Divide and Conquer Technique

Presentation by:
S.V.Jansi Rani
AP/CSE
SSN College of Engineering

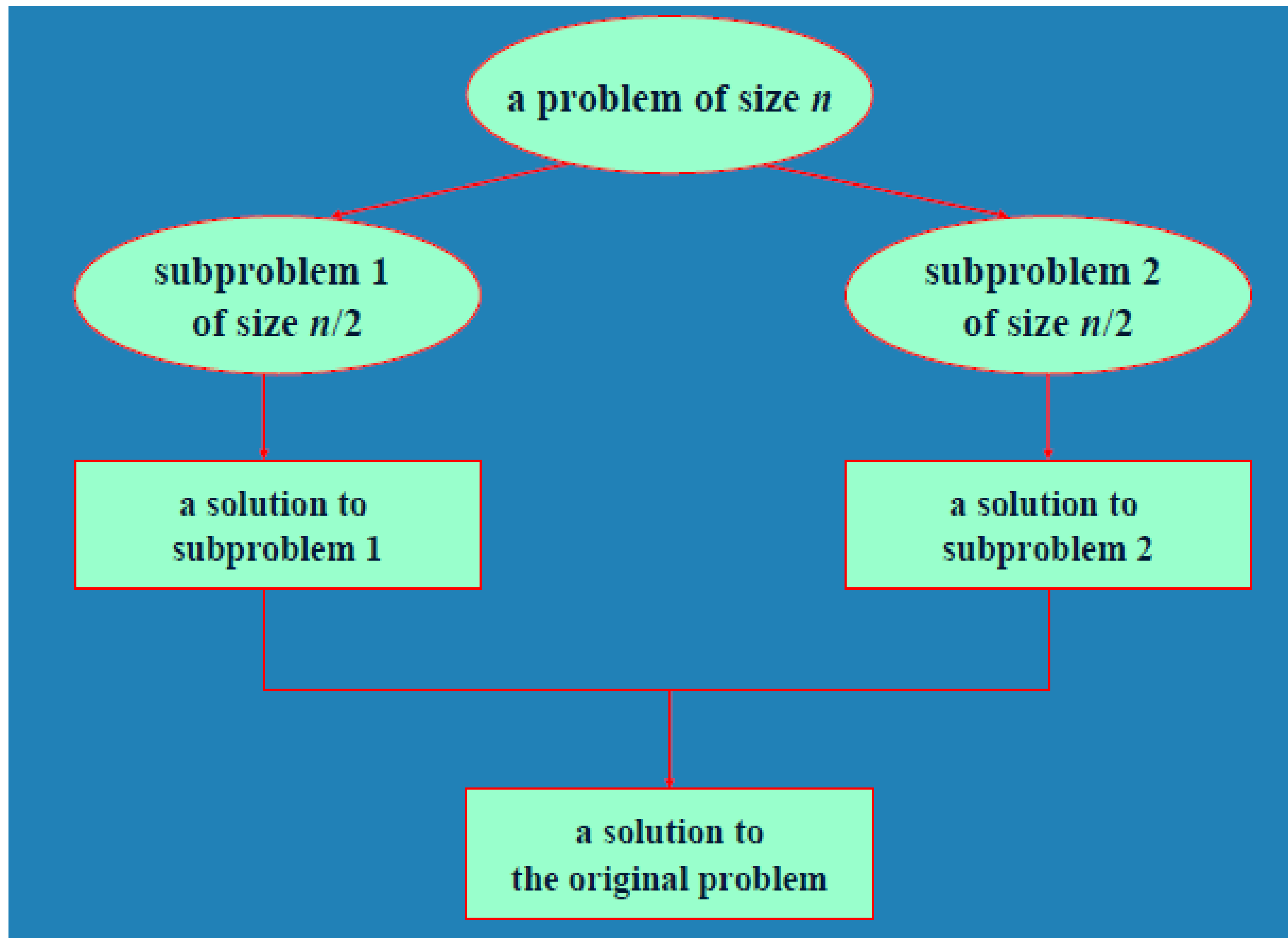
Outline

- Divide and Conquer Method
- Merge Sort
- Quick Sort
- Binary Search

Divide and Conquer Method

- Patterned after the strategy employed by Napoleon
- Divide an instance of a problem recursively into two or more smaller instances until the solutions to the small instances are obtainable.
- Top-down approach used by recursive routines

Divide and Conquer Method



```
Algorithm DAndC(P)
{
if Small(P) then return S(P);
else
{
divide P into smaller instances P1, P2, …,Pk, K≥1;
Apply DAndC to each of these sub problems;
return Combine(DAndC(p1), DAndC(p2),… DAndC(pk));
}
}
```

The Master Theorem

The recurrence $T(n) = aT\left(\frac{n}{b}\right) + n^c$ has the solution

$$T(n) = \begin{cases} \Theta(n^c) & a < b^c \\ \Theta(n^c \log n) & a = b^c \\ \Theta(n^{\log_b a}) & a > b^c \end{cases}$$

MERGE SORT

Merge Sort

Basic Strategy:

1.Divide the array A[1 .. n] into two sub arrays A[1 ..m] and A[m+1 .. n], where m = floor(n/2)

2. Recursively mergesort the sub arrays A[1 ..m] and A[m+1 .. n].

3. Merge the newly-sorted sub arrays A[1 ..m] and A[m+1 .. n] into a single sorted list.

- The first step is completely trivial—we only need to compute the median index m—and we can delegate the second step to the Recursive sub routine

- All the real work is done in the final step; the two sorted sub arrays A[1 ..m] and A[m+1 .. n] can be merged using a simple linear-time algorithm.

- For simplicity, we separate out the merge step as a subroutine

Example

[Example]

ALGORITHM

```
MERGESORT(A[], l,h):
If (l<h)
  mid =(l+h)/2
  MERGESORT(l,mid)
  MERGESORT(mid+1,h)
  Merge(A,l,mid,h)
Else
  return
```

```
Merge(A[],l,mid,h)
 ht=l, j = mid+1, i=1, k
while ((ht<=mid)&&(j<=h))
    { if(A[ht]<=A[j])
            B[i] = A[ht]; ht++;
      else
            B[i] = A[j]; j++;
     i++;
    }
if(ht>md) /* copy remaining second
    array elements
  for(k=j; k<=h; k++)
    B[i] =A[k];  i++;
else
for(k=ht; k<=md; k++)
    B[i] =A[k];  i++;
for(k=l; k<=h; k++)
    A[k] =B[k];
```

TRACE ?

Complexity

- Input size (list length) to the original MergeSort call is n. Input size to each recursive call is n
- If the running time of the original call is T(n), then the running time each recursive call is T(n/2)
- Each call to MergeSort makes two recursive calls.
- Each call to MergeSort calls Merge. Merge scans the items of the input list once. Therefore, the running time of Merge is n.
- Running time of a MergeSort call is

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n$$

- When the input size is 1 or 0, MergeSort sorts the list trivially. Therefore, the best case is

T(n) = 1

The running time of MergeSort is given by the recurrence relation

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

Assume that n is a power of 2, say, $n = 2^h$, and hence, $h = \log_2 n$. This will keep our analysis simple but general enough. We will solve it by unrolling.

Use Master's theorm and find out the order ??????????????

$$
\begin{aligned}
T(n) &= n + 2T\left(\frac{n}{2}\right) \\
&= n + 2\left[\frac{n}{2} + 2T\left(\frac{n}{2^2}\right)\right] \\
&= n + n + 2^2 T\left(\frac{n}{2^2}\right) \\
&= 2n + 2^2\left[\frac{n}{2^2} + 2T\left(\frac{n}{2^3}\right)\right] \\
&= 2n + n + 2^3 T\left(\frac{n}{2^3}\right) \\
&\dots \\
&= ni + 2^i T\left(\frac{n}{2^i}\right) \\
&\dots \\
&= nh + 2^h T\left(\frac{n}{2^h}\right) \\
&= nh + 2^h T(1) \\
&= n \log_2 n + n
\end{aligned}
$$

Quick Sort

ALGORITHM

```
QUICKSORT(A[1 .. n]):
If (n > 1)
  Choose a pivot element
  A[p]
  k = PARTITION(A, p)
  QUICKSORT(A[1 .. k -1])
  QUICKSORT(A[k+1 … n])
Else
    return
```

```
PARTITION(A[1 .. n], p):
if (p <> n)
    swap A[p] and A[n]
    i = 0; j=n
while (i < j)
    repeat i = i +1 until (i = j or A[i] >= A[n])
    repeat j = j-1 until (i = j or A[ j] <=A[n])
    if (i < j)
        swap A[i] and A[ j]
if (I <> n)
    swap A[i] and A[n]
return i
```

Example

Time Complexity of Quick Sort

PARTITION runs in O(n) time: j - i = n at the beginning,

j - i = 0 at the end, and we do a constant amount of work each time we increment i or decrement j.

For QUICKSORT, we get a recurrence that depends on k, the rank of the chosen pivot:

$$T(n) = T(k - 1) + T(n - k) + O(n)$$

<u>Best Case:</u>

If we could choose the pivot to be the median element of the array A,

we would have k = T(n/2), the two subproblems would be as close to the same size as possible, the recurrence would become

$$T(n) = 2T(n/2) + n = O(n \log n)$$

Worst Case: Sorted array as input

- The worst-case is when the pivot always ends up in the first or last element. That is, partitions the array as unequally as possible.
- In this case
  - T(n) = T(n–1) + T(1–1) + n = T(n–1) + n

    = n + (n–1) + … + 1

    = n(n + 1)/2 ⇒ O(n2)

Average case : randomly ordered array of size n

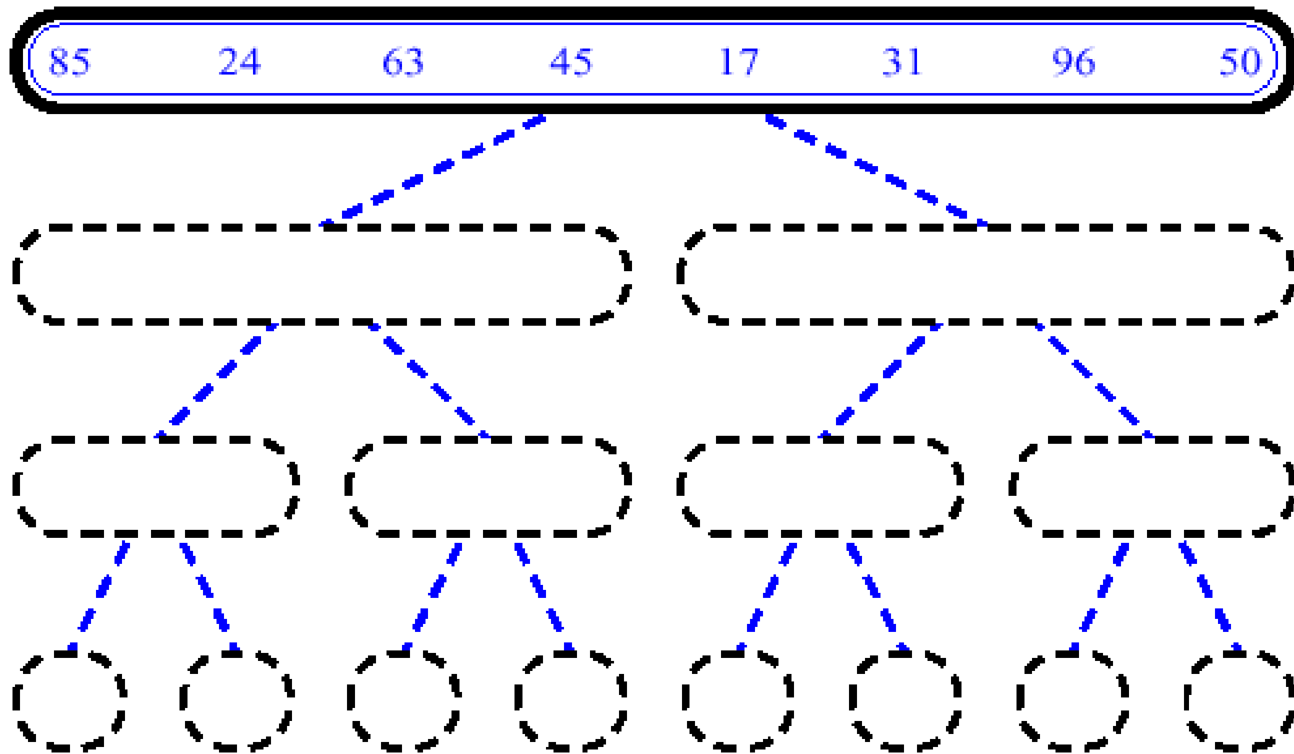
- It is rather complex, but is where the algorithm earns its name.
- Assume the partition split can happen in each position s(0 <= s <= n-1)

$$T(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n + T(s) + T(n-1-s)]$$ for n>1

