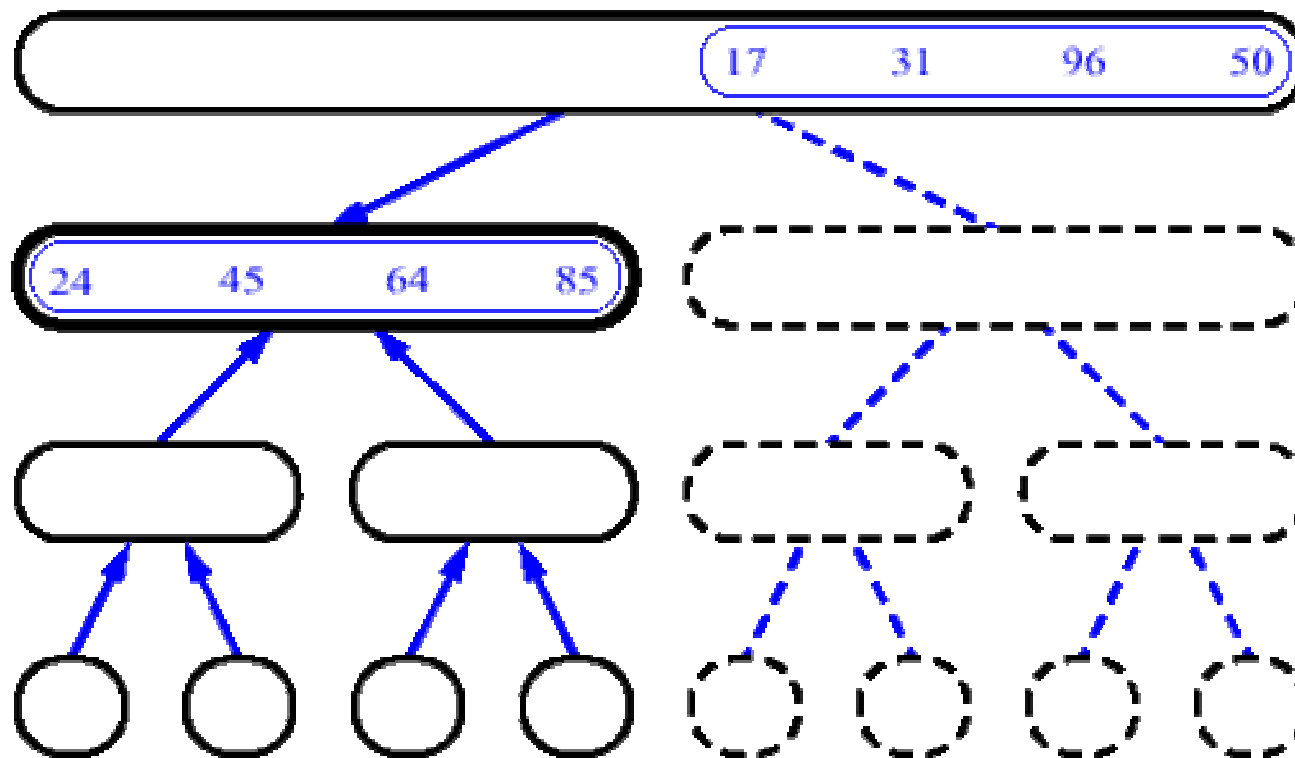
MergeSort (Example) - 15



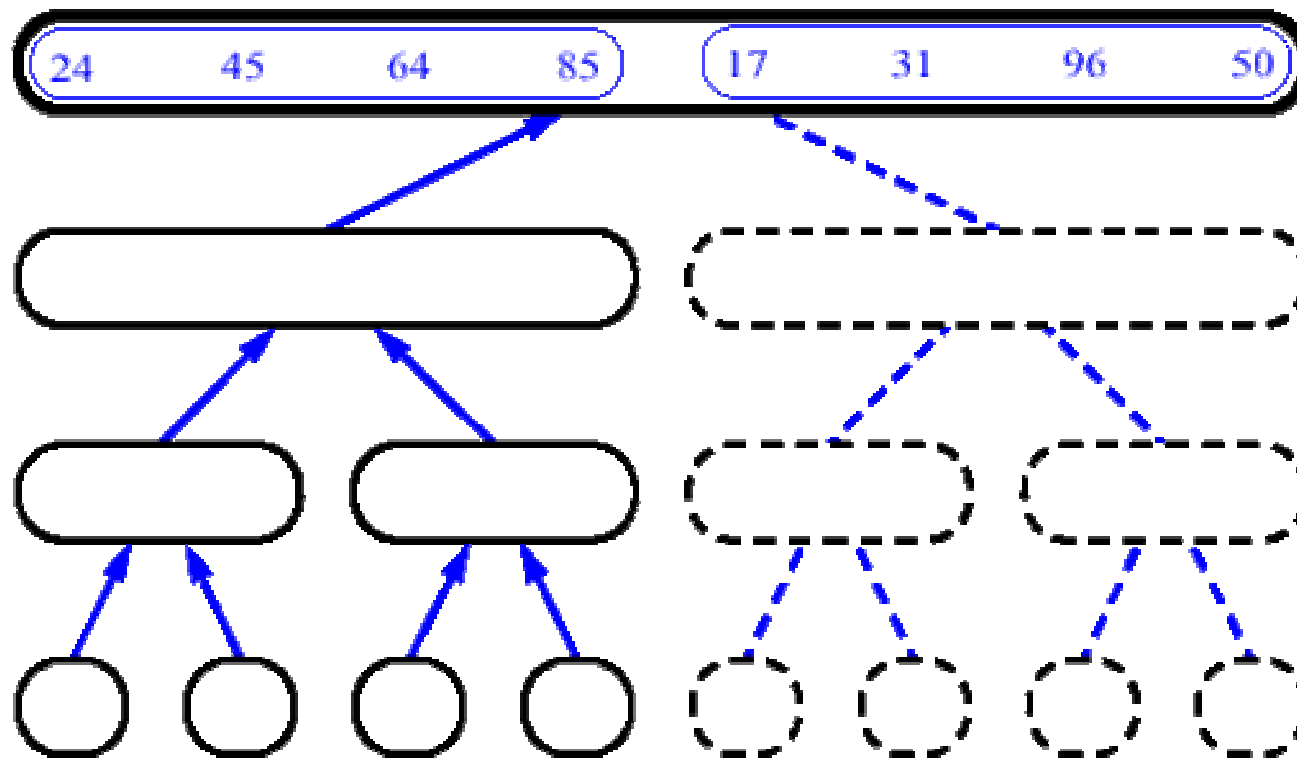
MergeSort (Example) - 16



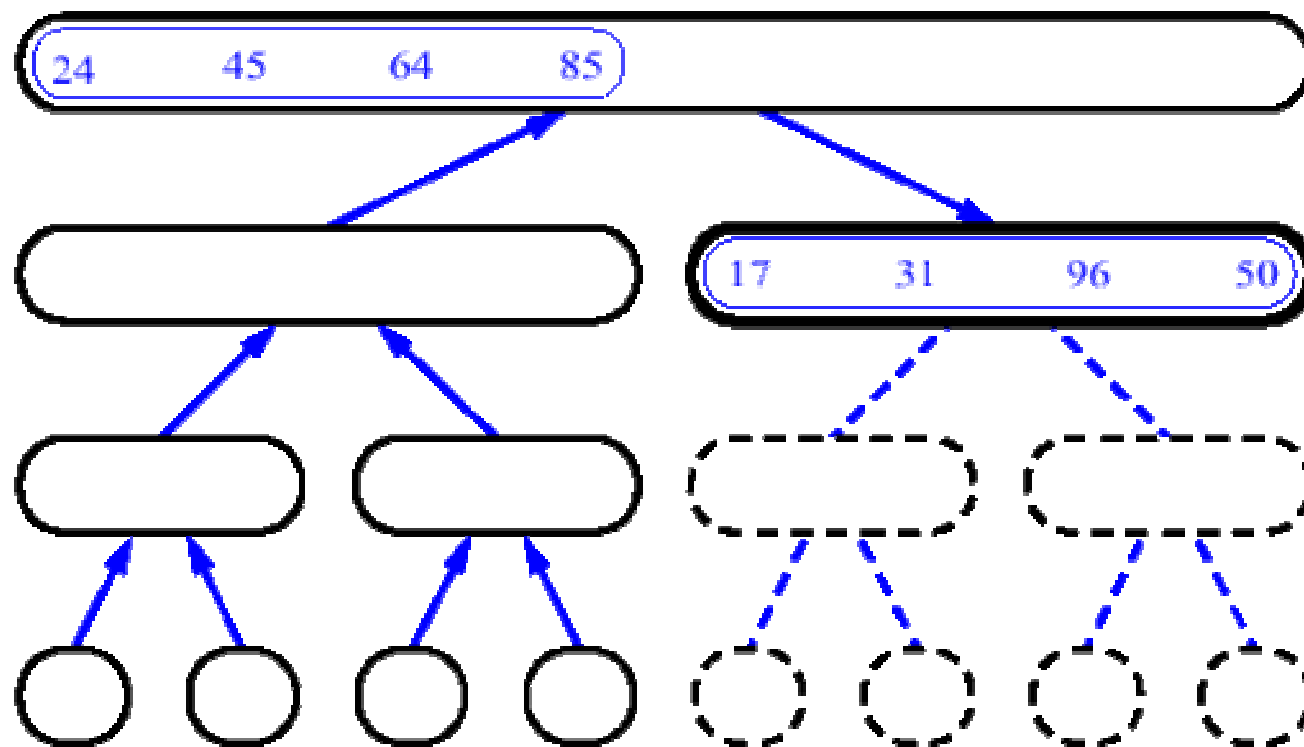
MergeSort (Example) - 17



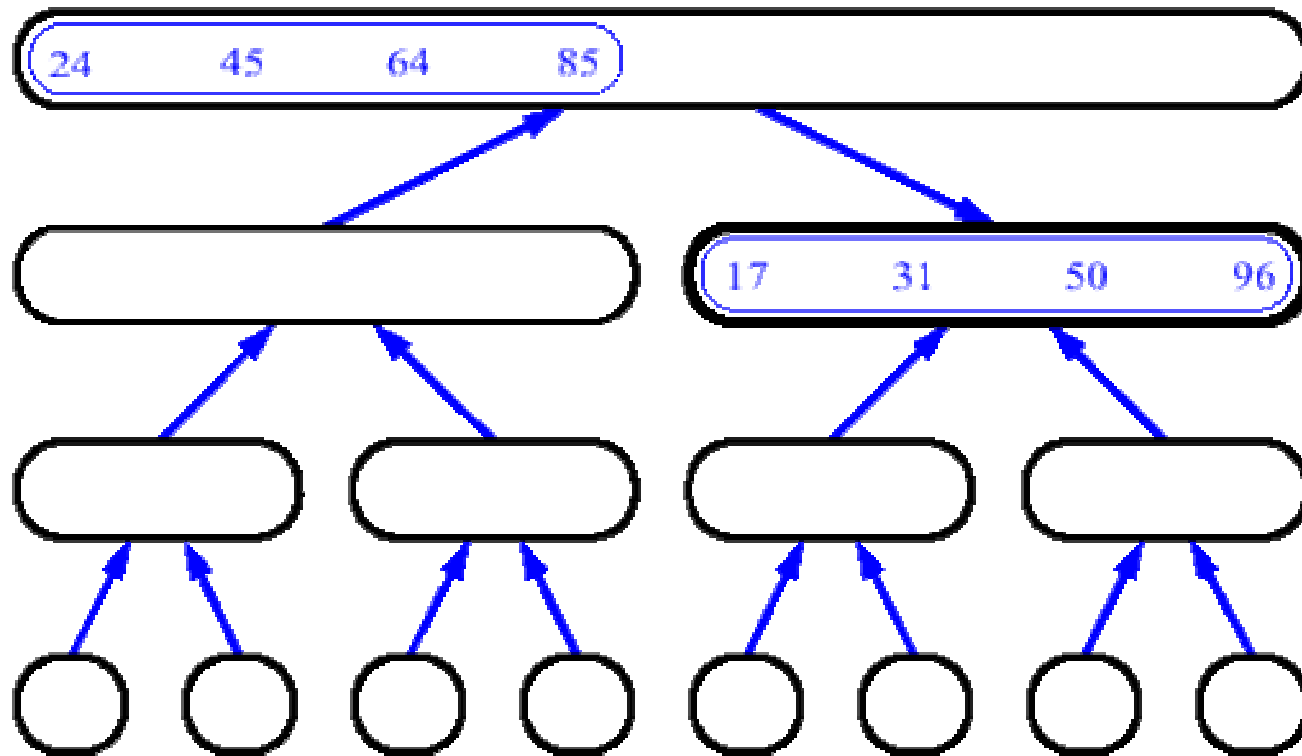
MergeSort (Example) - 18



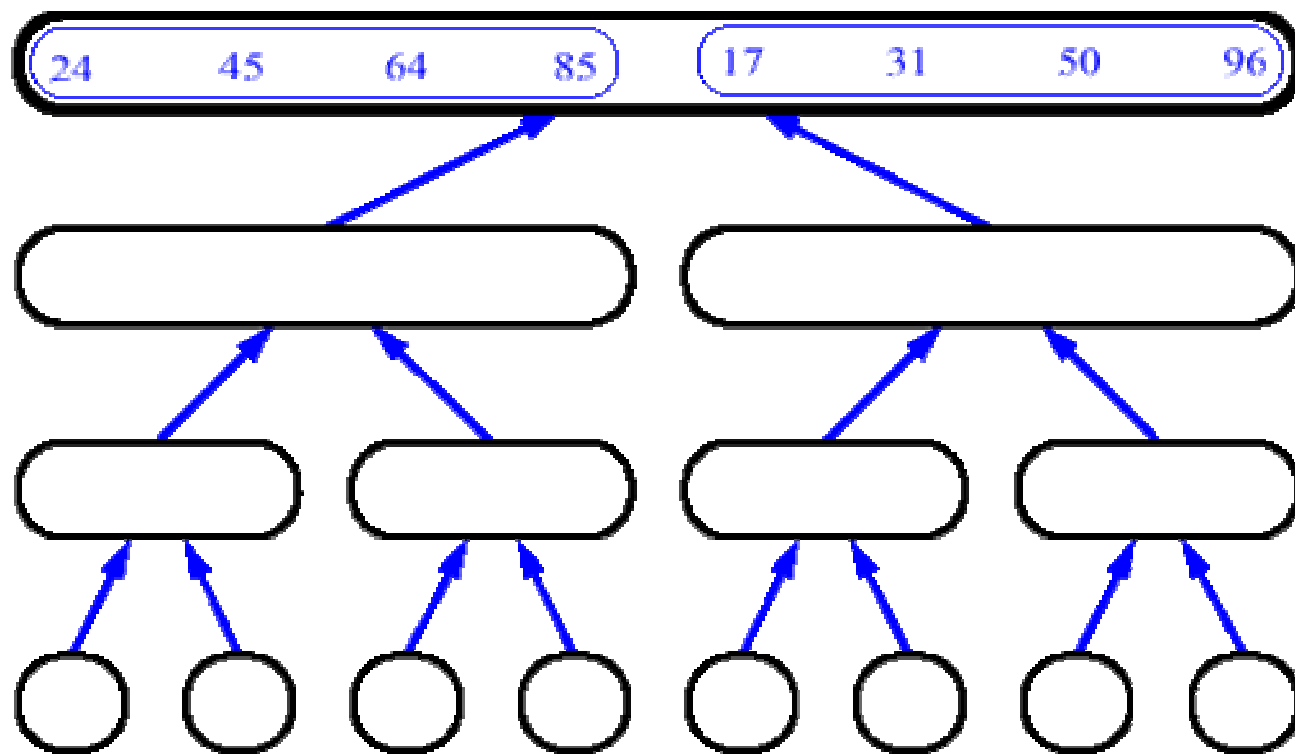
MergeSort (Example) - 19



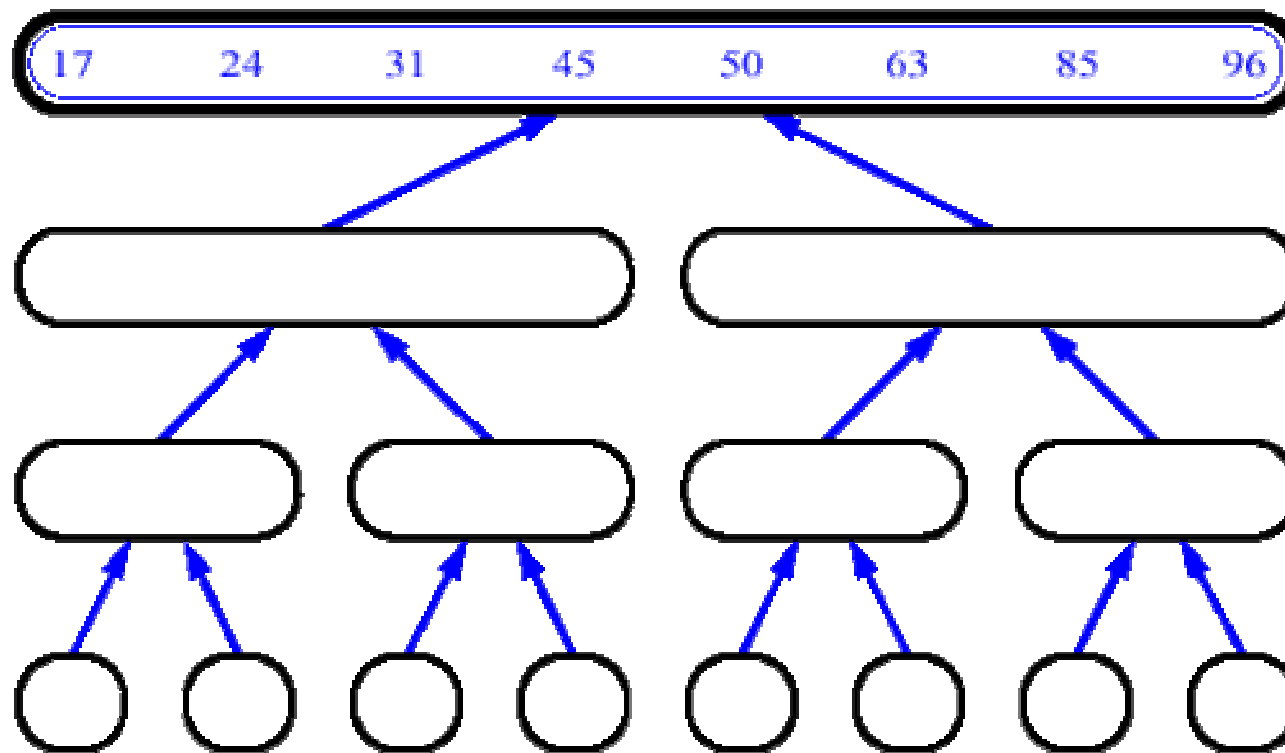
MergeSort (Example) - 20



MergeSort (Example) - 21



MergeSort (Example) - 22



