Merge Sort Algorithm

```
MergeSort(A, left, right) {
   if (left < right) {
       mid = floor((left + right) / 2);
       MergeSort(A, left, mid);
       MergeSort(A, mid+1, right);
       Merge(A, left, mid, right);
   }
}

// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A
//    (how long should this take?)
```

Merge Sort Algorithm

```
MergeSort(A, left, right) {
  if (left < right) {
      mid = floor((left + right) / 2);
      MergeSort(A, left, mid);
      MergeSort(A, mid+1, right);
      Merge(A, left, mid, right);
  }
}

// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A
//    (how long should this take?)
```

Merge Sort Algorithm

```
MergeSort(A, left, right) {
   if (left < right) {
      mid = floor((left + right) / 2);
      MergeSort(A, left, mid);
      MergeSort(A, mid+1, right);
      Merge(A, left, mid, right);
   }
}

// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A
//    (how long should this take?)
```

Merge Sort Algorithm

```
MergeSort(A, left, right) {
   if (left < right) {
      mid = floor((left + right) / 2);
      MergeSort(A, left, mid);
      MergeSort(A, mid+1, right);
      Merge(A, left, mid, right);
   }
}

// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A
//    (how long should this take?)
```

Merge Sort Algorithm

```
MergeSort(A, left, right) {
   if (left < right) {
      mid = floor((left + right) / 2);
      MergeSort(A, left, mid);
      MergeSort(A, mid+1, right);
      Merge(A, left, mid, right);
   }
}

// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A
//    (how long should this take?)
```

Merge Sort Algorithm

```
MergeSort(A, left, right) {
   if (left < right) {
       mid = floor((left + right) / 2);
       MergeSort(A, left, mid);
       MergeSort(A, mid+1, right);
       Merge(A, left, mid, right);
   }
}

// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A
//    (how long should this take?)
```

Merge()

```
Merge(A, left, mid, right)
  n1 = mid-left+1
  n2 = right-mid
  create two sorted subarrays L[0..n1] and R[0..n2]
  for i=0 to n1-1
    do L[i]=A[left+i]
  for j=0 to n2-1
    do R[j]=A[mid+j+1]
  L[n1]=∞
  R[n2]=∞
  i=0
  j=0
  for k=left to right
    do if L[i] ≤ R[j]
      then A[k]=L[i]
        i=i+1
      else A[K]=R[j]
        j=j+1
```

Merge()

$\Theta(1)$

```
Merge(A, left, mid, right)
  n1 = mid-left+1
  n2 = right-mid
  create two sorted subarrays L[0..n1] and R[0..n2]
  for i=0 to n1-1
    do L[i]=A[left+i]
  for j=0 to n2-1
    do R[j]=A[mid+j+1]
  L[n1]=∞
  R[n2]=∞
  i=0
  j=0
  for k=left to right
    do if L[i] ≤ R[j]
      then A[k]=L[i]
        i=i+1
      else A[K]=R[j]
        j=j+1
```

Merge()

```
Merge(A, left, mid, right)
  n1 = mid-left+1
  n2 = right-mid
  create two sorted subarrays L[0..n1] and R[0..n2]
  for i=0 to n1-1
    do L[i]=A[left+i]
  for j=0 to n2-1
    do R[j]=A[mid+j+1]
  L[n1]=∞
  R[n2]=∞
  i=0
  j=0
  for k=left to right
    do if L[i] ≤ R[j]
      then A[k]=L[i]
        i=i+1
      else A[K]=R[j]
        j=j+1
```

$\Theta(1)$

$\Theta(n_1)$

$\Theta(n_2)$

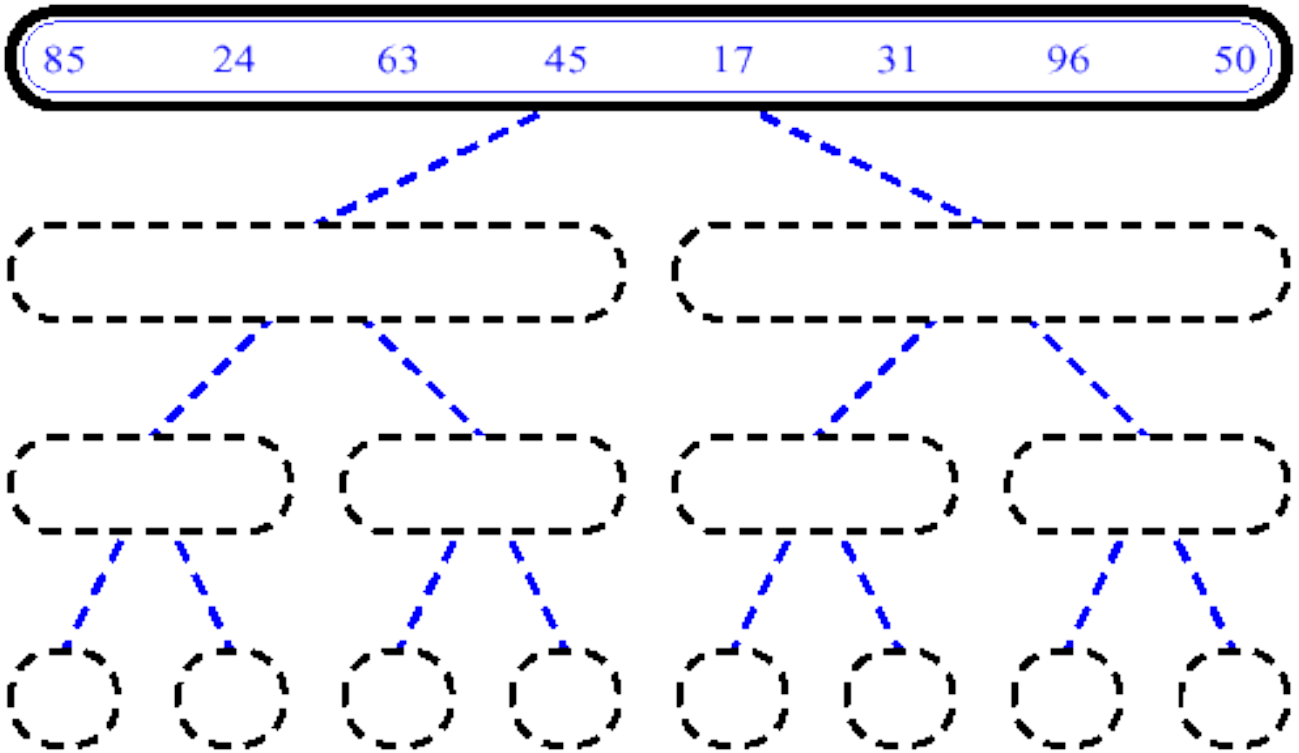
Merge()

```
Merge(A, left, mid, right)
  n1 = mid-left+1                                   Θ(1)
  n2 = right-mid                                    Θ(1)
  create two sorted subarrays L[0..n1] and R[0..n2] Θ(1)
  for i=0 to n1-1                                   Θ(n1)
    do L[i]=A[left+i]
  for j=0 to n2-1                                   Θ(n2)
    do R[j]=A[mid+j+1]
  L[n1]=∞                                           Θ(1)
  R[n2]=∞                                           Θ(1)
  i=0                                               Θ(1)
  j=0                                               Θ(1)
  for k=left to right                               Θ(n1+n2)
    do if L[i] ≤ R[j]                               Θ(n1+n2)
      then A[k]=L[i]                                Θ(n1+n2)
        i=i+1                                       Θ(n1+n2)
      else A[K]=R[j]                                Θ(n1+n2)
        j=j+1                                       Θ(n1+n2)
```

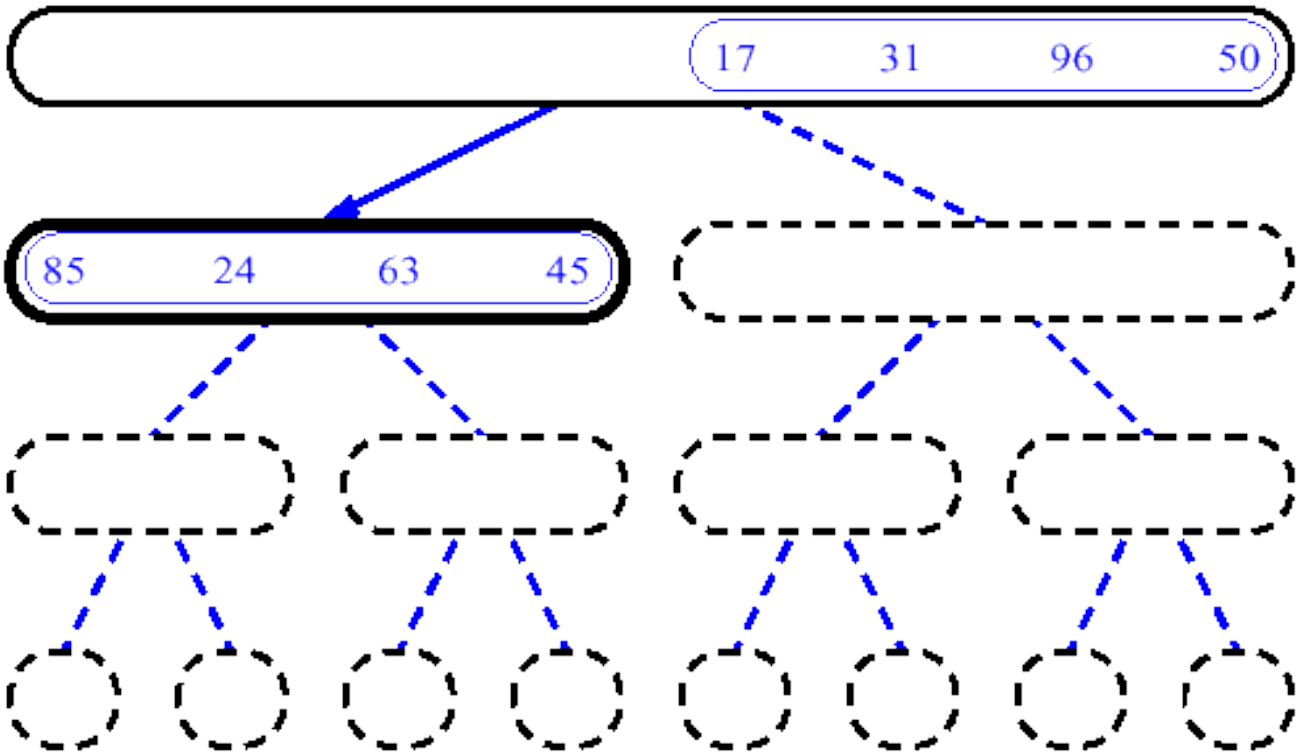
Merge()

```
Merge(A, left, mid, right)
  n1 = mid-left+1                                          Θ(1)
  n2 = right-mid                                           Θ(1)
  create two sorted subarrays L[0..n1] and R[0..n2]        Θ(1)
  for i=0 to n1-1                                          Θ(n1)
    do L[i]=A[left+i]
  for j=0 to n2-1                                          Θ(n2)
    do R[j]=A[mid+j+1]
  L[n1]=∞                                                  Θ(1)
  R[n2]=∞                                                  Θ(1)
  i=0                                                      Θ(1)
  j=0                                                      Θ(1)
  for k=left to right                                      Θ(n)
    do if L[i] ≤ R[j]
      then A[k]=L[i]
        i=i+1
      else A[K]=R[j]
        j=j+1
```