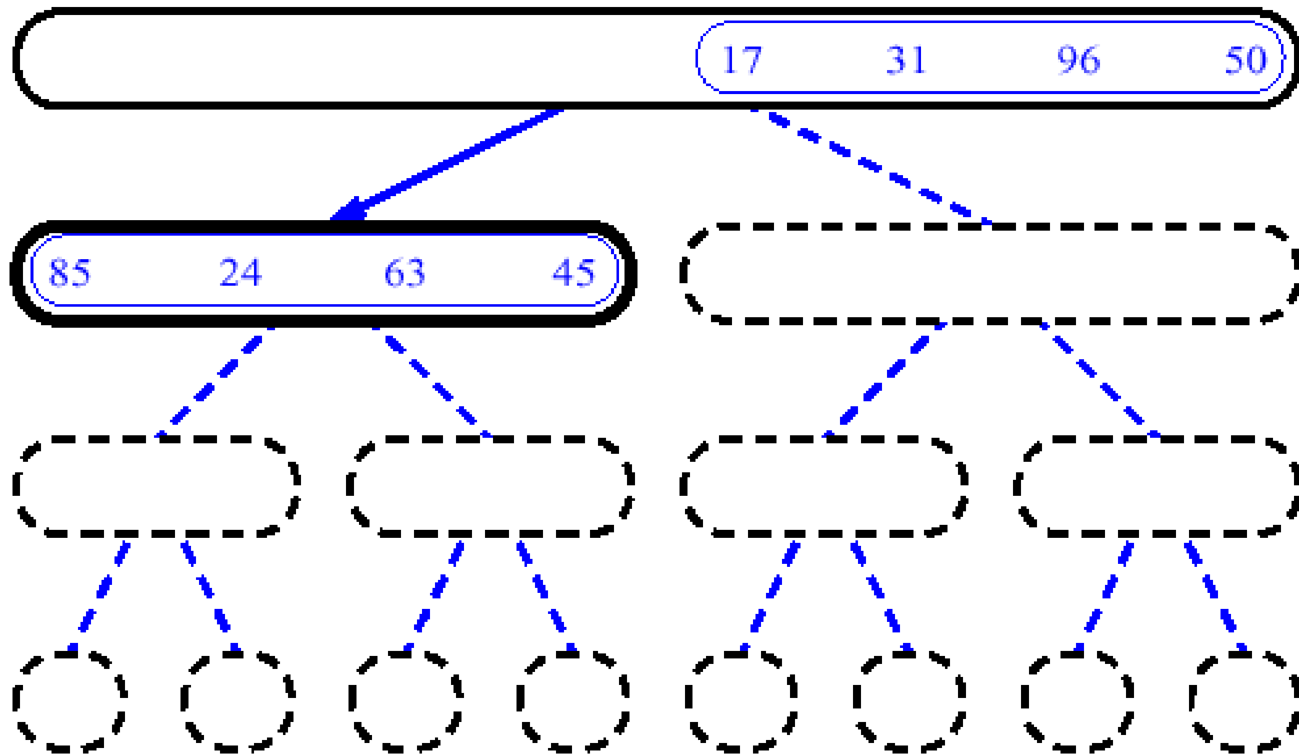
T(0) =0 , T(1) =0

T(n) ≈ 2nln n ≈ 1.38 n log n

MergeSort (Example) - 1

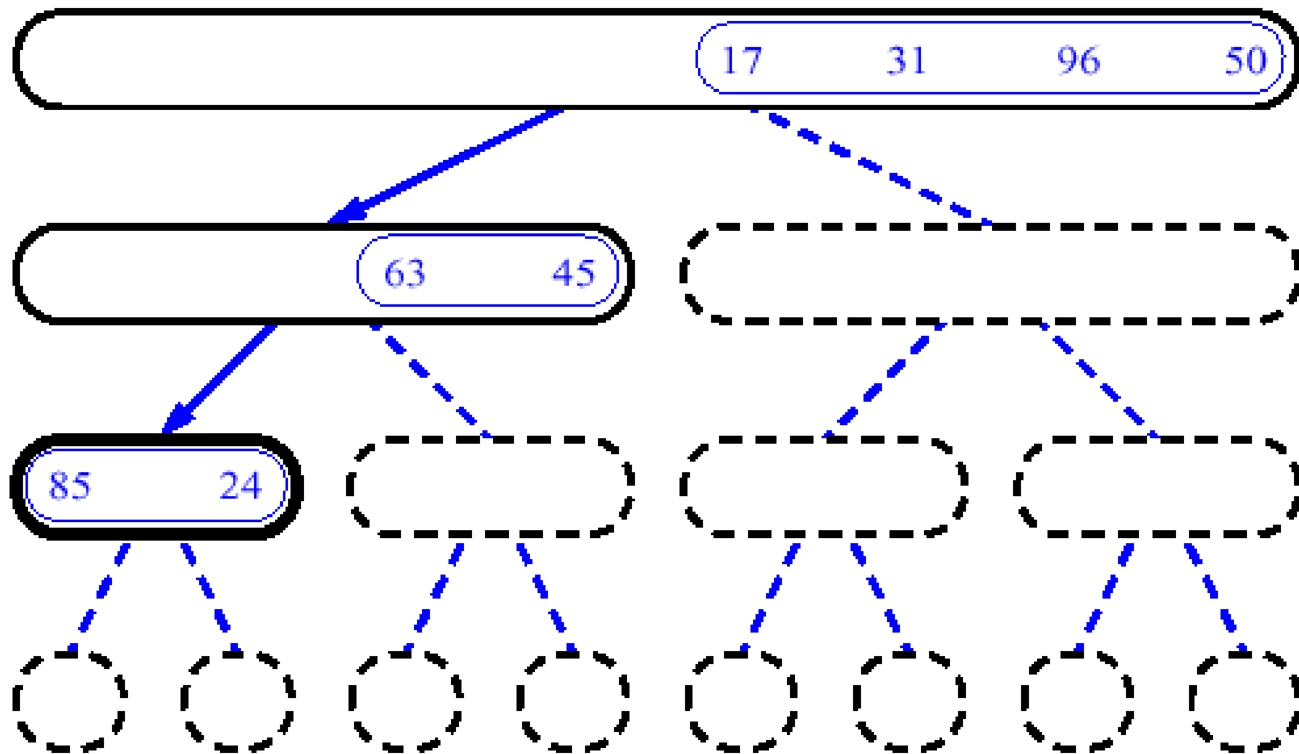


MergeSort (Example) - 2

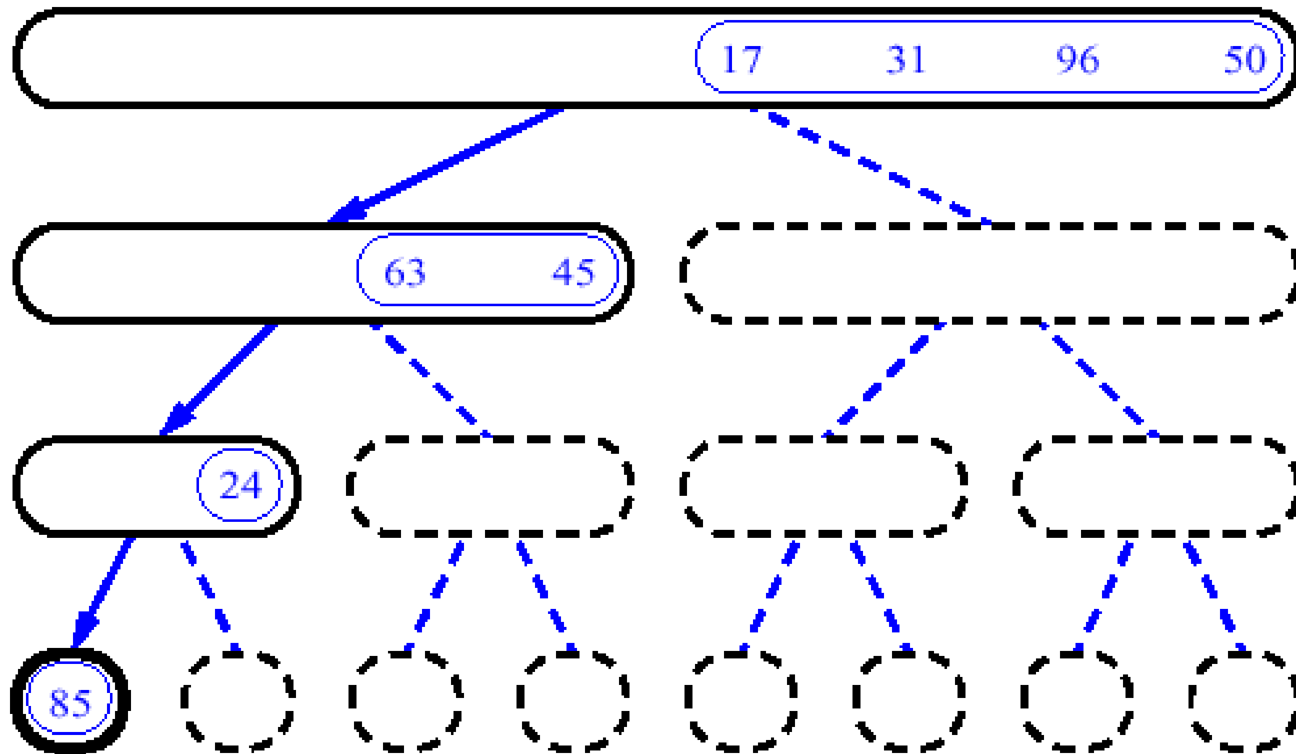


SSN

MergeSort (Example) - 3

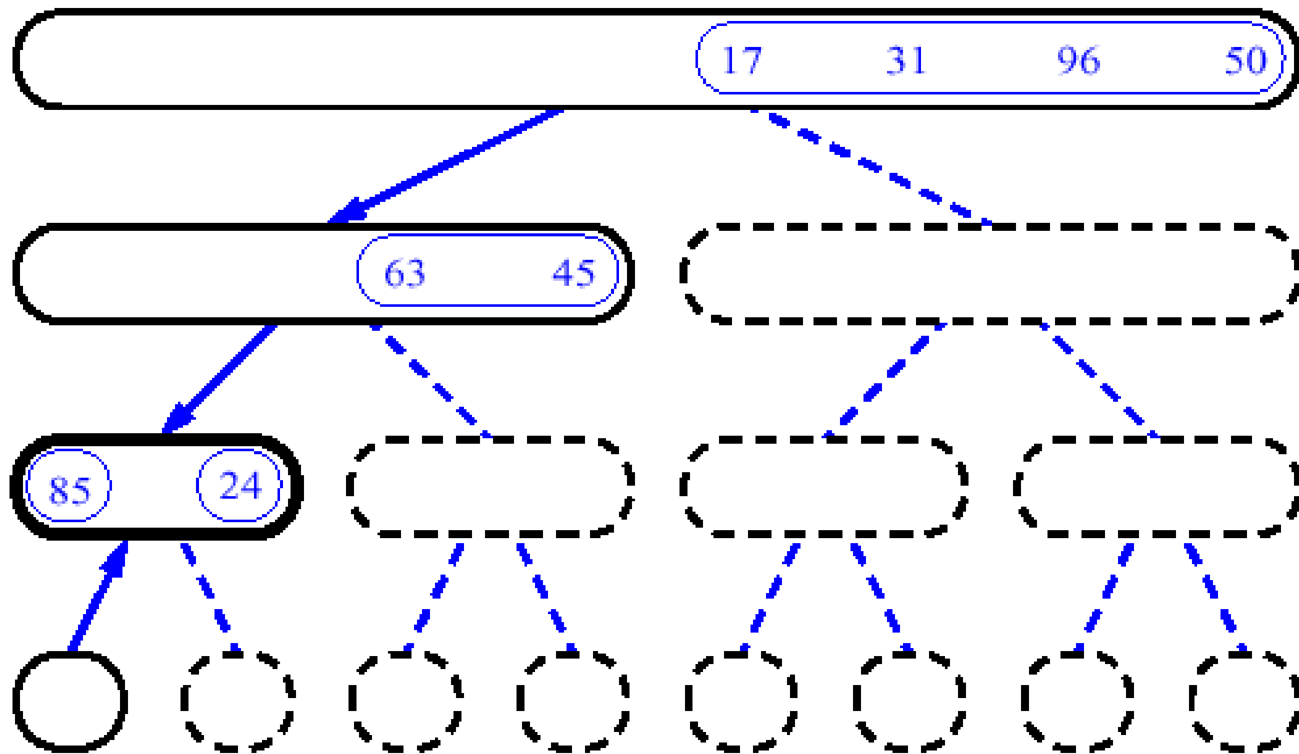


MergeSort (Example) - 4



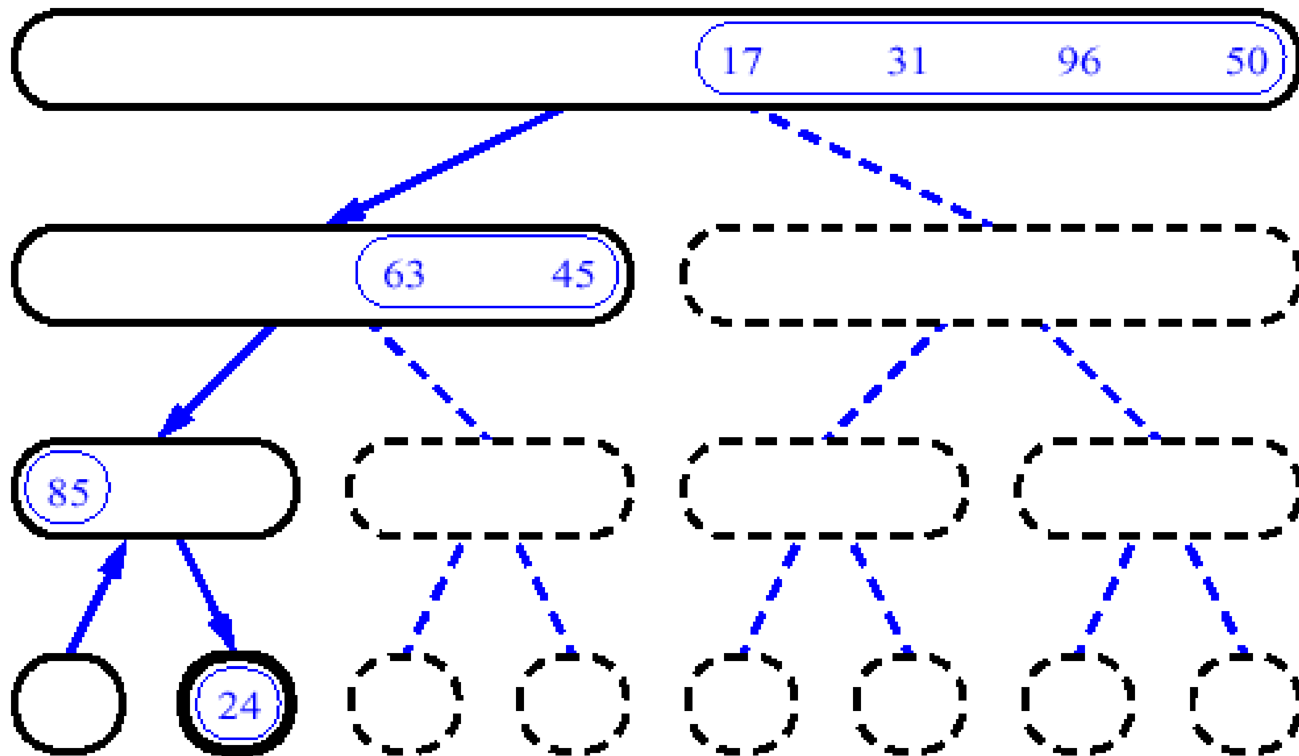
SSN

MergeSort (Example) - 5

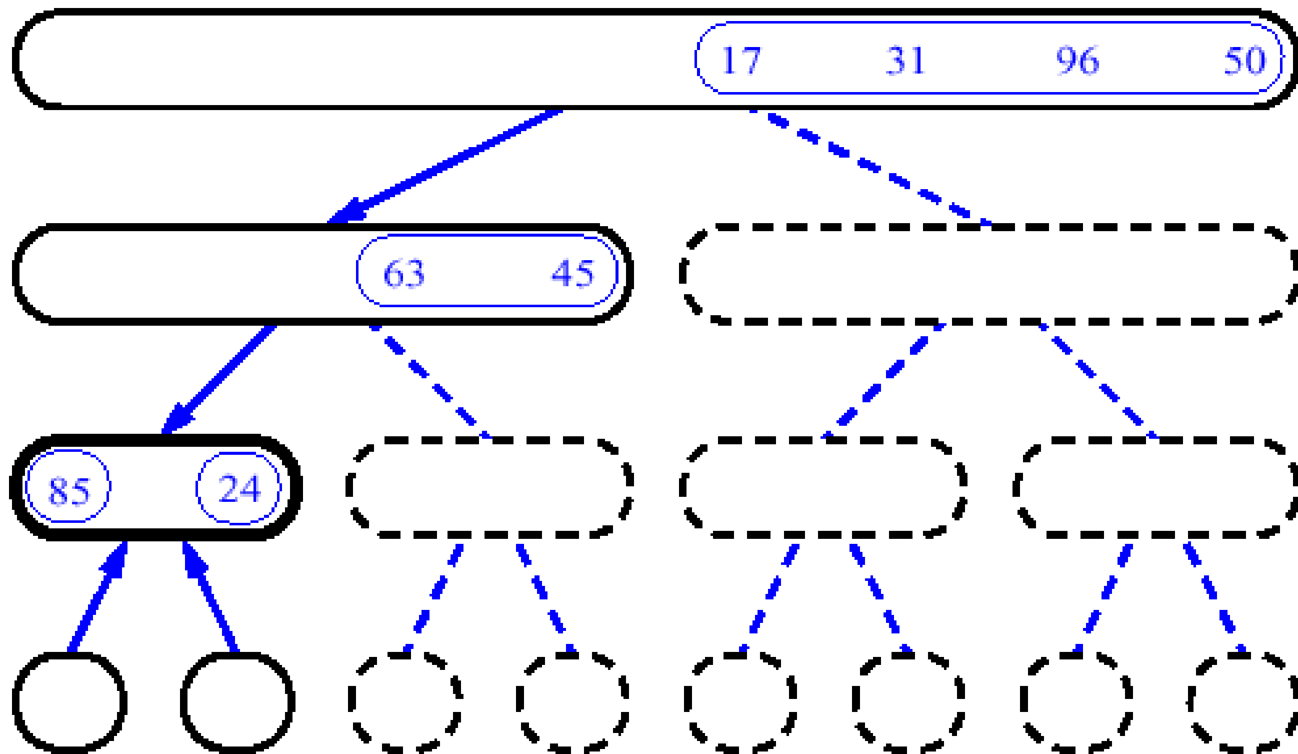


SSN

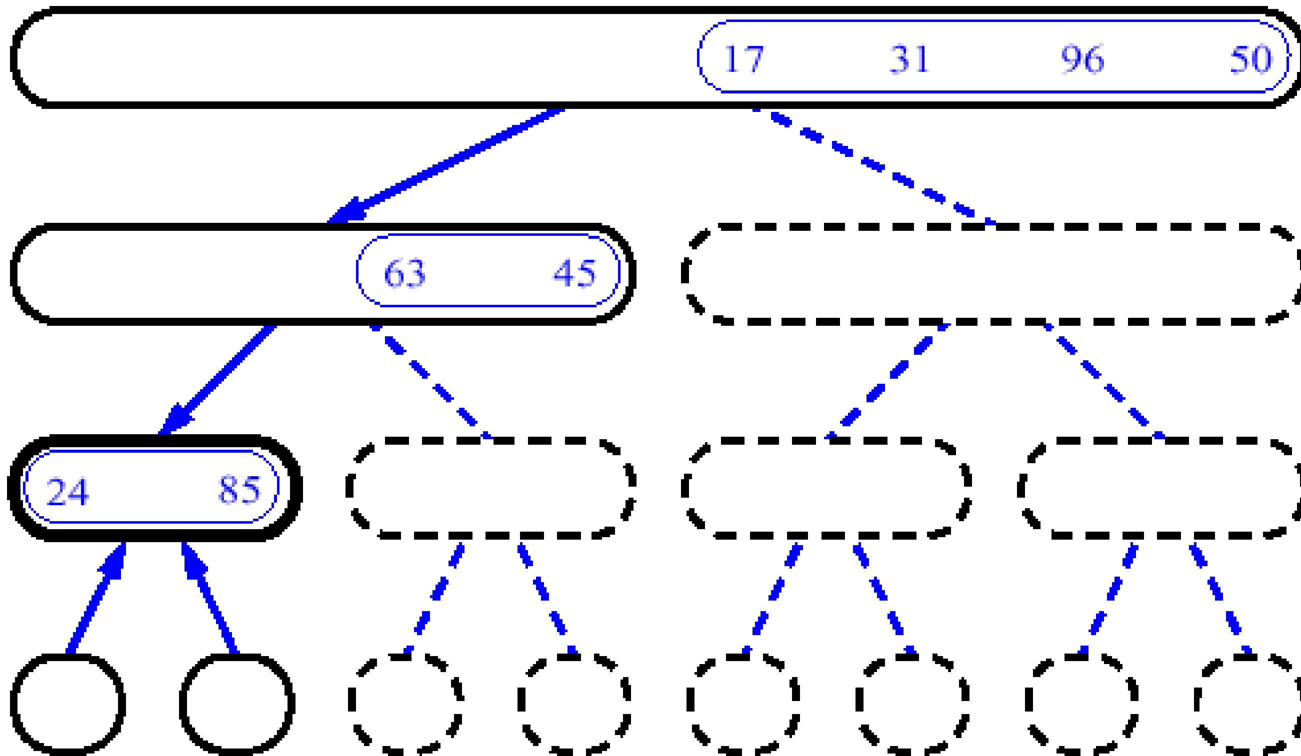
MergeSort (Example) - 6



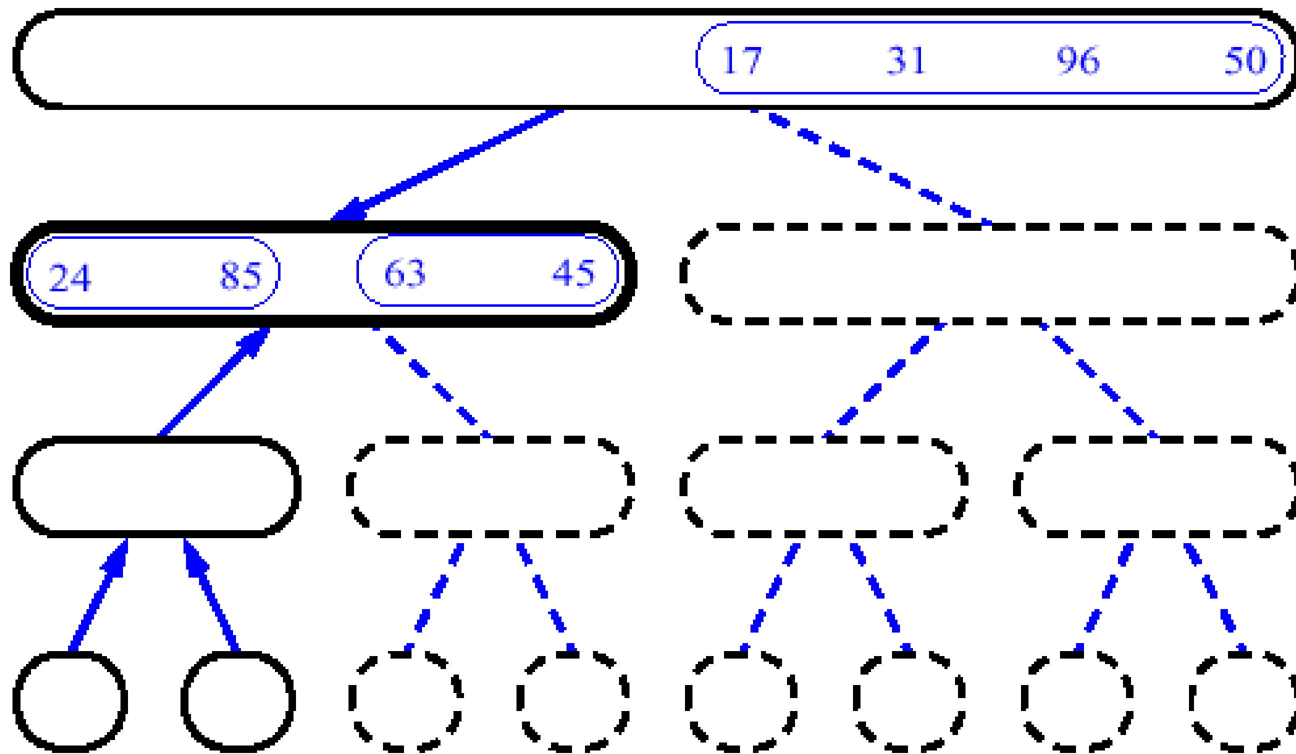
MergeSort (Example) - 7