Merge Sort

```
private void MergeSort(Comparable[] arr, int lowerBound,
                      int upperBound)
{
    if (lowerBound > upperBound)           // if range is 0 or 1,
        return;                           // no need to sort
    else
    {
        // find midpoint
        int mid = (lowerBound+upperBound) / 2;
        // sort low half
        MergeSort(arr, lowerBound, mid);
        // sort high half
        MergeSort(arr, mid+1, upperBound);
        // merge them
        merge(arr, lowerBound, mid, upperBound);
    } // end else
} // end MergeSort()
```

Merge Sort: merge

```
private void merge(Comparable[] arr, int low1, int high1, int high2)
{
    int n = high2 - low1 + 1;           // # of items
    Comparable[] tmp=new Comparable[n]; // tmp array
    int j = 0;                          // tmp index
    int low2 = high1 + 1;
    int i1 = low1;                      // index in the first part
    int i2 = low2;                      // index in the second part

    while (i1 <= high1 && i2 <= high2)
        if (arr[i1].compareTo(arr[i2]) < 0)
            tmp[j++] = arr[i1++];
        else
            tmp[j++] = arr[i2++];

    while (i1 <= high1) // copy remaining elements in the first part
        tmp[j++] = arr[i1++];

    while (i2 <= high2) // copy remaining elements in the second part
        tmp[j++] = arr[i2++];

    for (j=0; j<n; j++) // copy everything back to original array
        arr[low1+j] = tmp[j];
} // end merge()
```

Mergesort

theArray:

8	1	4	3	2
---	---	---	---	---

Divide the array in half



Sort the halves

Merge the halves:

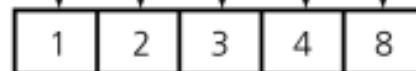
- a. $1 < 2$, so move 1 from left half to tempArray
- b. $4 > 2$, so move 2 from right half to tempArray
- c. $4 > 3$, so move 3 from right half to tempArray
- d. Right half is finished, so move rest of left half to tempArray

Temporary array
tempArray:



Copy temporary array back into
original array

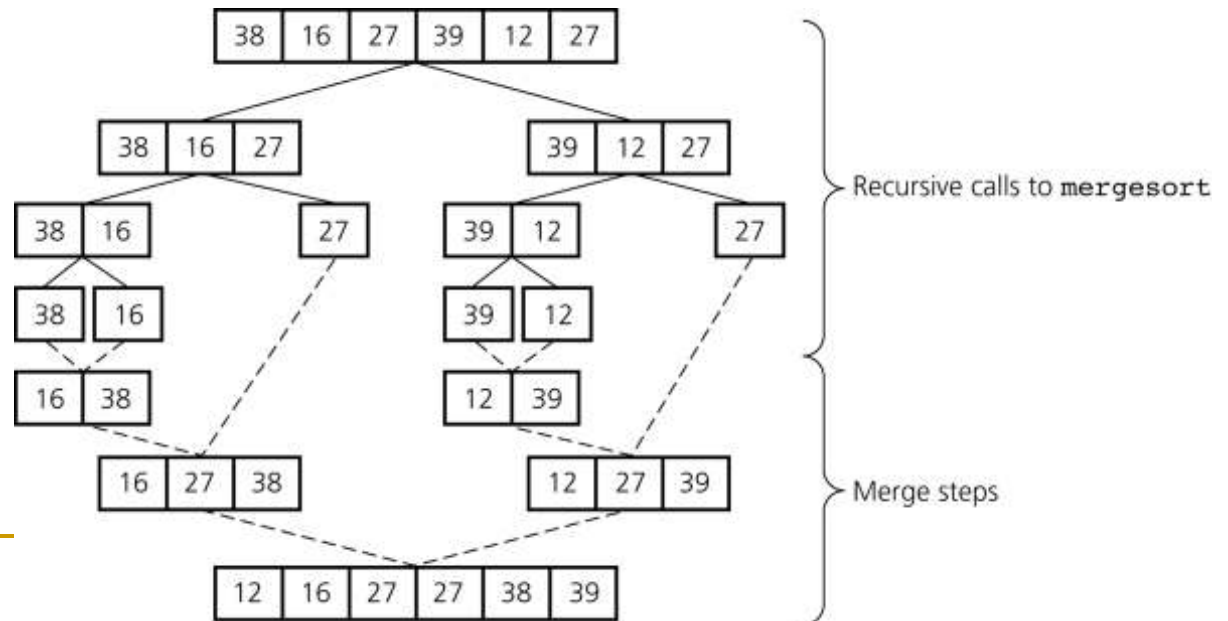
theArray:



A mergesort with an auxiliary temporary array

Merge Sort Summarized

- To sort n numbers
 - if $n=1$ done!
 - recursively sort 2 lists of numbers $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ elements
 - merge 2 sorted lists in $O(n)$ time



Running time of MergeSort

- The running time can be expressed as a recurrence:

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

Repeated Substitution Method

$$\begin{array}{ll} T(n) = 2T(n/2) + cn & n > 1 \\ = 1 & n=1 \end{array}$$

- $T(n) = 2T(n/2) + cn$
$$\begin{aligned} &= 2 \{ 2T(n/2^2) + c \cdot n/2 \} + cn \\ &= 2^2 T(n/2^2) + c \cdot 2n \\ &= 2^2 \{ 2T(n/2^3) + c \cdot n/2^2 \} + c \cdot 2n \\ &= 2^3 T(n/2^3) + c \cdot 3n \\ &= \dots\dots \\ &= 2^k T(n/2^k) + c \cdot k \cdot n \\ &= \dots \\ &= 2^{\log n} T(1) + c \cdot (\log n) \cdot n \text{ when } n/2^k = 1 \Rightarrow k = \log_2 n \\ &= 2^{\log n} \cdot 1 + c \cdot (\log n) \cdot n \\ &= n + c \cdot n \log n \quad \text{where } 2^{\log n} = n \end{aligned}$$
 - Therefore, $T(n) = O(n \log n)$
-

The Substitution method

$$T(n) = 2T(n/2) + cn$$

- **Guess:** $T(n) = O(n \log n)$
- **Proof** by Mathematical Induction:

Prove that $T(n) \leq d n \log n$ for $d > 0$

$$T(n) \leq 2(d \cdot n/2 \cdot \log n/2) + cn$$

(where $T(n/2) \leq d \cdot n/2 (\log n/2)$ by induction hypothesis)

$$\leq dn \log n/2 + cn$$

$$= dn \log n - dn + cn$$

$$= dn \log n + (c-d)n$$

$$\leq dn \log n \quad \text{if } d \geq c$$

- Therefore, $T(n) = O(n \log n)$
-

Up to here will be on midterm