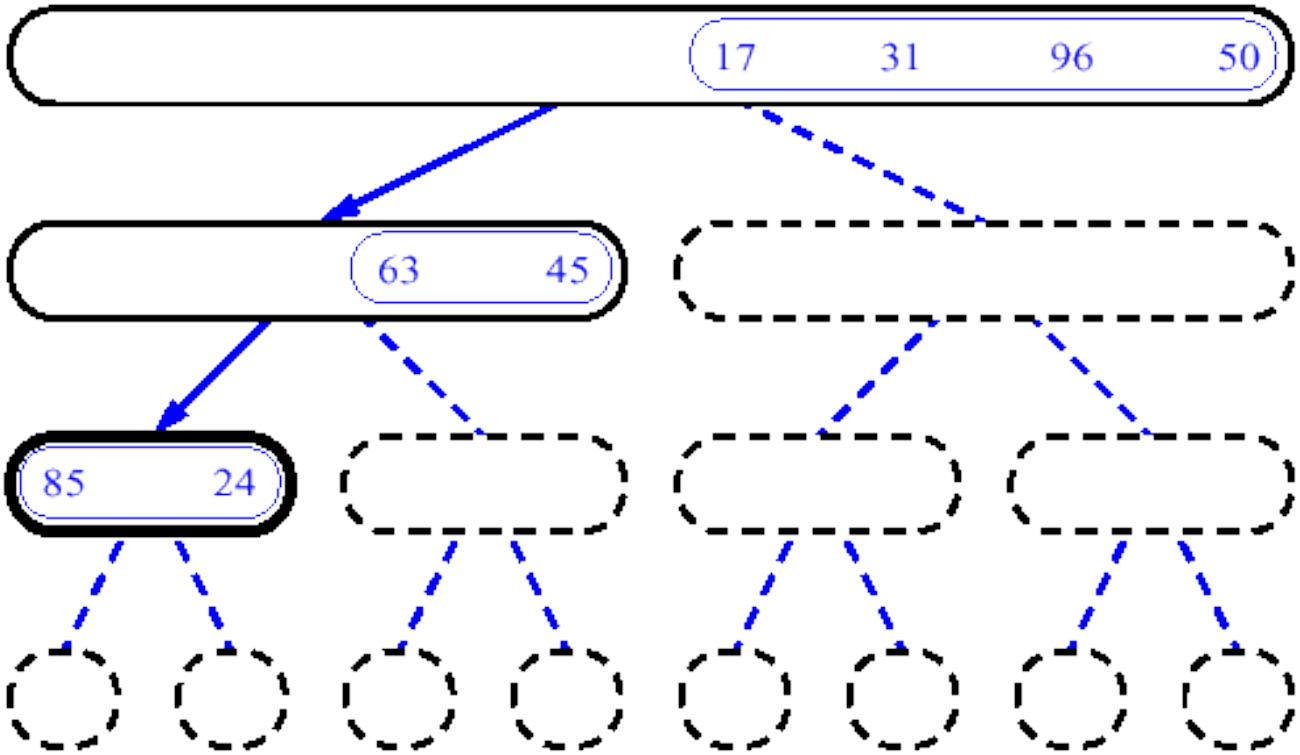
`MergeSort()` running on a array



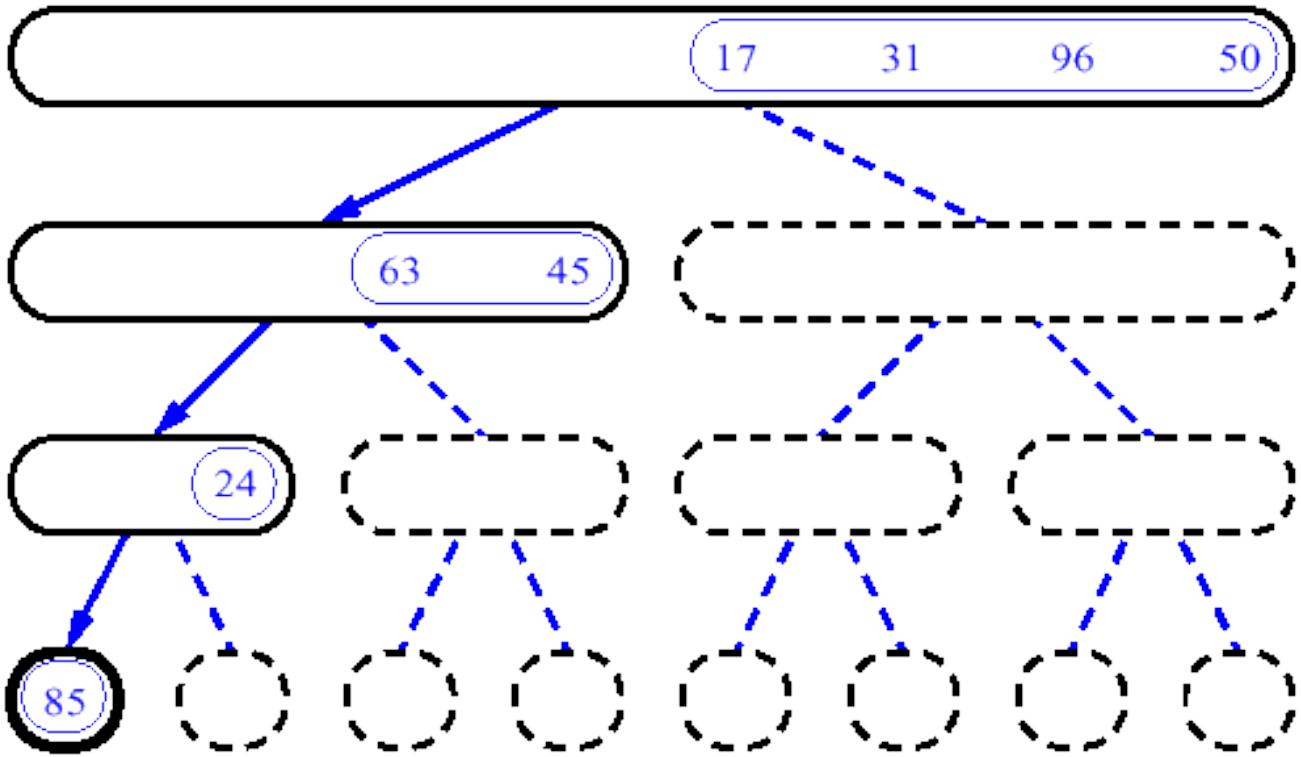
`MergeSort()` running on a array



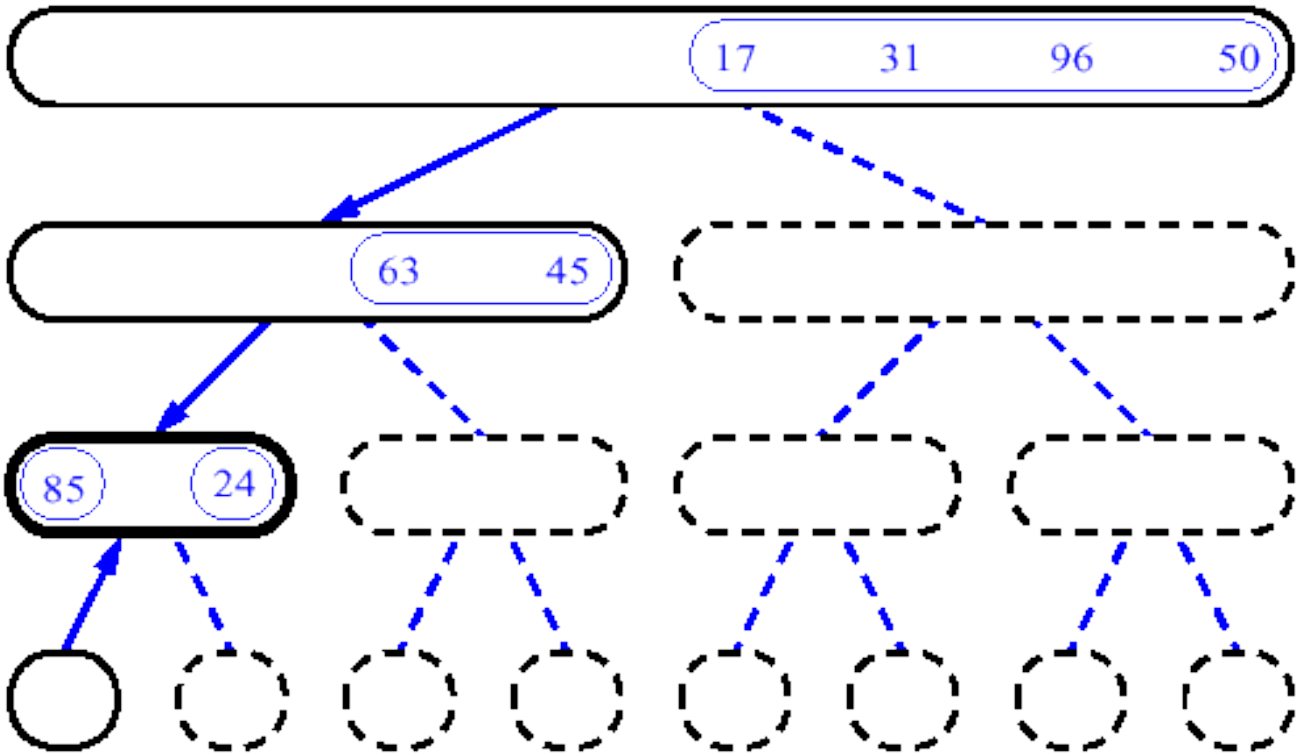
`MergeSort()` running on a array



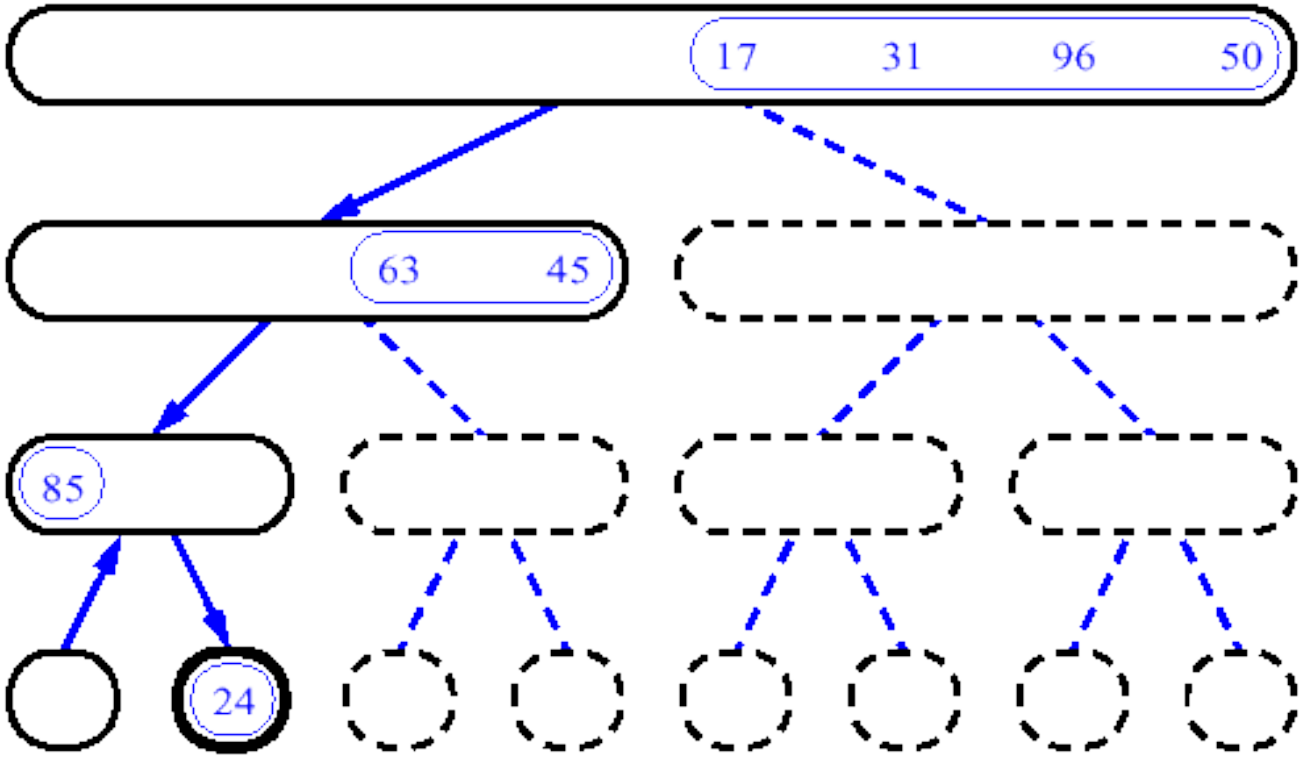
`MergeSort()` running on a array



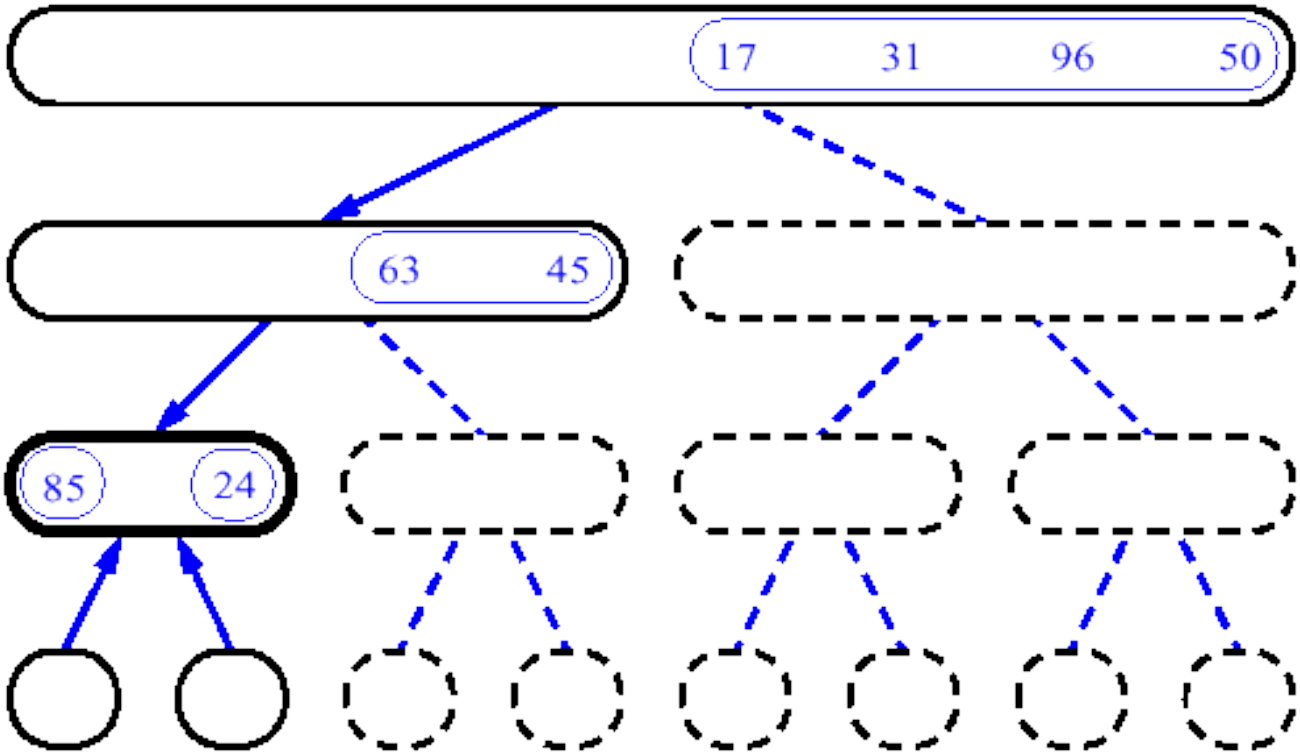
`MergeSort()` running on a array



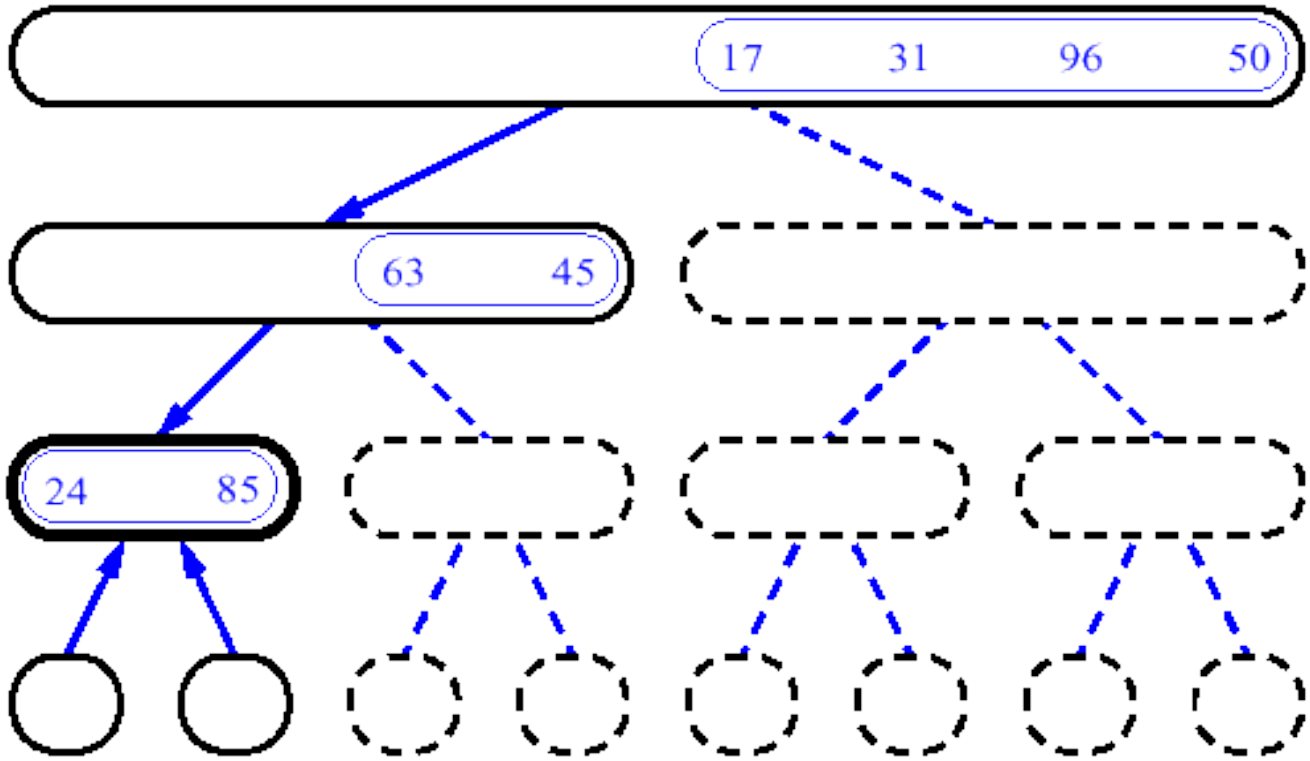
`MergeSort()` running on a array



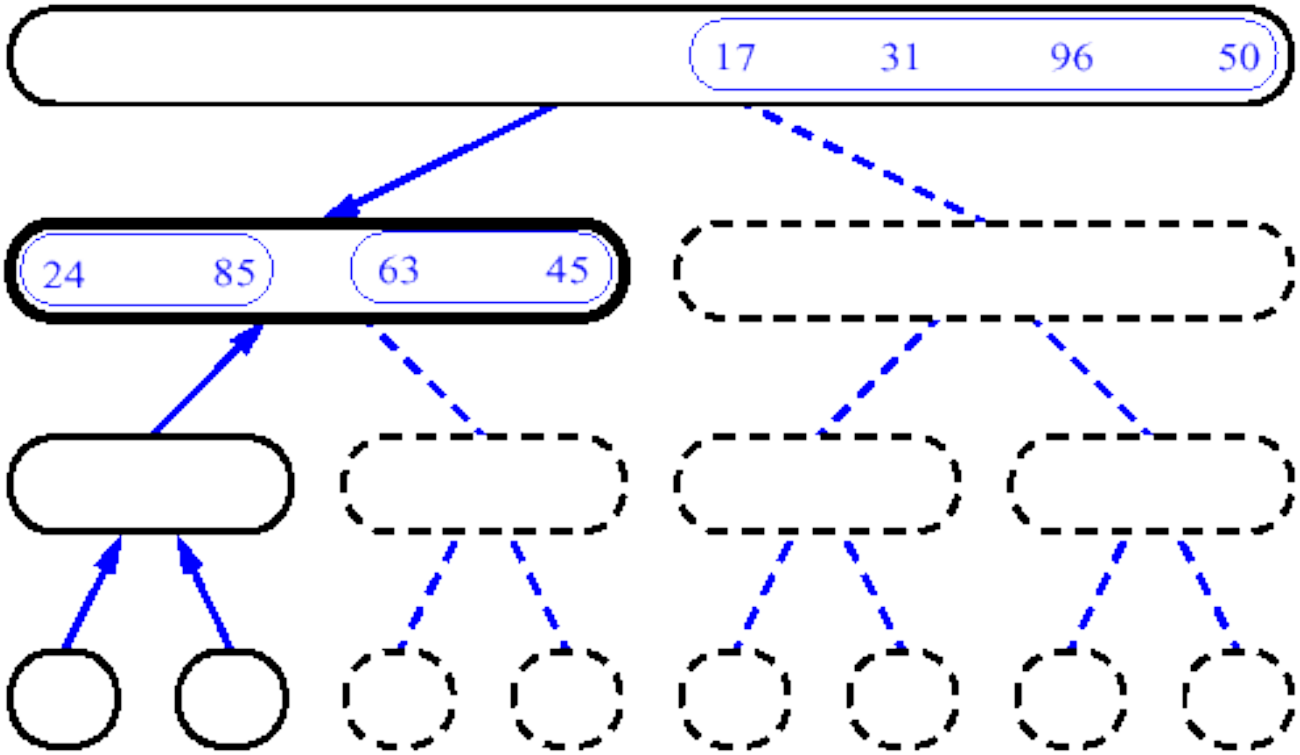
`MergeSort()` running on a array



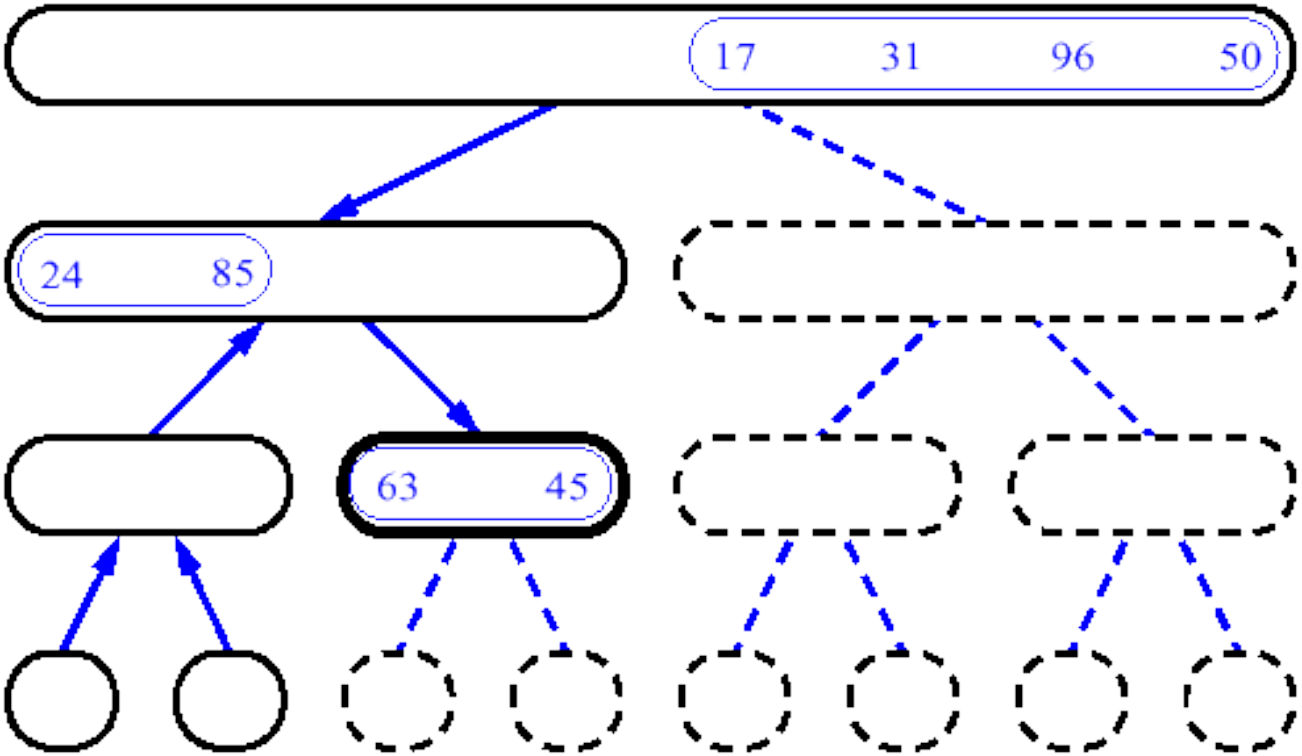
`MergeSort()` running on a array



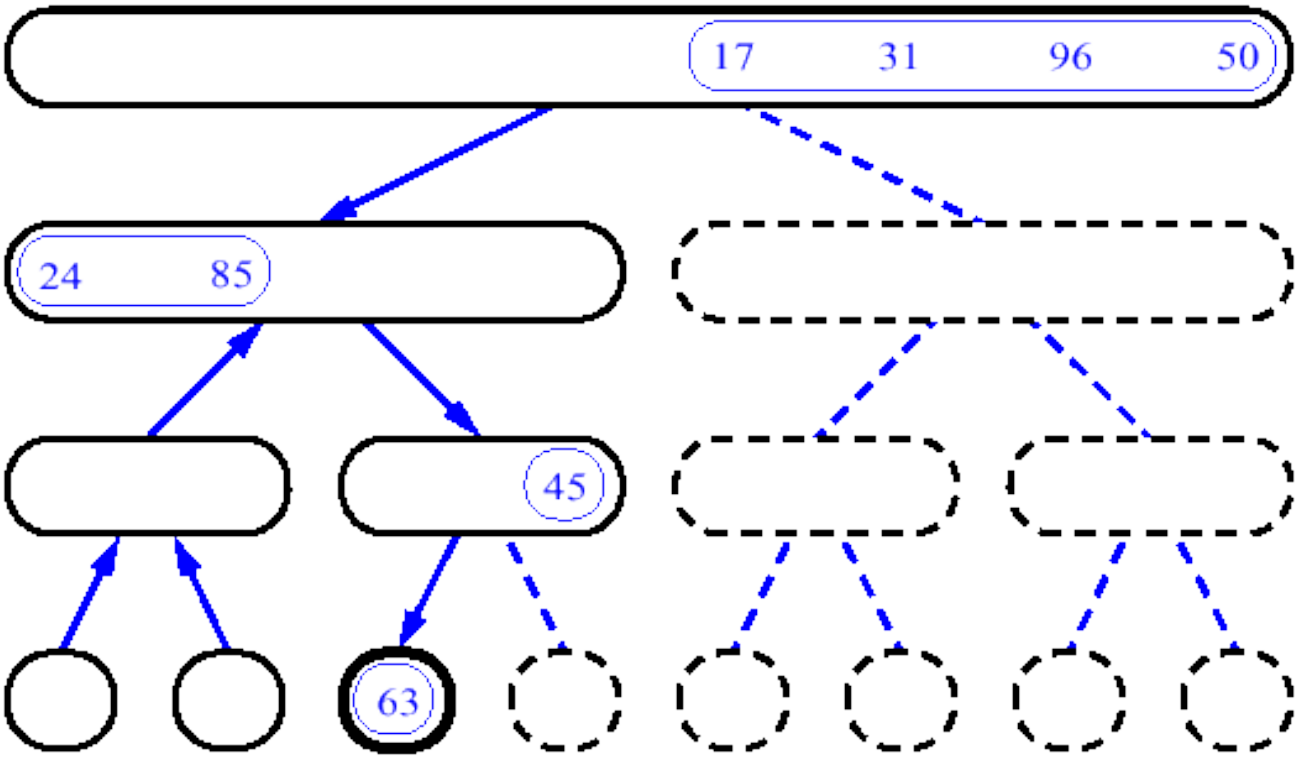
`MergeSort()` running on a array



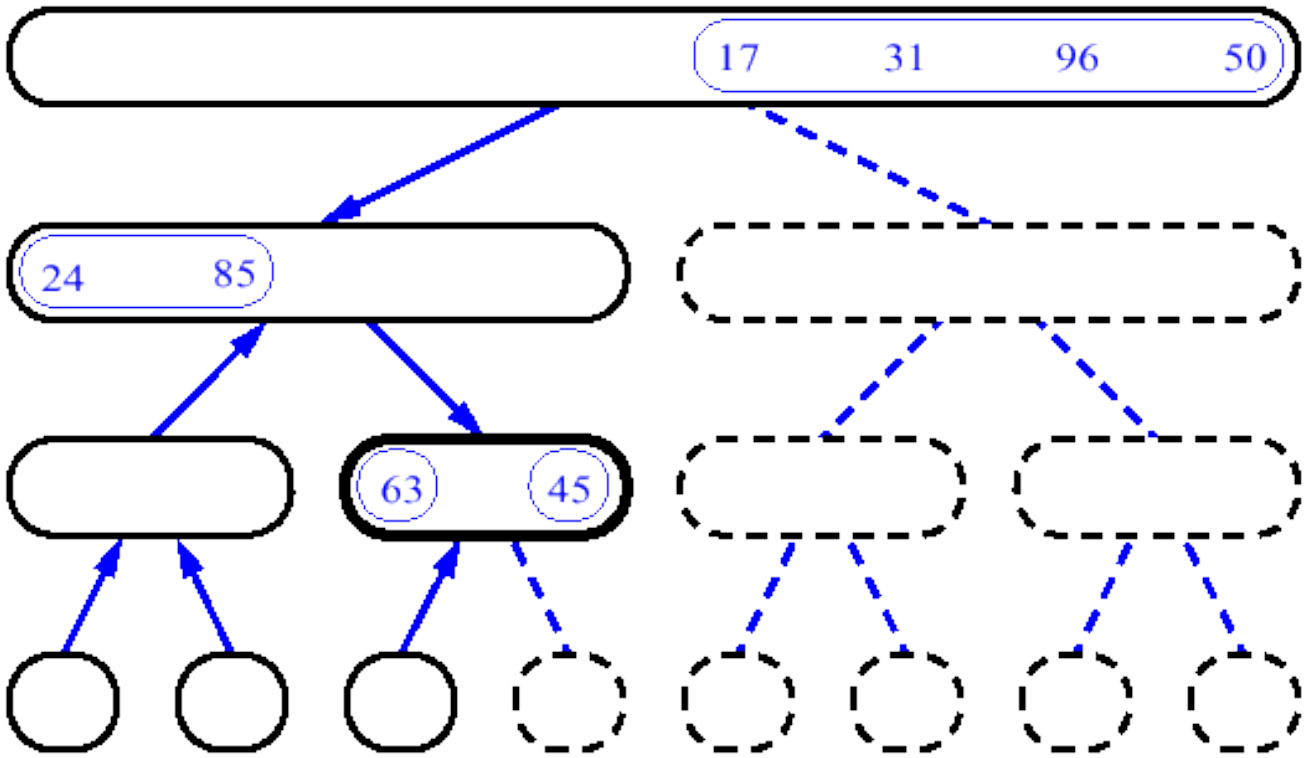
`MergeSort()` running on a array



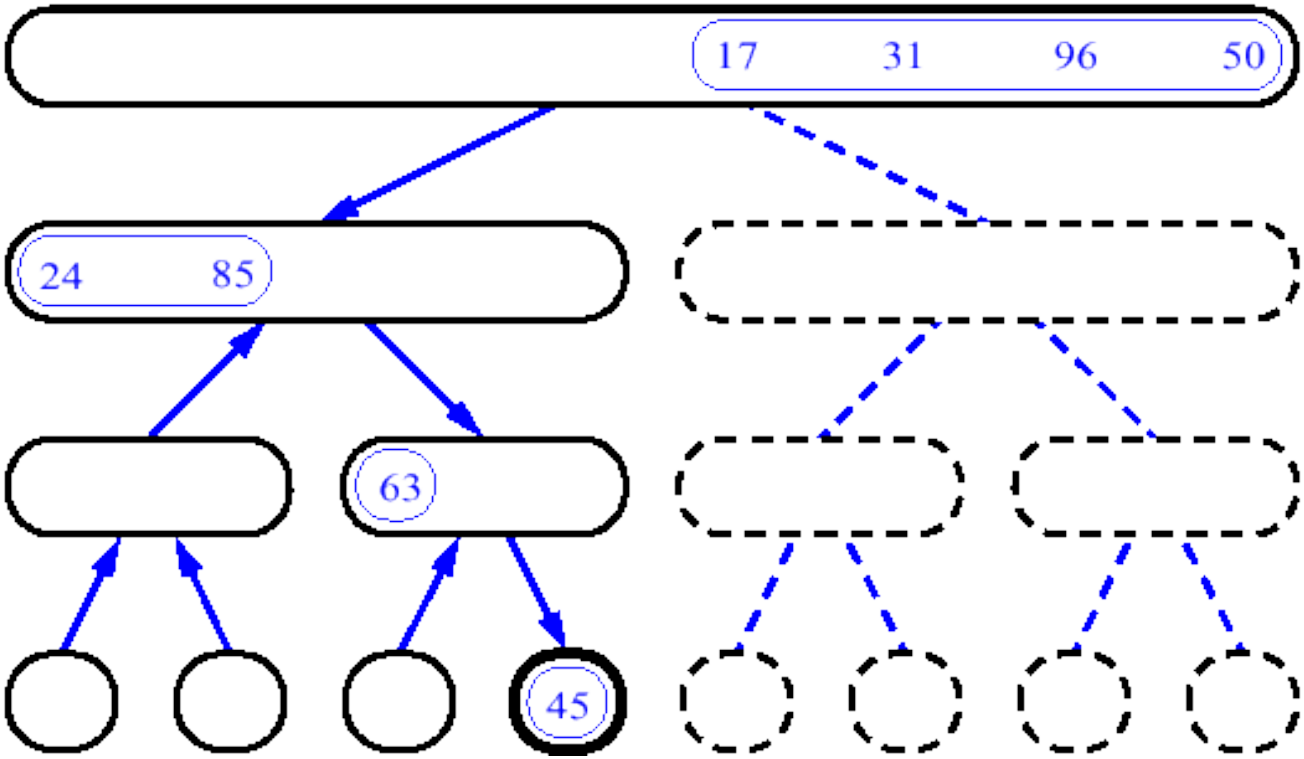
`MergeSort()` running on a array



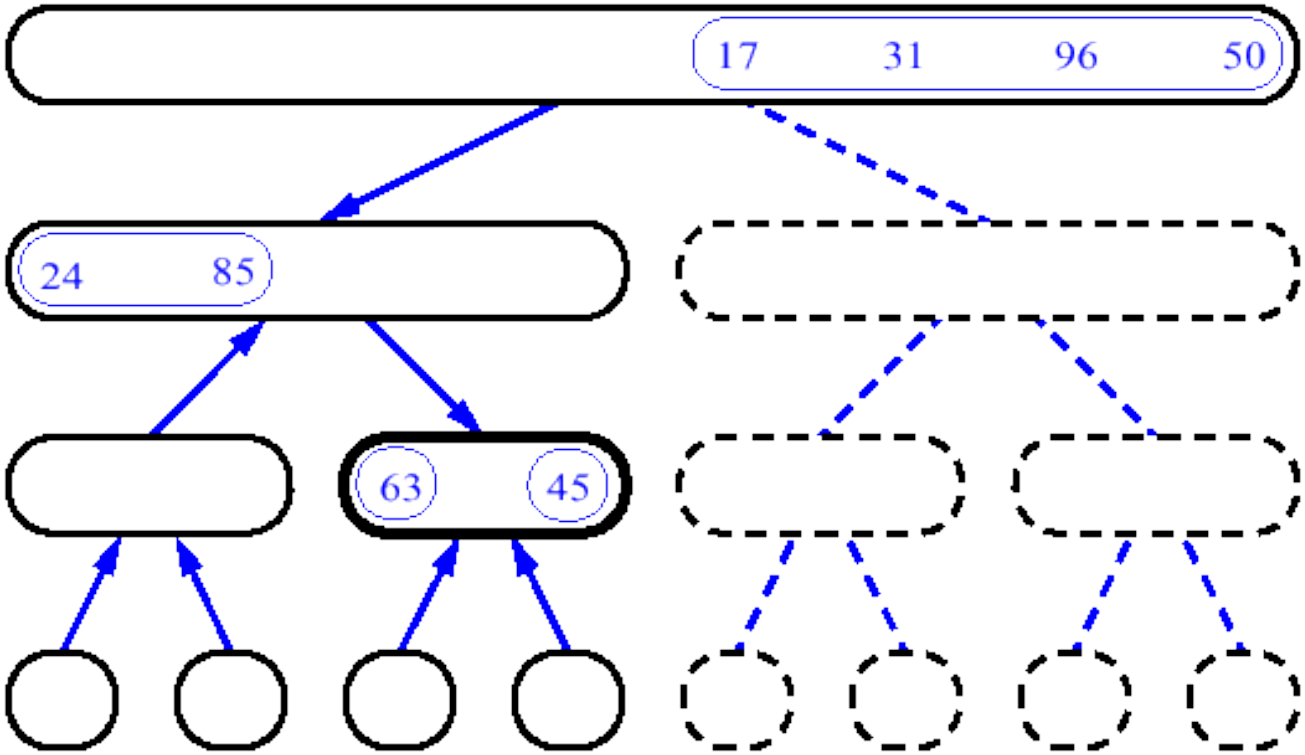
`MergeSort()` running on a array



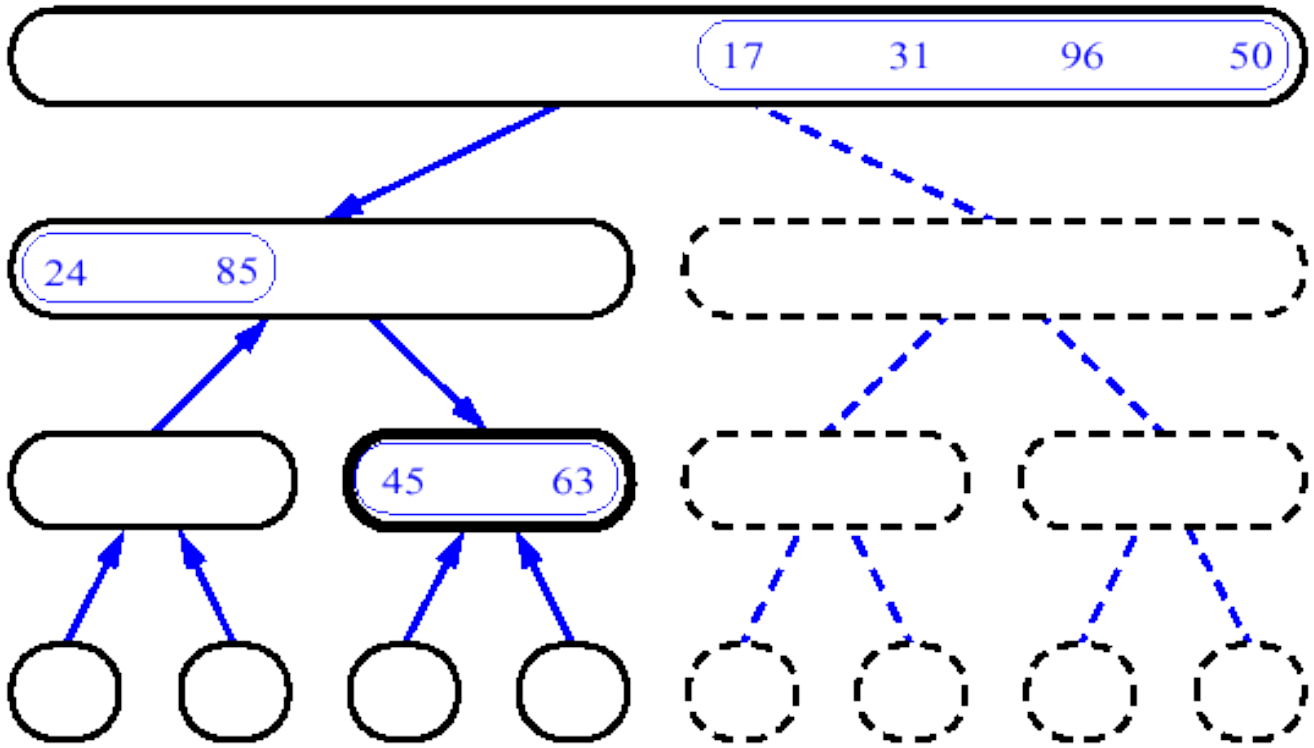