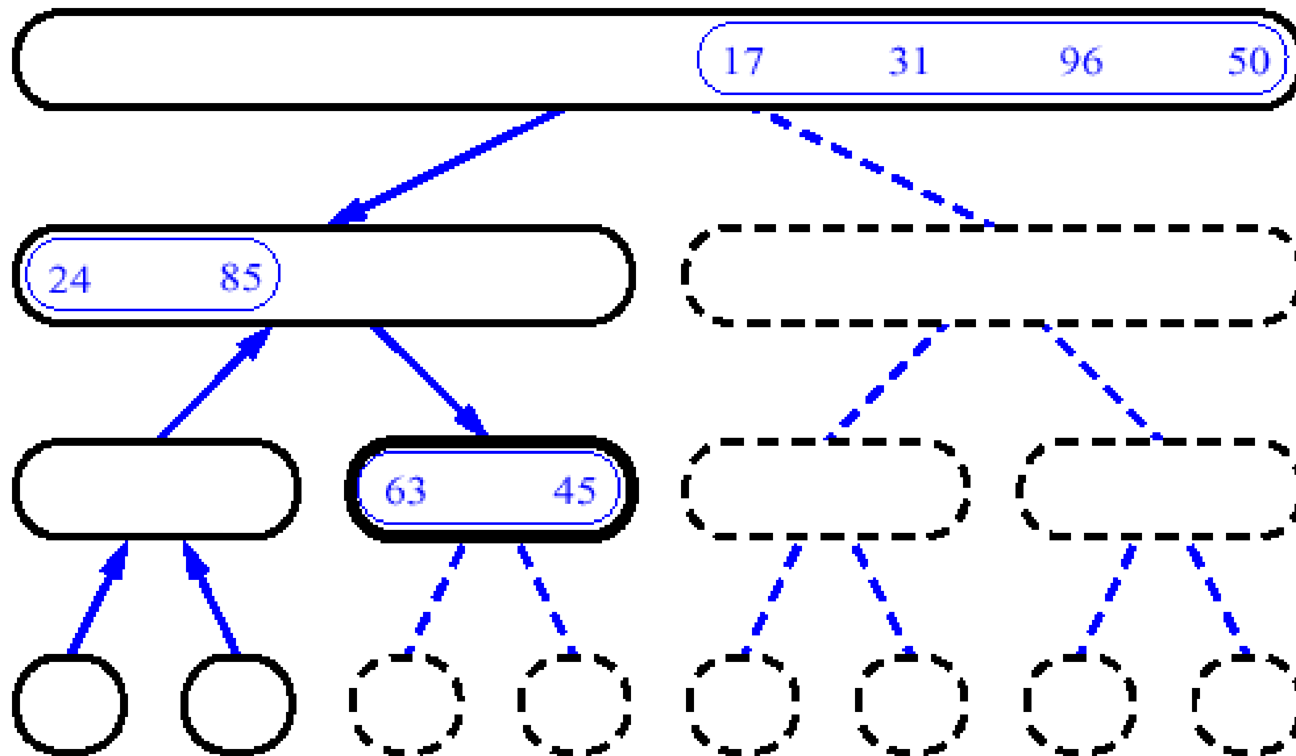
MergeSort (Example) - 8



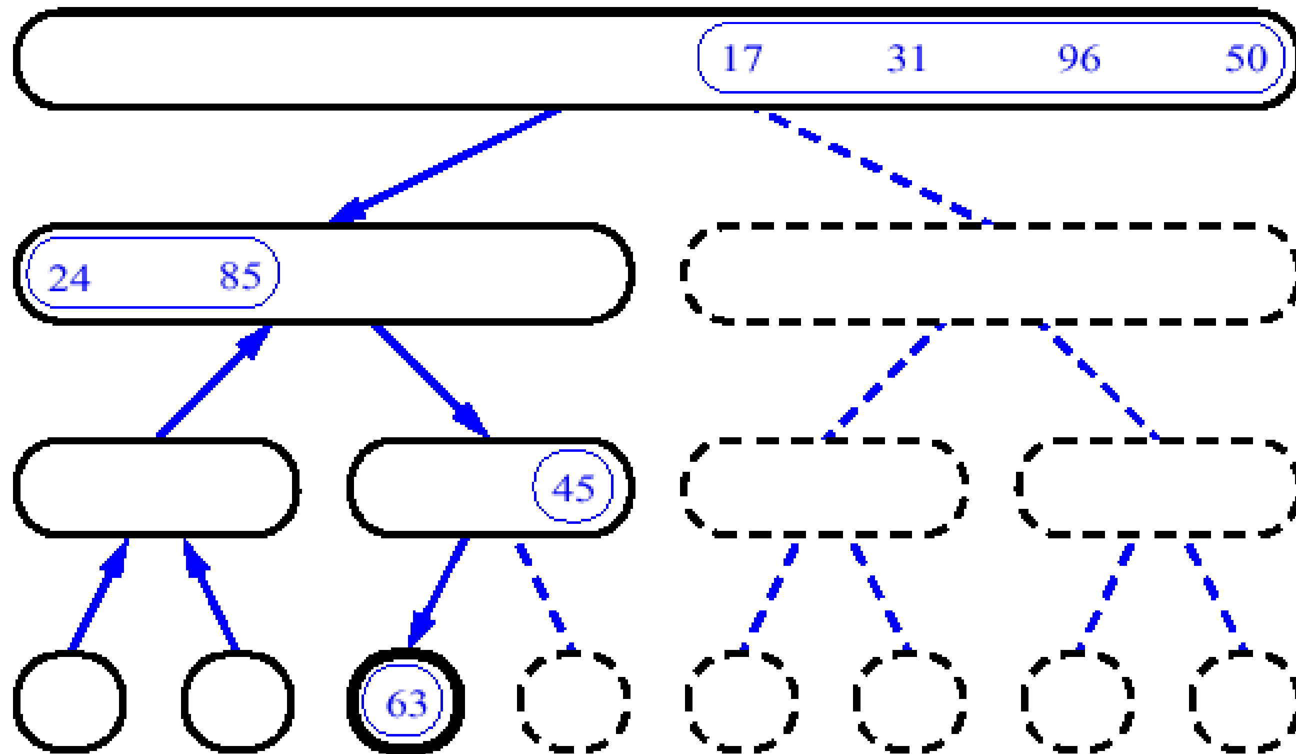
MergeSort (Example) - 9



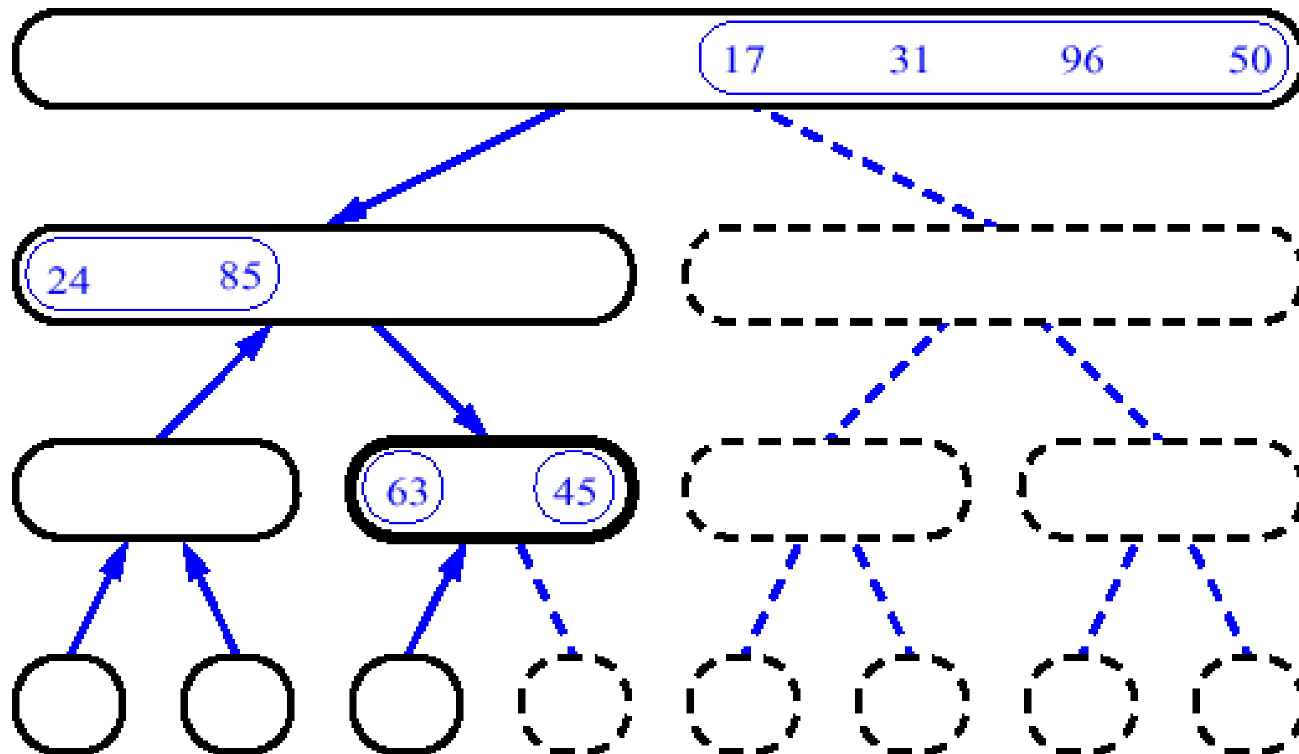
MergeSort (Example) - 10



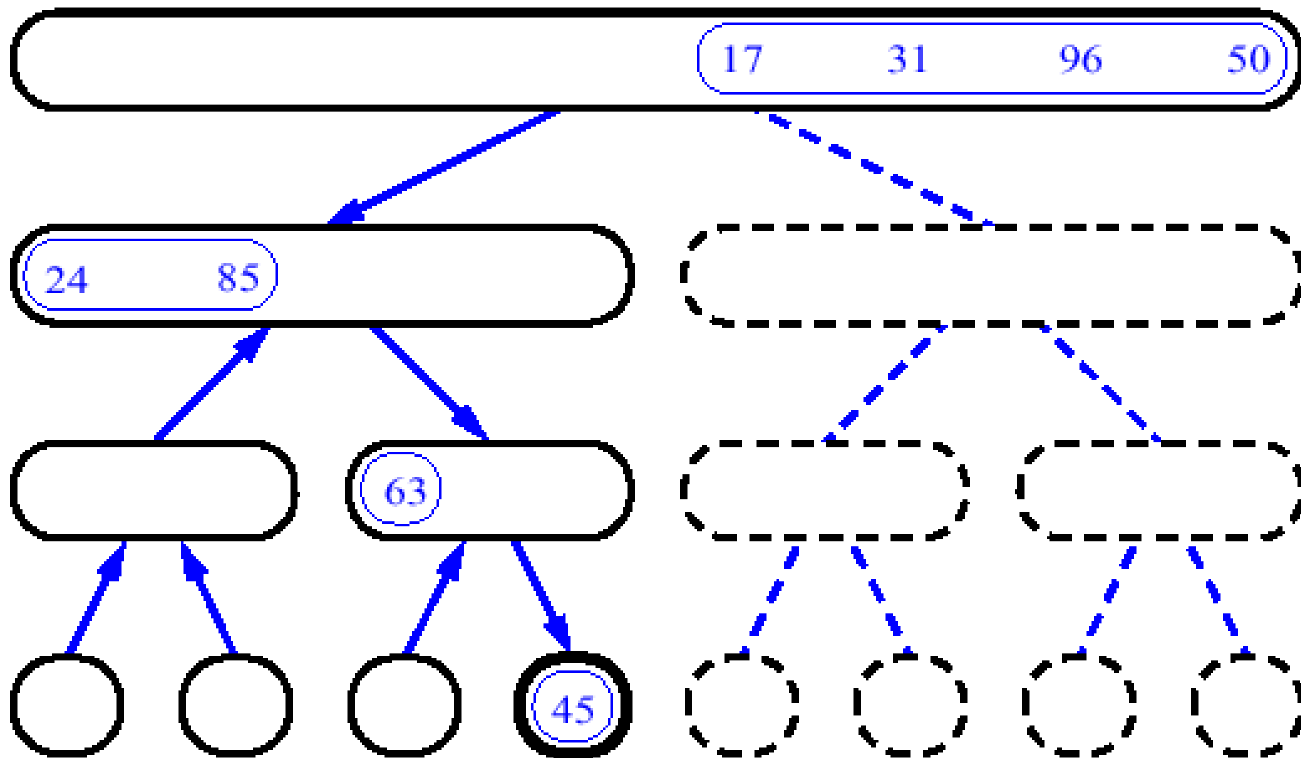
MergeSort (Example) - 11



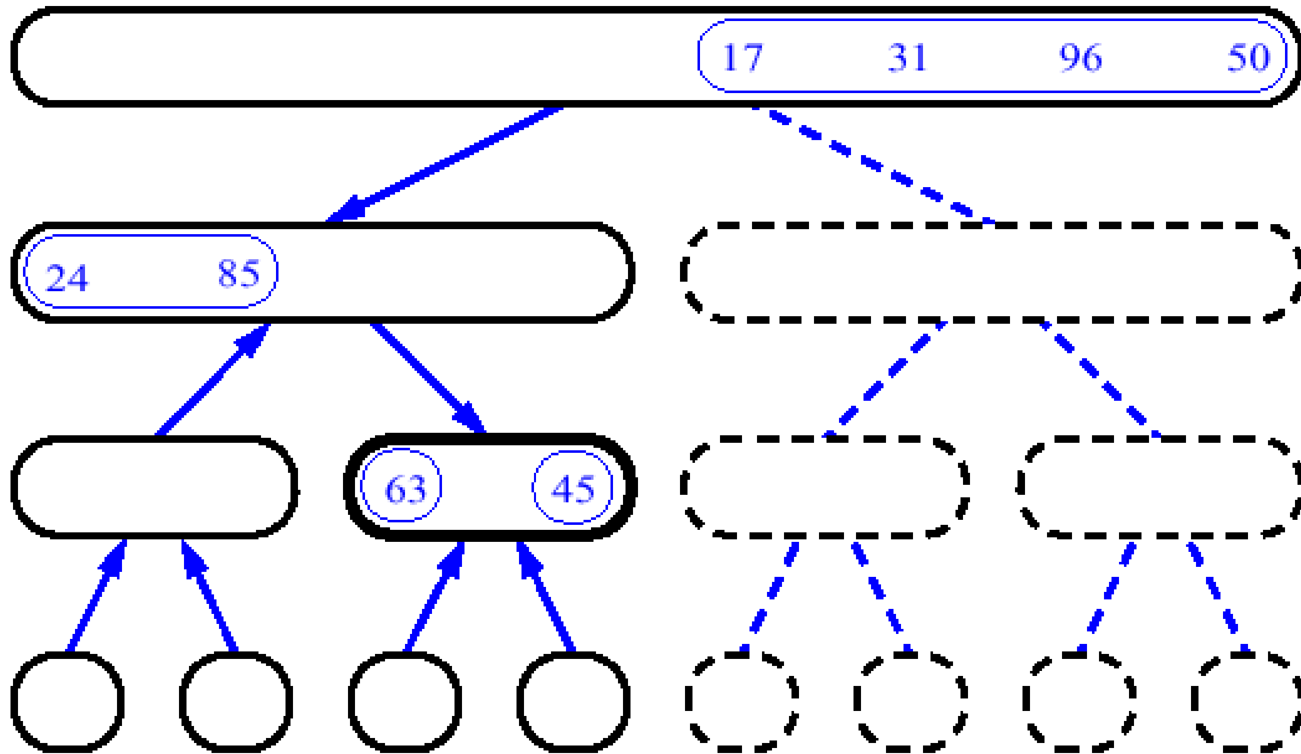
MergeSort (Example) - 12



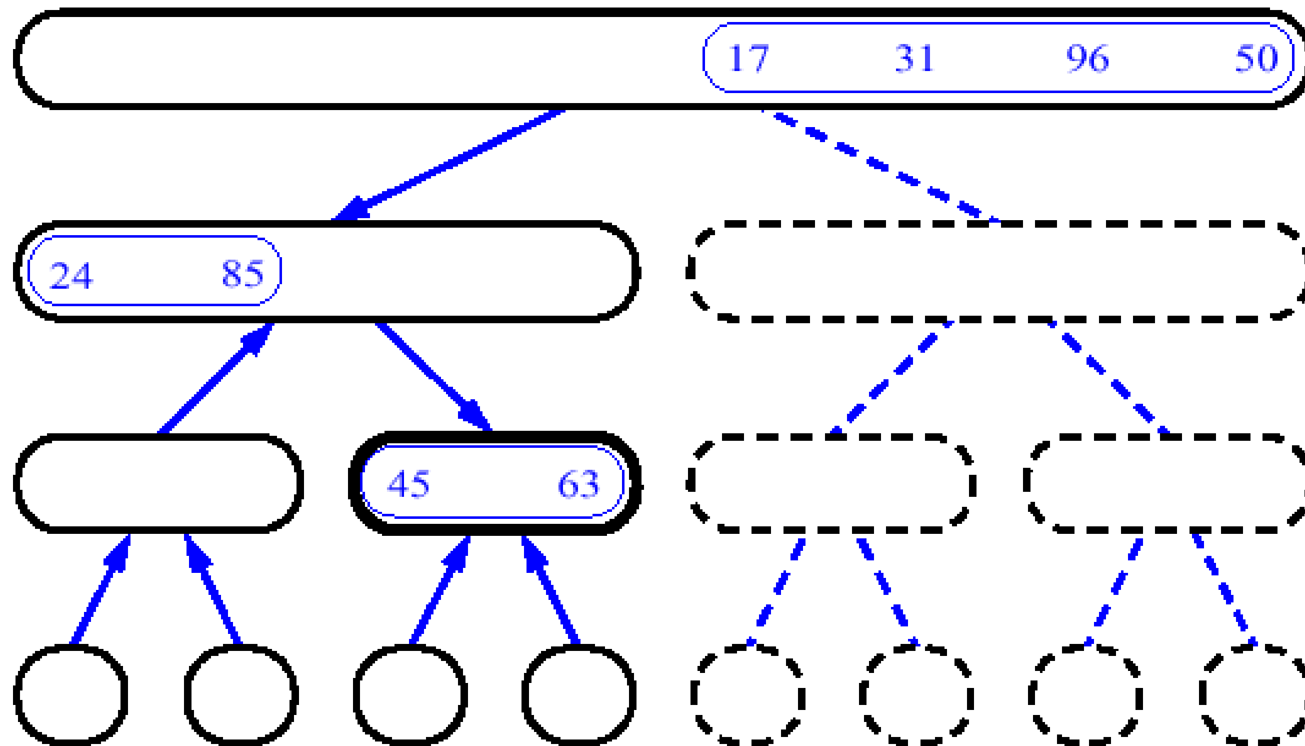
MergeSort (Example) - 13



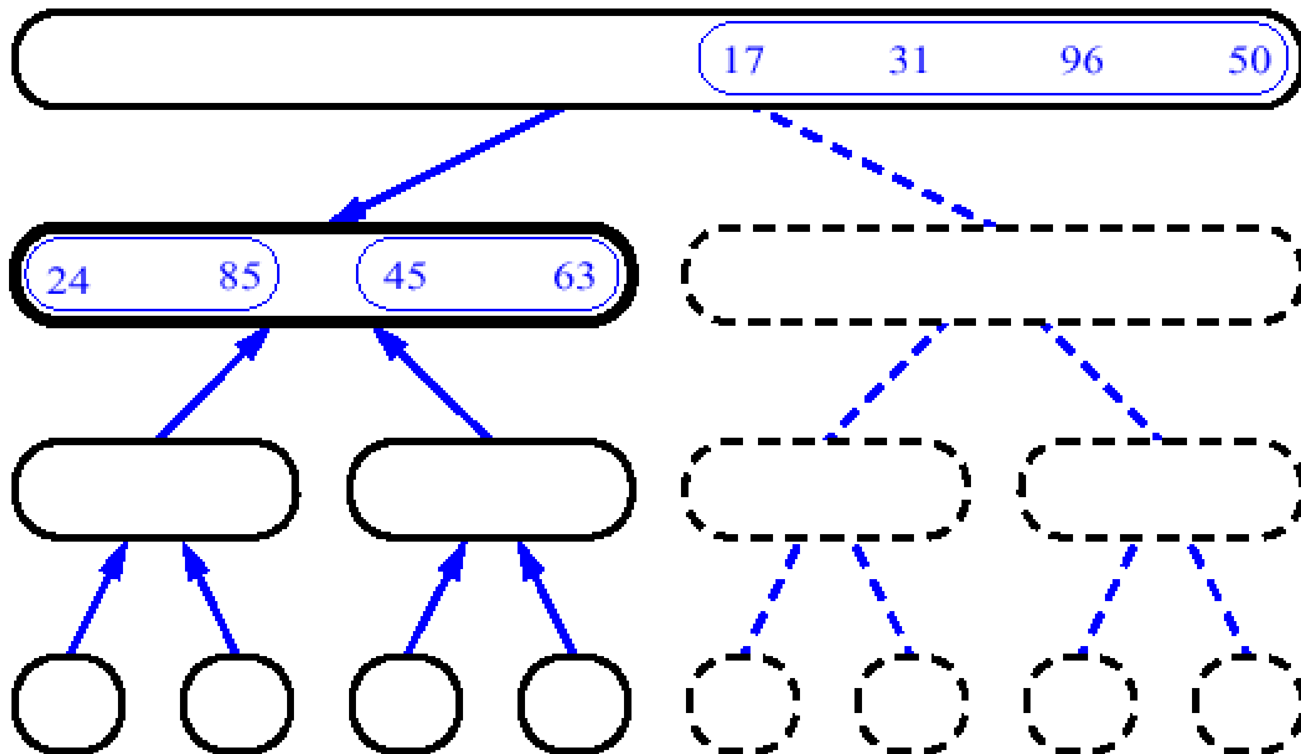
MergeSort (Example) - 14



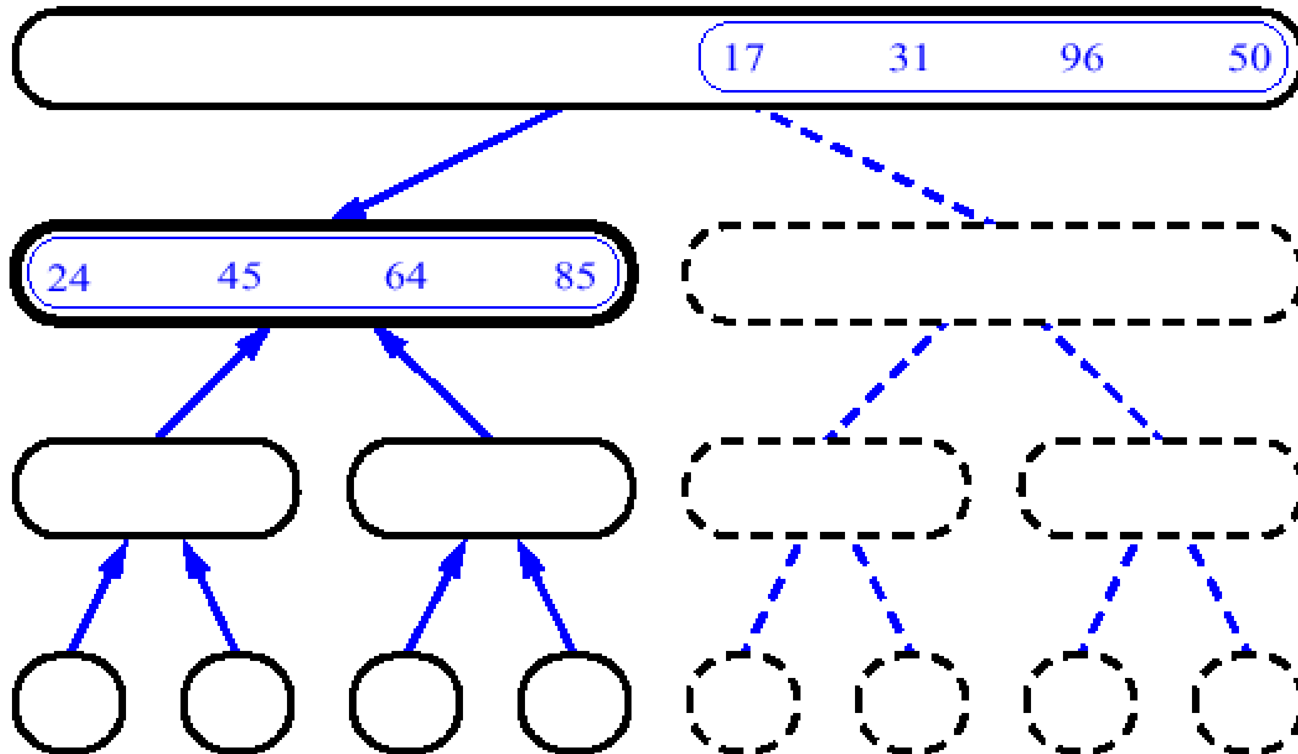
MergeSort (Example) - 15



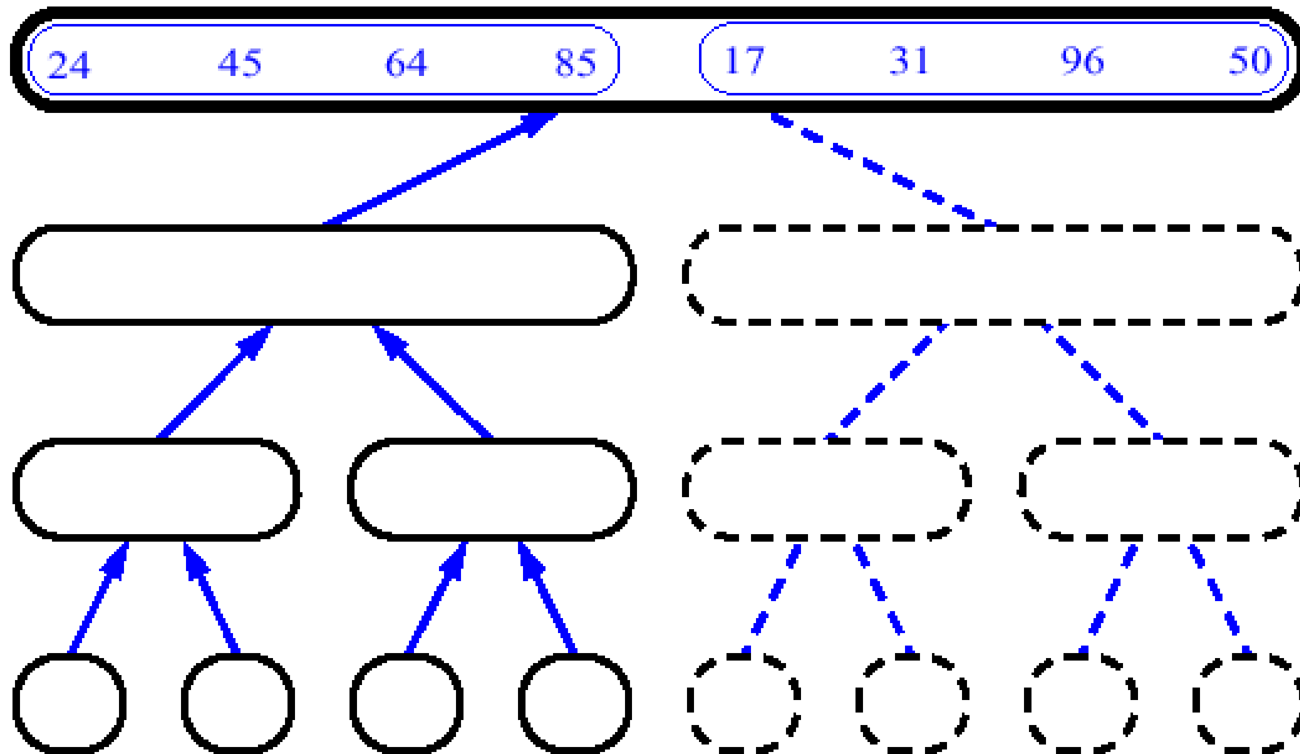
MergeSort (Example) - 16



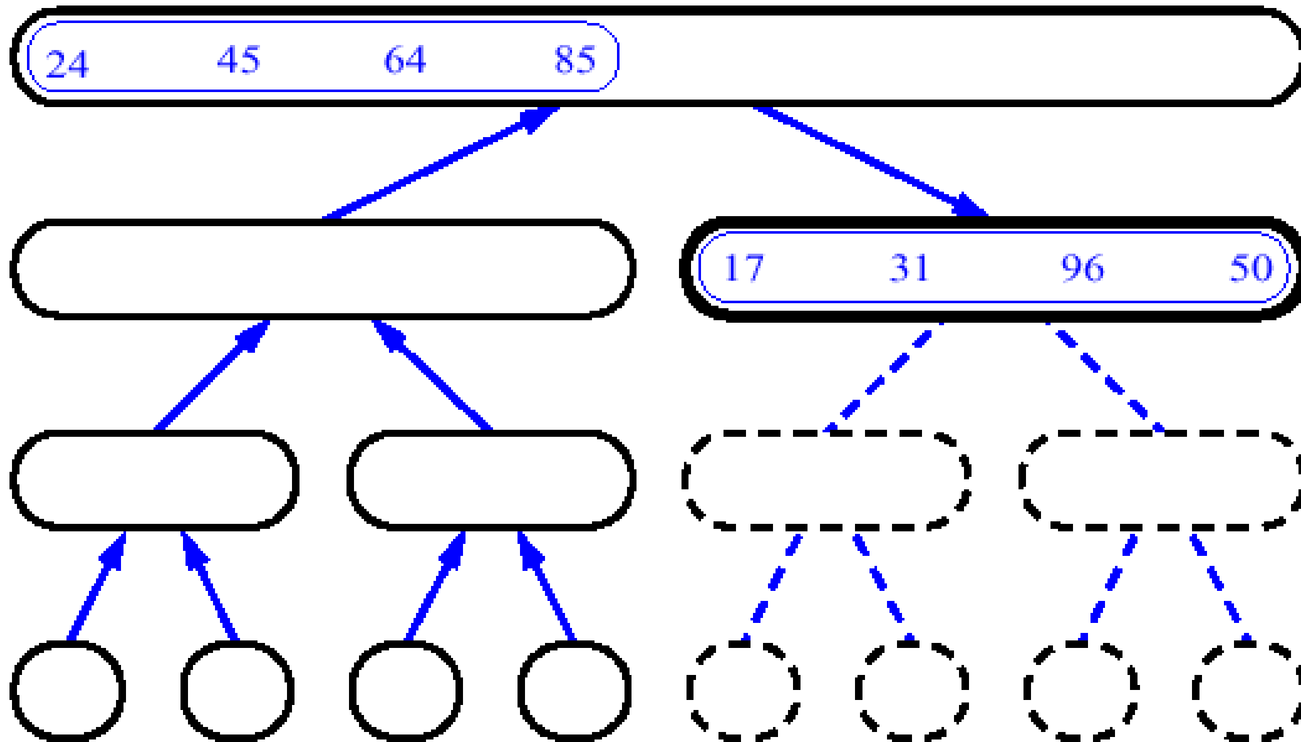
MergeSort (Example) - 17



MergeSort (Example) - 18

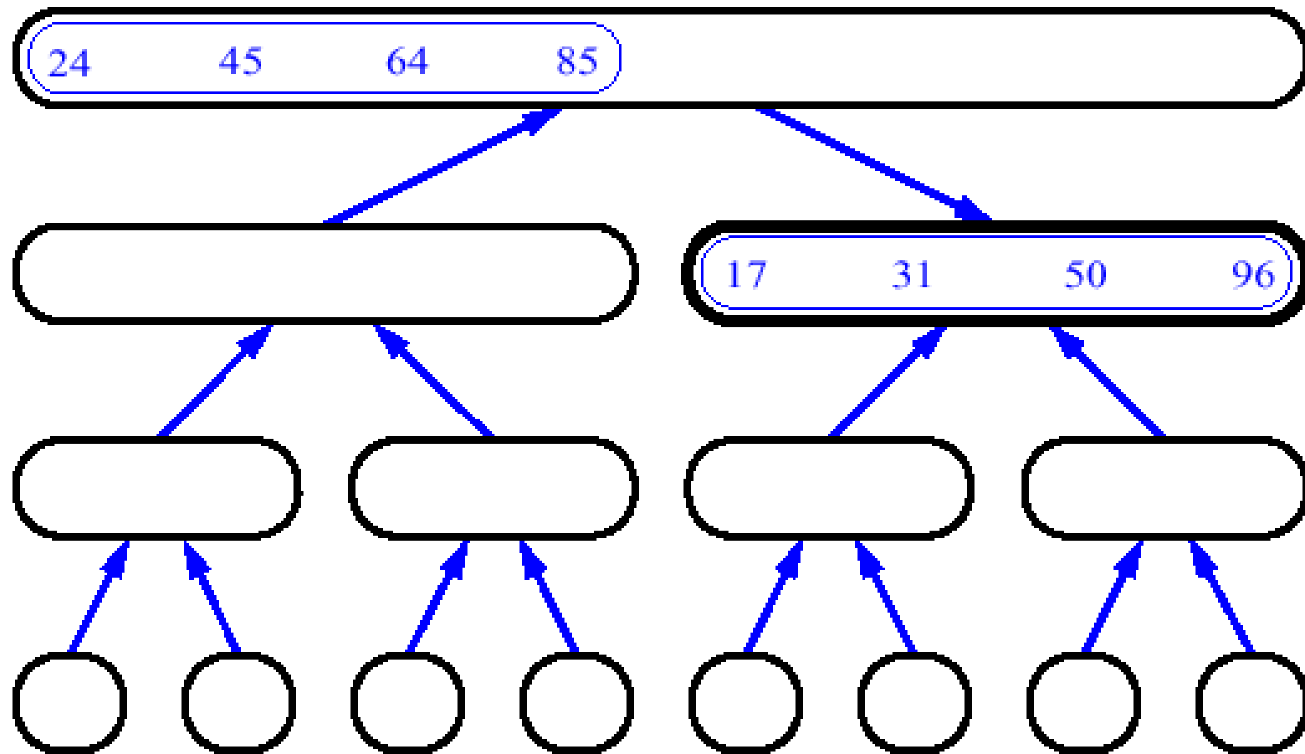


MergeSort (Example) - 19

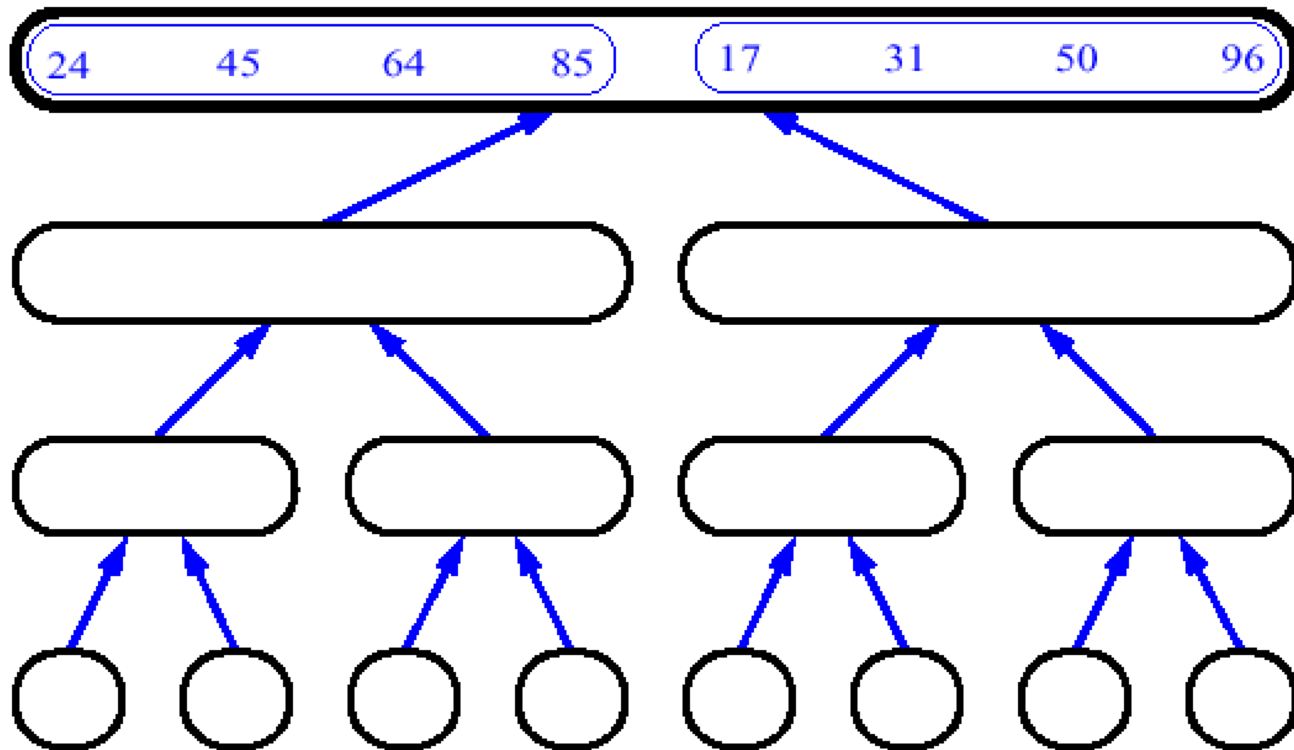


SSN

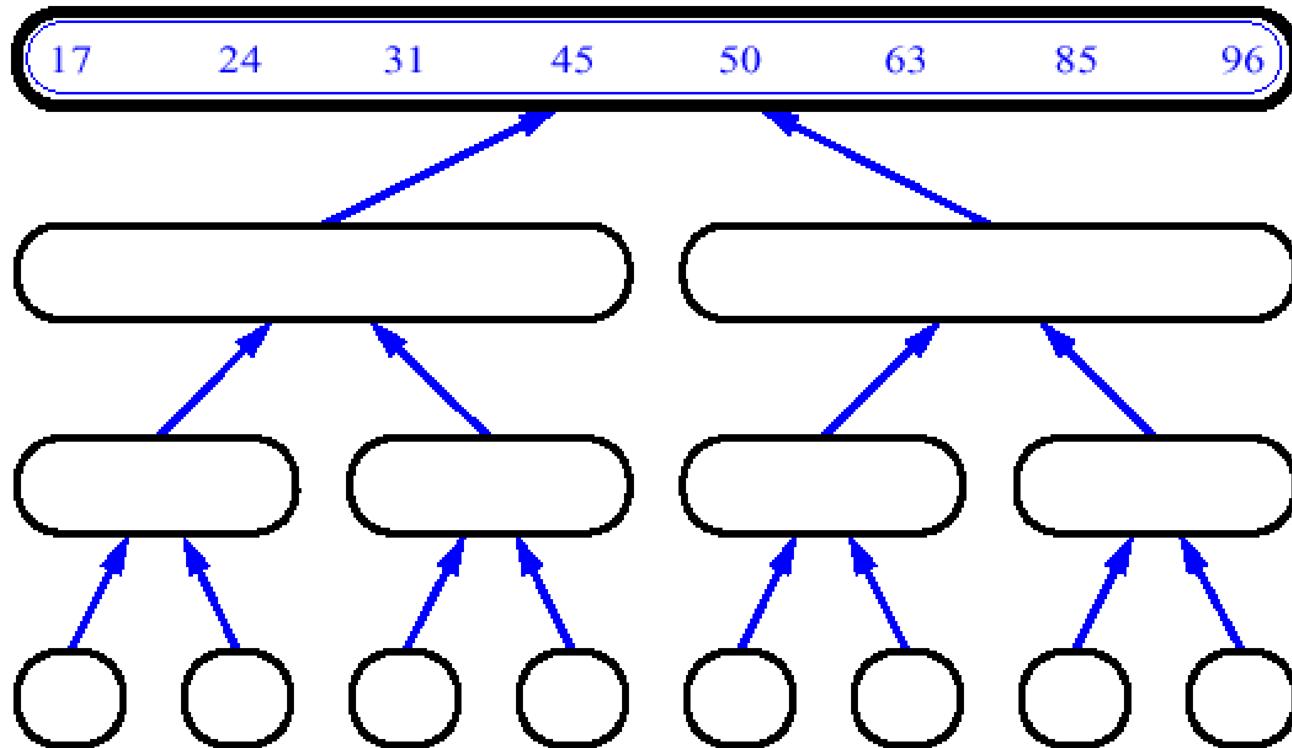
MergeSort (Example) - 20



MergeSort (Example) - 21



MergeSort (Example) - 22



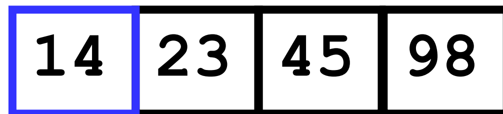
14	23	45	98

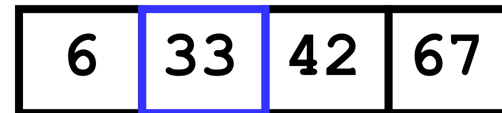
6	33	42	67

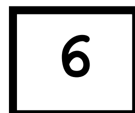
14	23	45	98

6	33	42	67