`MergeSort()` running on a array



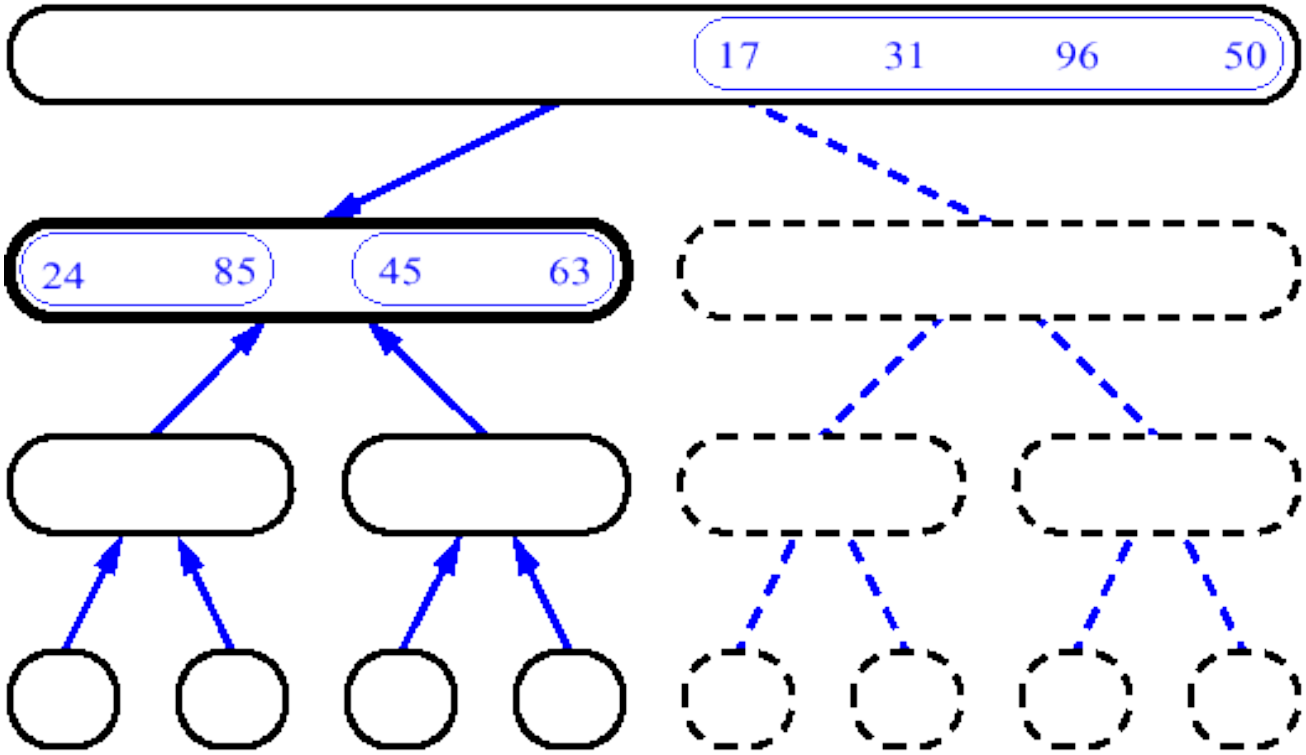
`MergeSort()` running on a array



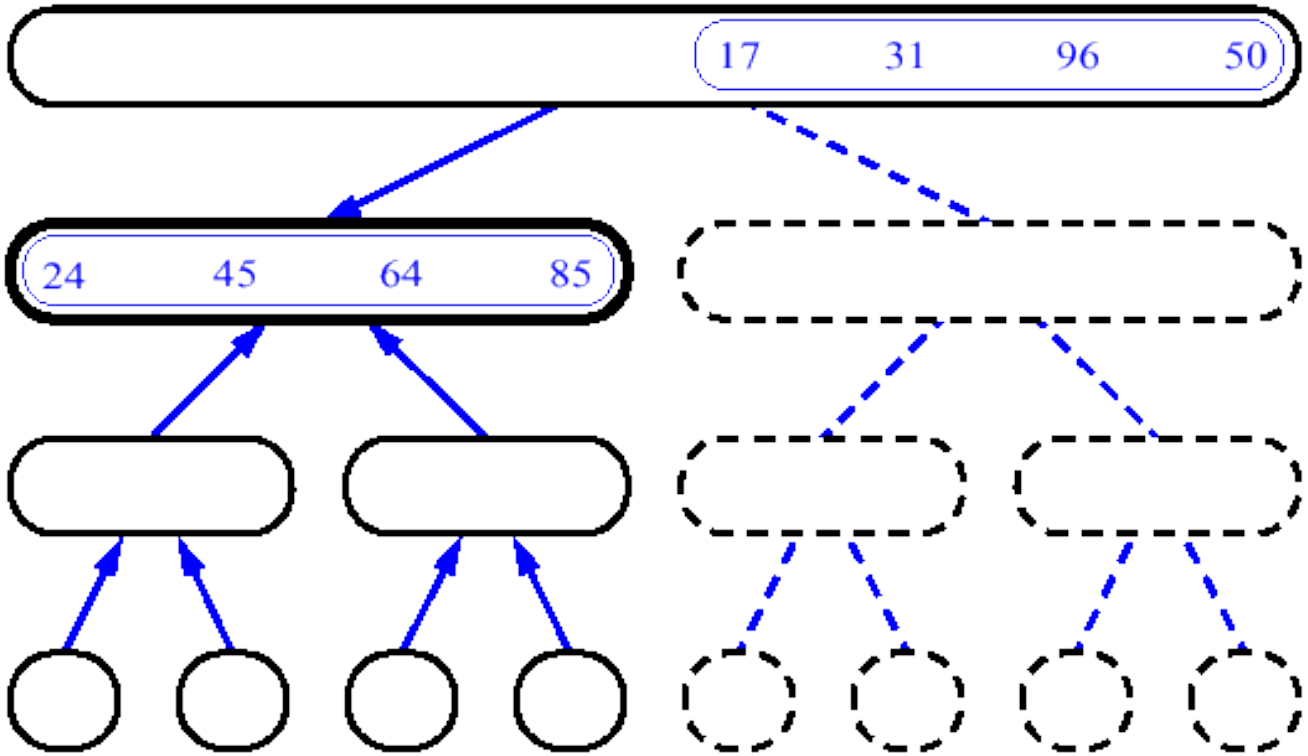
`MergeSort()` running on a array



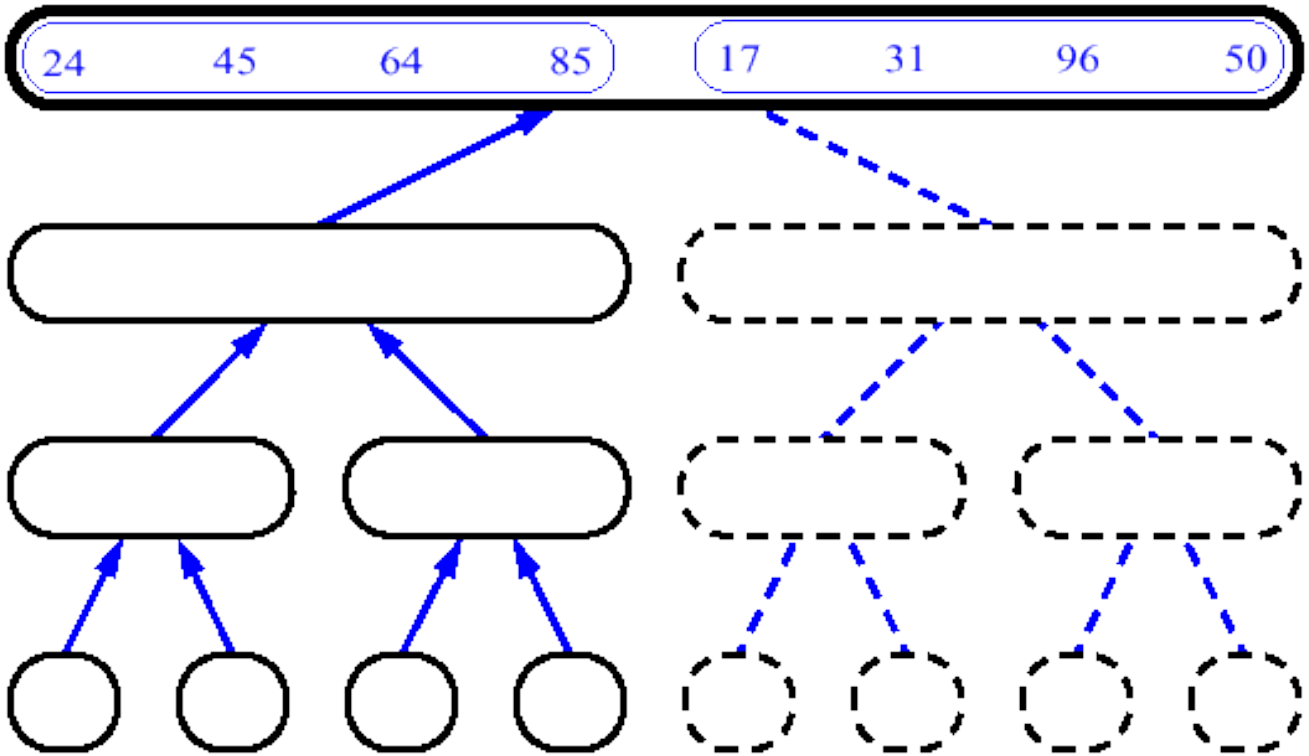
`MergeSort()` running on a array



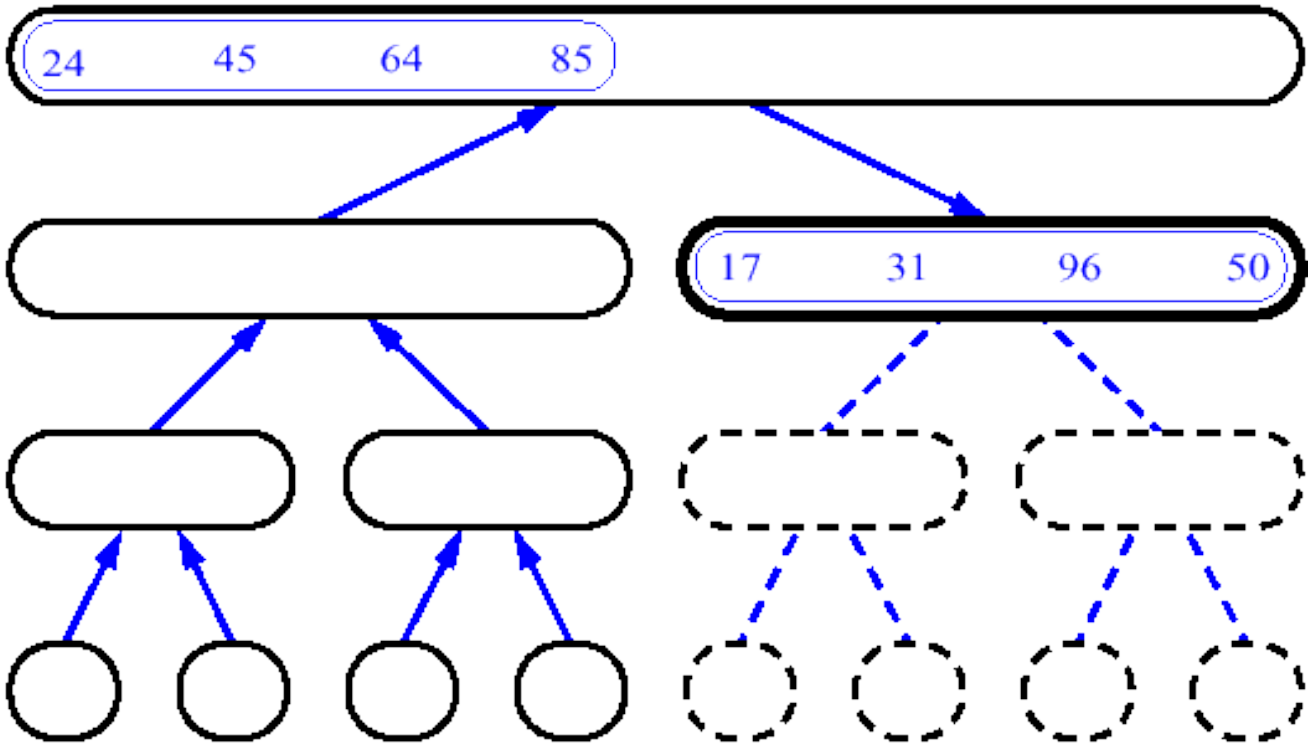
`MergeSort()` running on a array



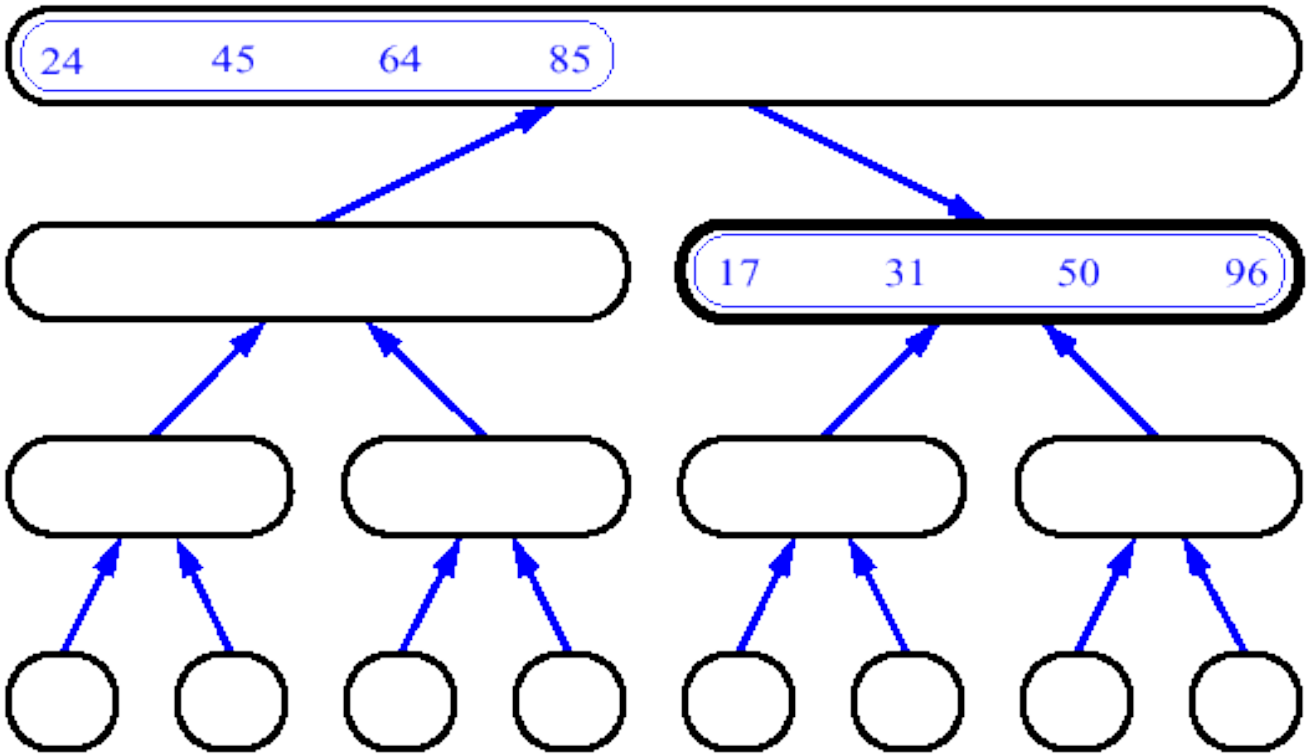
`MergeSort()` running on a array



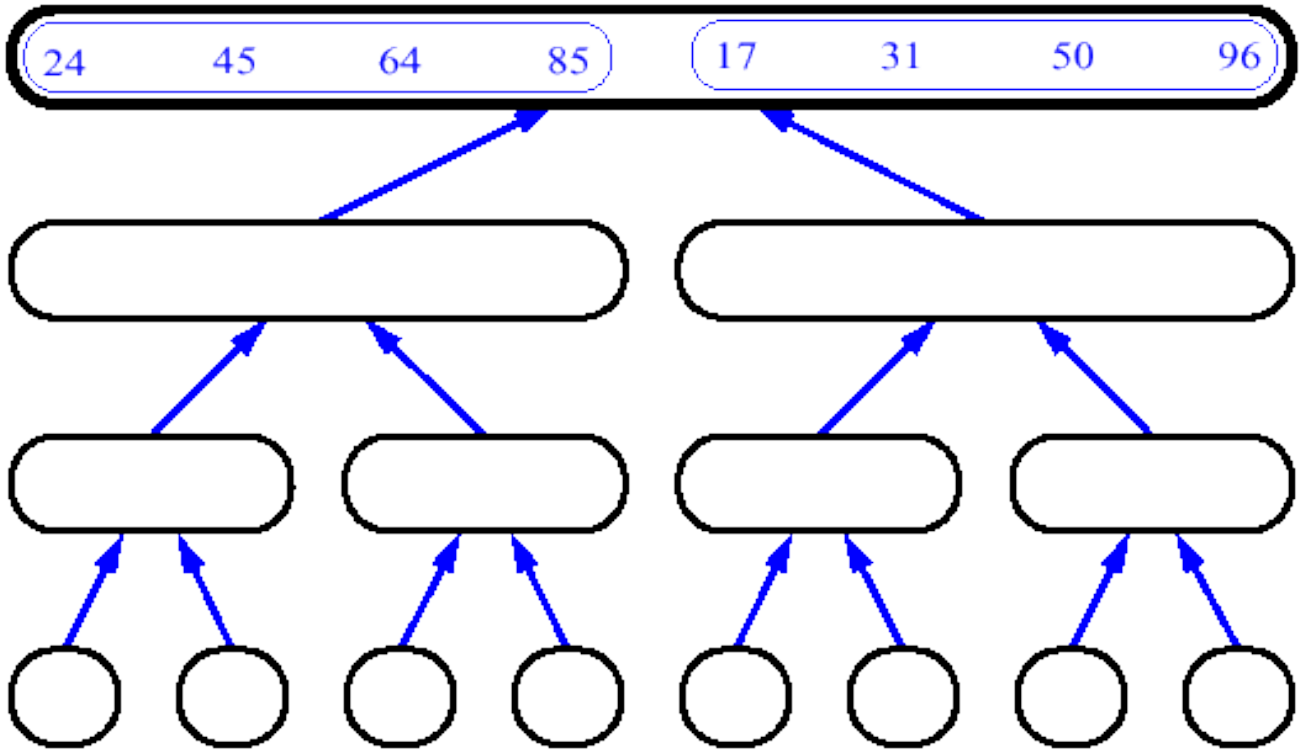
`MergeSort()` running on a array



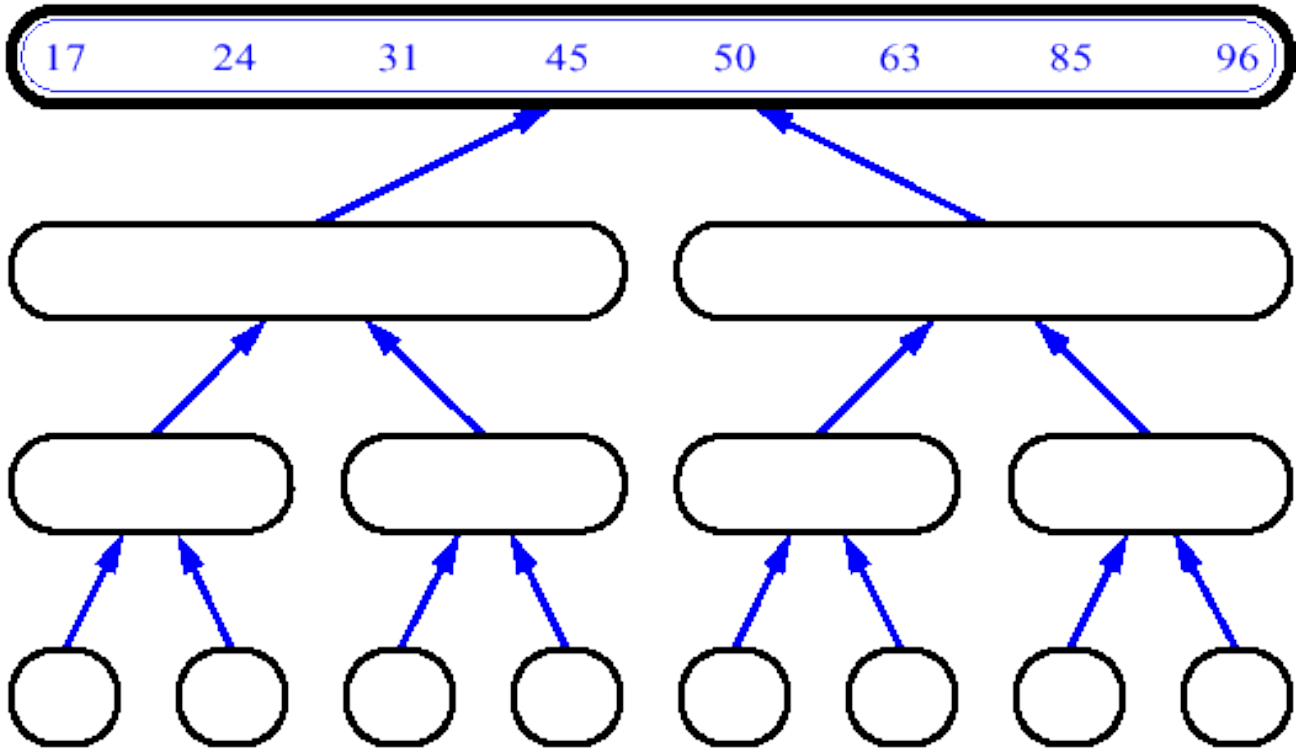
`MergeSort()` running on a array



`MergeSort()` running on a array



`MergeSort()` running on a array



Complexity Analysis of Merge Sort

Statement	Effort

```
MergeSort(A, left, right) {                          T(n)
    if (left < right) {
        mid = floor((left + right) / 2);
        MergeSort(A, left, mid);
        MergeSort(A, mid+1, right);
        Merge(A, left, mid, right);
    }
}
```

Complexity Analysis of Merge Sort

Statement	Effort

```