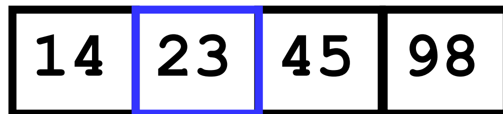
Merge

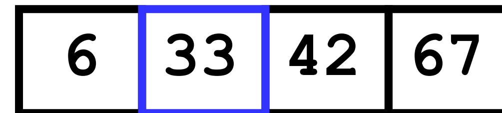

14
23
45
98

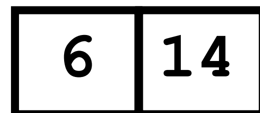

6
33
42
67


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

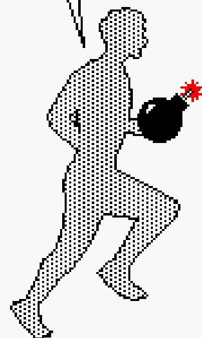
6	14	23	33	42	45	67	98

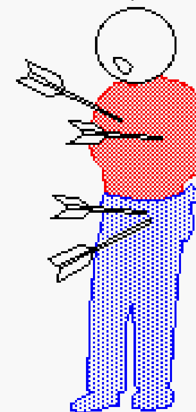
Merge

Quicksort

Thank goodness! It's Quicksort Man! Help me!

I'm on my way, Bubble Sort Man.





Quick-Sort

- To understand quick-sort, let's look at a high-level description of the algorithm

1) **Divide / Partition** : If the sequence S has 2 or more elements, select an element x from S to be your **pivot**. Any arbitrary element, like the last, will do. Remove all the elements of S and divide them into 3 sequences:

    L, holds S's elements less than x

    E, holds S's elements equal to x

    G, holds S's elements greater than x

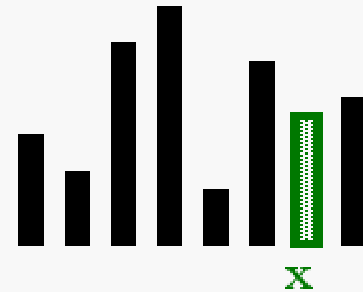
2) **Recurse**: Recursively sort L and G

3) **Conquer**: Finally, to put elements back into S in order, first inserts the elements of L, then those of E, and those of G.

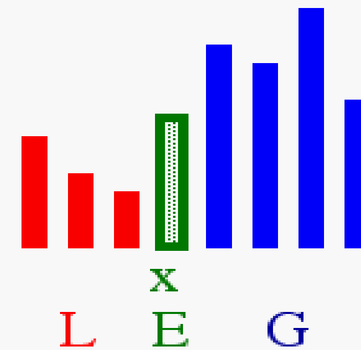
Here are some pretty diagrams....