MergeSort(A, left, right) {                          T(n)
   if (left < right) {                               Θ(1)
      mid = floor((left + right) / 2);
      MergeSort(A, left, mid);
      MergeSort(A, mid+1, right);
      Merge(A, left, mid, right);
   }
}
```

Complexity Analysis of Merge Sort

Statement	Effort
`MergeSort(A, left, right) {`	`T(n)`
`  if (left < right) {`	`Θ(1)`
`    mid = floor((left + right) / 2);`	`Θ(1)`
`    MergeSort(A, left, mid);`	
`    MergeSort(A, mid+1, right);`	
`    Merge(A, left, mid, right);`	
`  }`	
`}`	

Complexity Analysis of Merge Sort

Statement	Effort
`MergeSort(A, left, right) {`	`T(n)`
`  if (left < right) {`	`Θ(1)`
`  mid = floor((left + right) / 2);`	`Θ(1)`
`  MergeSort(A, left, mid);`	`T(n/2)`
`  MergeSort(A, mid+1, right);`	
`  Merge(A, left, mid, right);`	
`  }`	
`}`	

Complexity Analysis of Merge Sort

Statement	Effort
`MergeSort(A, left, right) {`	`T(n)`
`  if (left < right) {`	`Θ(1)`
`  mid = floor((left + right) / 2);`	`Θ(1)`