Idea of Quick Sort

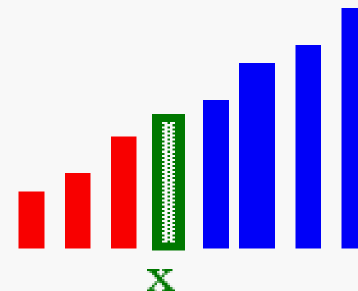
1) **Select**: pick an element



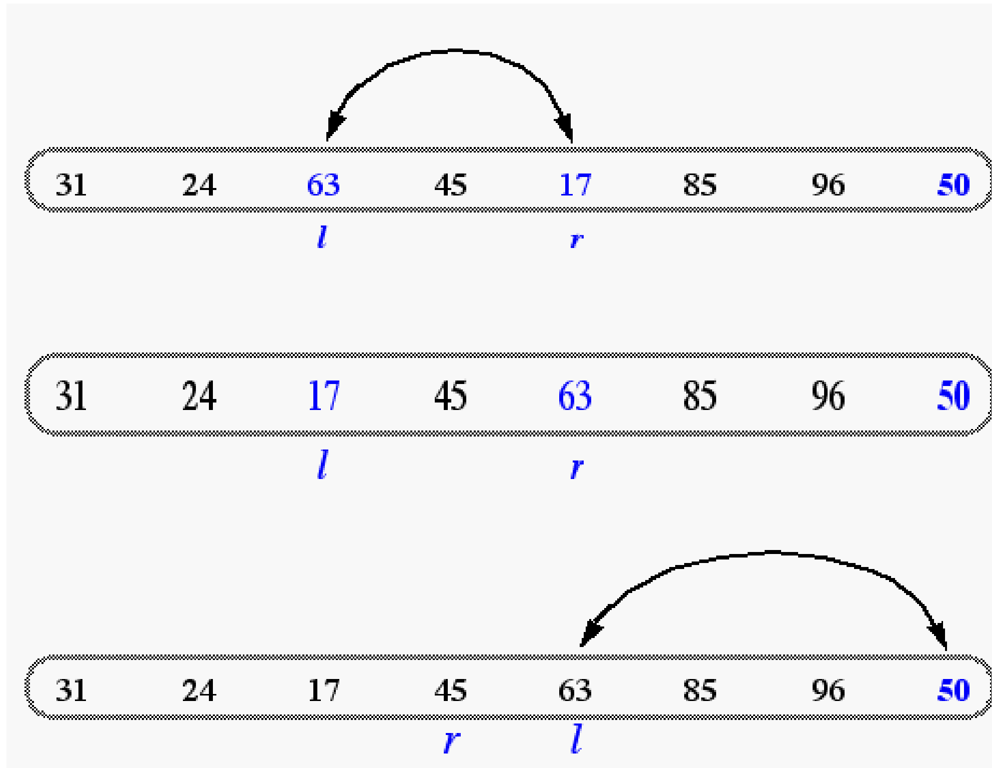
2) **Divide/ Partition**:
Rearrange elements so that x goes to itsnal position E



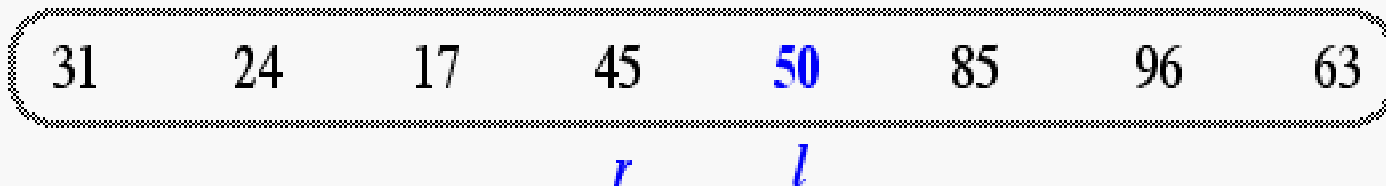
3) **Recurse and Conquer**:
recursively sort



Partition

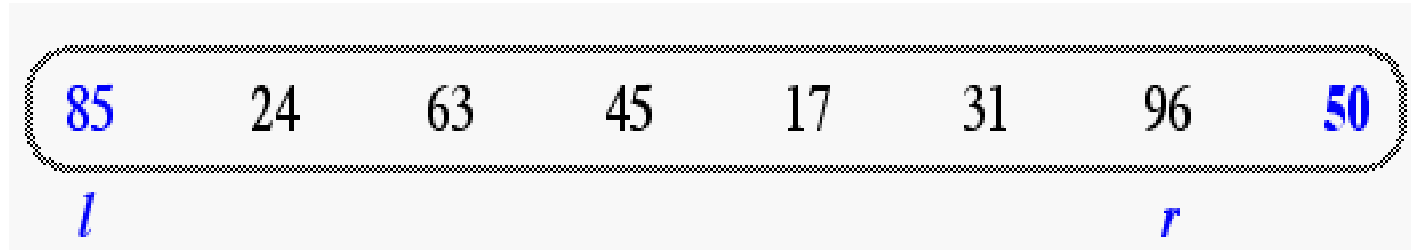


A final swap with the pivot completes the divide step

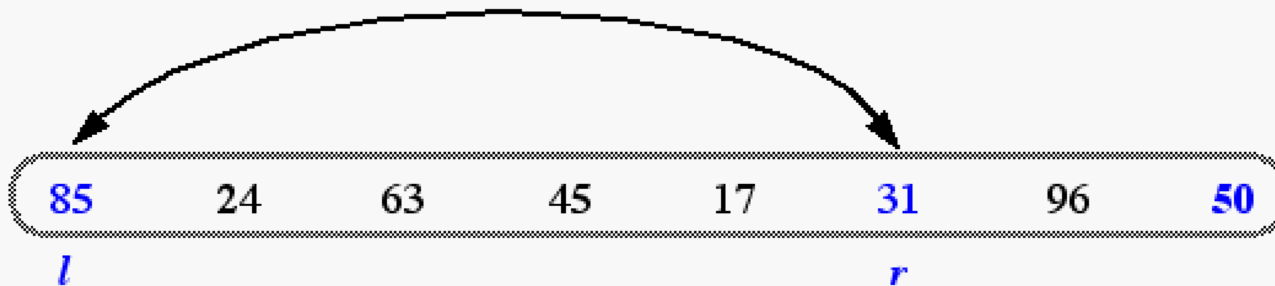


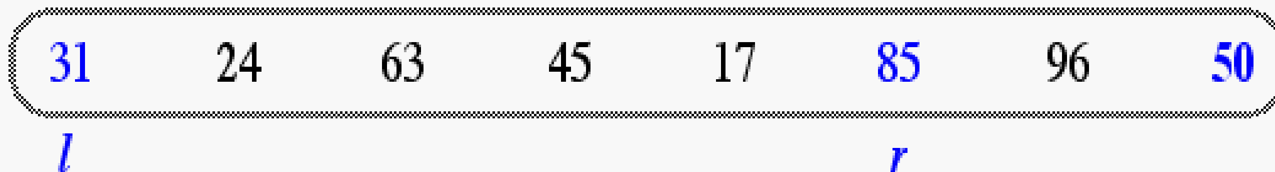
Partition

Divide step: l scans the sequence from the left, and r from the right.

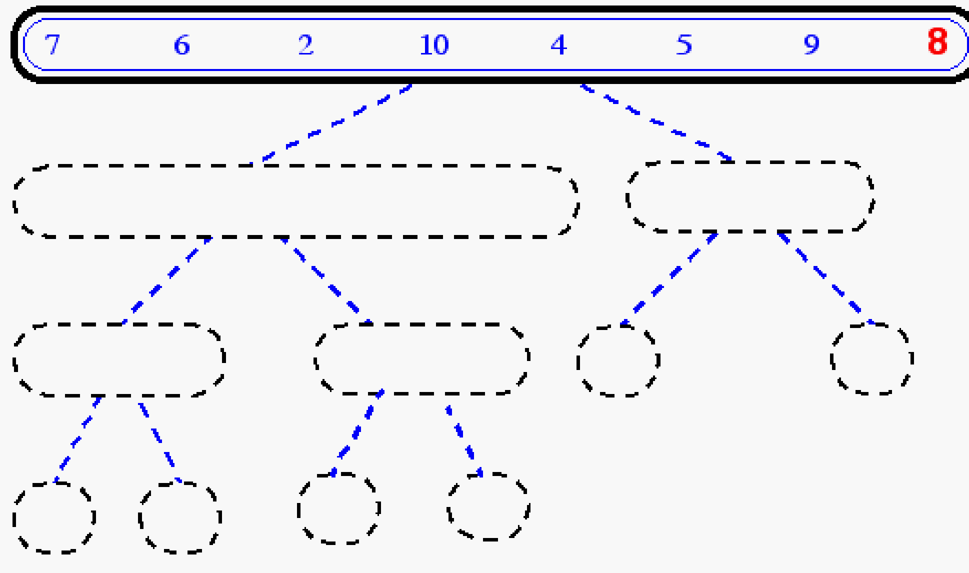


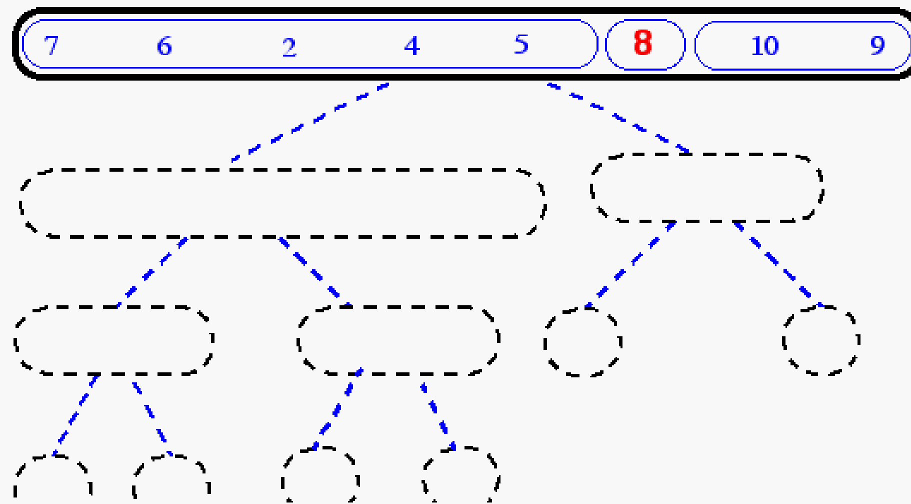
A swap is performed when l is at an element larger than the pivot and r is at one smaller than the pivot.