`  MergeSort(A, left, mid);`	`T(n/2)`
`  MergeSort(A, mid+1, right);`	`T(n/2)`
`  Merge(A, left, mid, right);`	
`  }`	
`}`	

Complexity Analysis of Merge Sort

Statement	Effort
`MergeSort(A, left, right) {`	`T(n)`
`  if (left < right) {`	`Θ(1)`
`    mid = floor((left + right) / 2);`	`Θ(1)`
`    MergeSort(A, left, mid);`	`T(n/2)`
`    MergeSort(A, mid+1, right);`	`T(n/2)`
`    Merge(A, left, mid, right);`	`Θ(n)`
`  }`	
`}`	

Complexity Analysis of Merge Sort

Statement	Effort

```
MergeSort(A, left, right) {                          T(n)
   if (left < right) {                               Θ(1)
      mid = floor((left + right) / 2);               Θ(1)
      MergeSort(A, left, mid);                       T(n/2)
      MergeSort(A, mid+1, right);                    T(n/2)
      Merge(A, left, mid, right);                    Θ(n)
   }
}
```

- So T(n) = $\Theta(1)$ when n = 1, and
  $2T(n/2) + \Theta(n)$ when n > 1

Complexity Analysis of Merge Sort

Statement	Effort

```
MergeSort(A, left, right) {                          T(n)
   if (left < right) {                               Θ(1)
      mid = floor((left + right) / 2);               Θ(1)
      MergeSort(A, left, mid);                       T(n/2)
      MergeSort(A, mid+1, right);                    T(n/2)
      Merge(A, left, mid, right);                    Θ(n)
   }
}
```

- So T(n) = $\Theta(1)$ when n = 1, and
  $2T(n/2) + \Theta(n)$ when n > 1
- So what (more succinctly) is T(n)?

Recurrences

- The expression that represents the **merge sort**:

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$

- is a ***recurrence***.

Recurrences

- The expression that represents the **merge sort**:

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$

- is a ***recurrence***.
  - Recurrence: an **equation** or **inequality** that describes a function in terms of its **value on smaller functions**

Recurrences: Factorial

- What is the **recurrence equation** for this algorithm?

```
fac(n) is
   if n = 1 then return 1
   else return n * fac(n-1)
```

- A recurrence defines **T(n)** in terms of **T** for smaller values

Recurrences: Factorial

- What is the **recurrence equation** for this algorithm?

```
fac(n) is
   if n = 1 then return 1
   else return n * fac(n-1)
```

- A recurrence defines **T(n)** in terms of **T** for smaller values

$T(n) = c$ if $n=1$

Recurrences: Factorial

- What is the **recurrence equation** for this algorithm?

```
fac(n) is
   if n = 1 then return 1
   else return n * fac(n-1)
```

- A recurrence defines **T(n)** in terms of **T** for smaller values

$$T(n) = c \quad \text{if } n=1$$
$$T(n) = T(n-1) + c \quad \text{if } n>1$$

Recurrences: Binary Search

- What is the **recurrence equation** for this algorithm?

```
BinSearch(A[1…n],q)
    if n=1
       then if A[n]=q then return n
                      else return 0

     k←(n+1)/2
     if q < A[k] then BinSearch(A[1…k-1],q)
                 else BinSearch(A[k…n],q)
```

Recurrences: Binary Search

- What is the **recurrence equation** for this algorithm?

```
BinSearch(A[1…n],q)
    if n=1
       then if A[n]=q then return n
                      else return 0

     k←(n+1)/2
     if q < A[k] then BinSearch(A[1…k-1],q)
                 else BinSearch(A[k…n],q)
```

$$T(n) = c \qquad \text{if } n=1$$

Recurrences: Binary Search

- What is the **recurrence equation** for this algorithm?

```
BinSearch(A[1…n],q)
    if n=1
       then if A[n]=q then return n
                      else return 0

     k←(n+1)/2
     if q < A[k] then BinSearch(A[1…k-1],q)
                 else BinSearch(A[k…n],q)
```

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

Other Recurrence Examples

$$s(n) = \begin{cases} 0 & n = 0 \\ c + s(n-1) & n > 0 \end{cases}$$

$$s(n) = \begin{cases} 0 & n = 0 \\ n + s(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

Solving Recurrences

- Substitution method
- Iteration method
- Master method