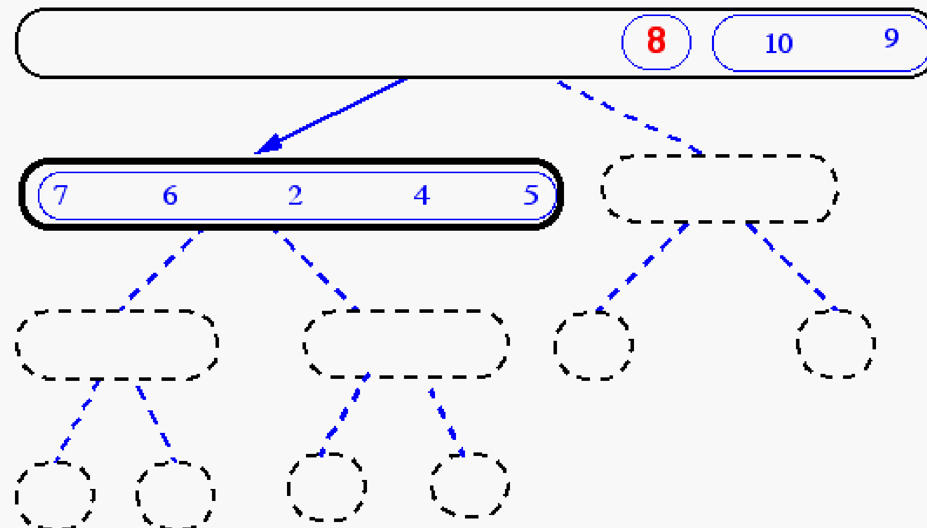


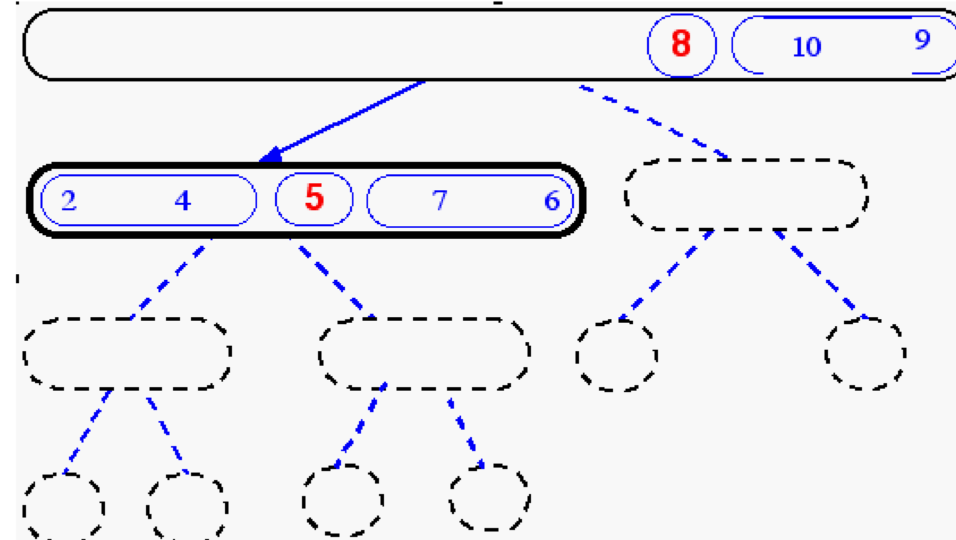
Quick-Sort Tree



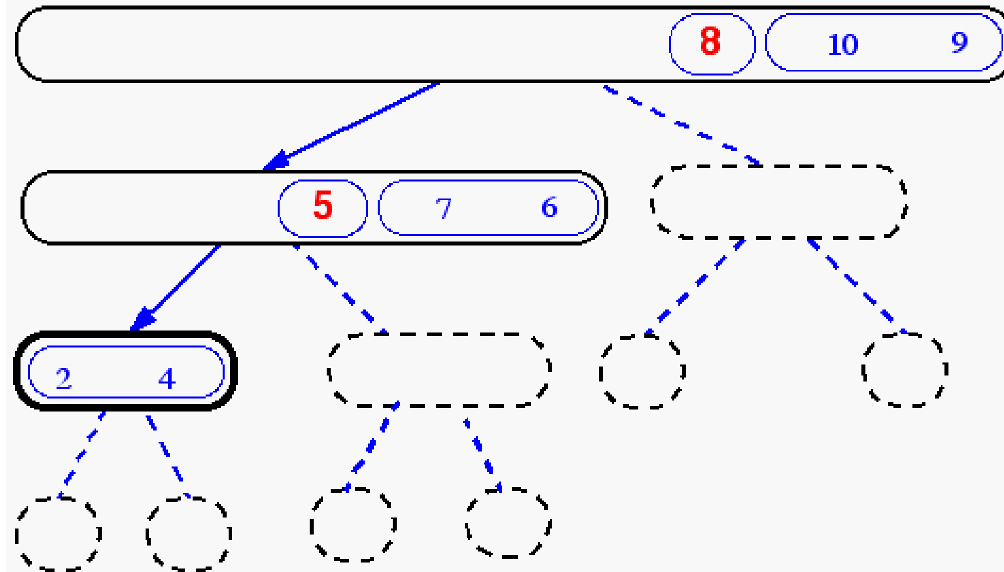


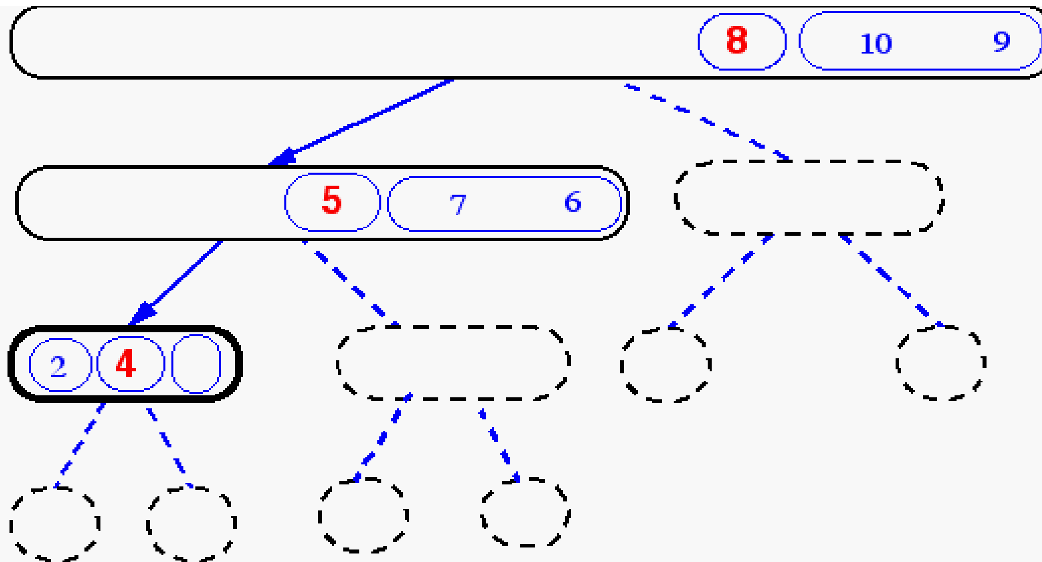
Quick-Sort Tree



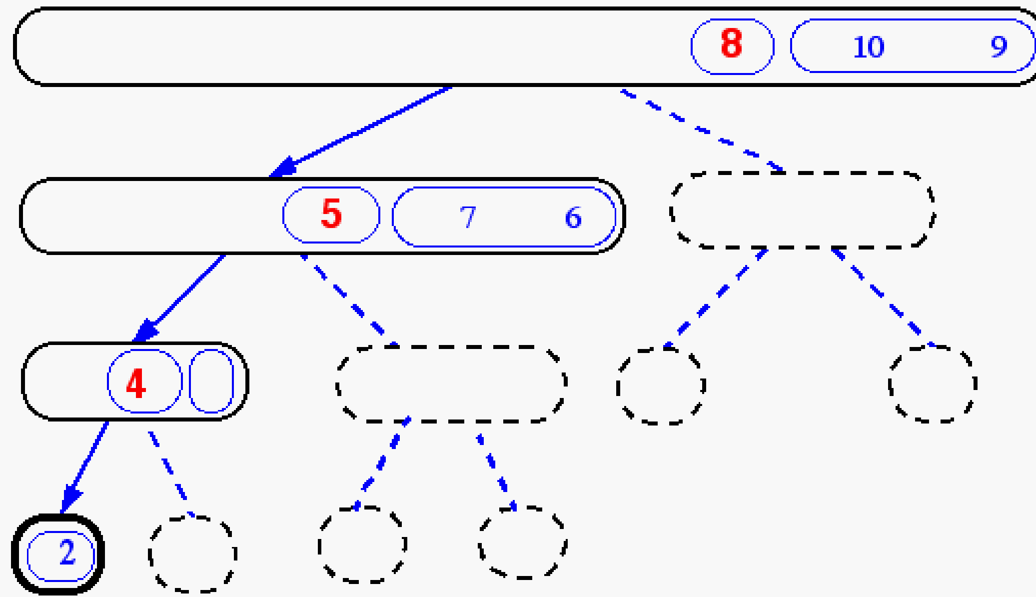


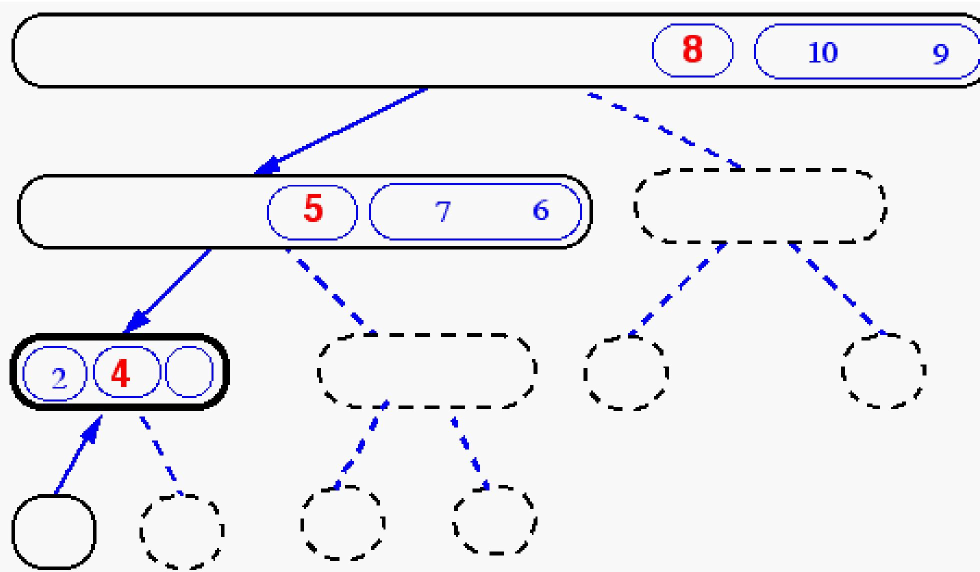
Quick-Sort Tree



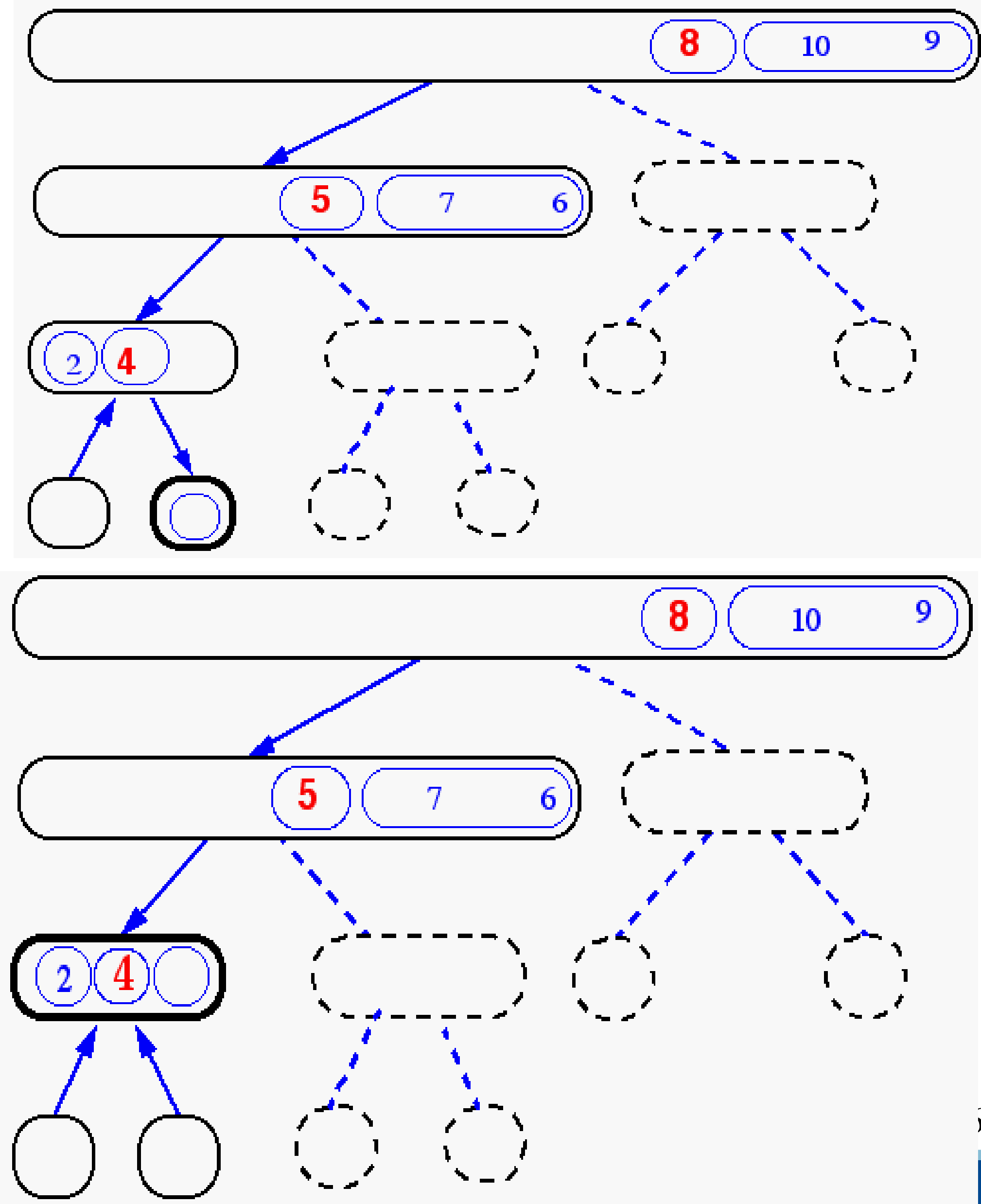


Quick-Sort Tree

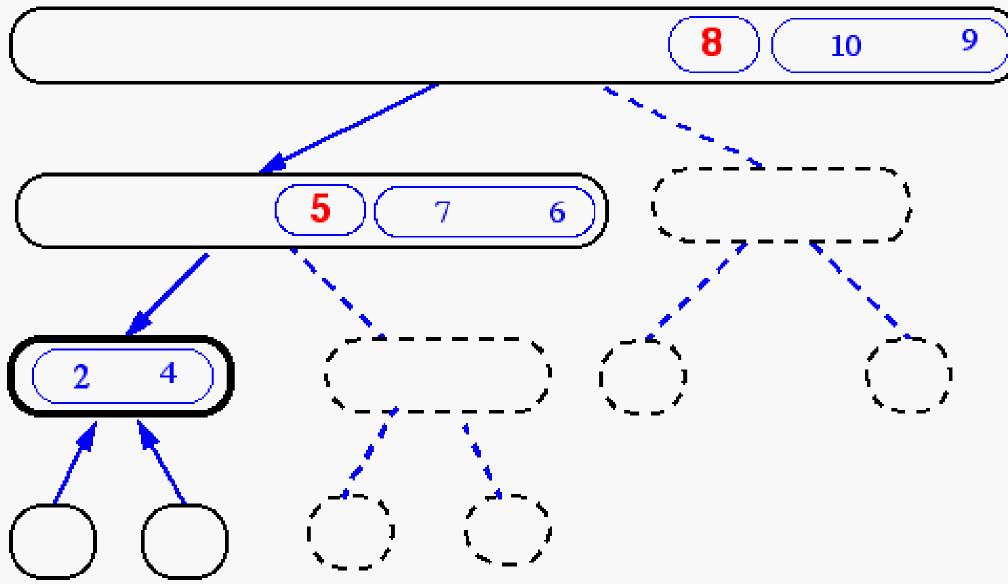


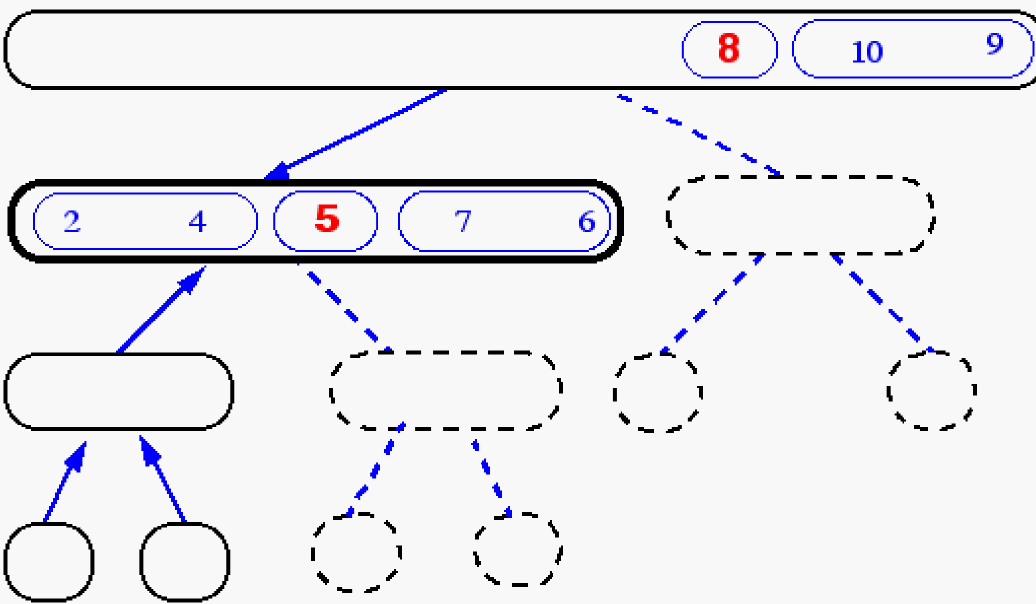


Quick-Sort Tree

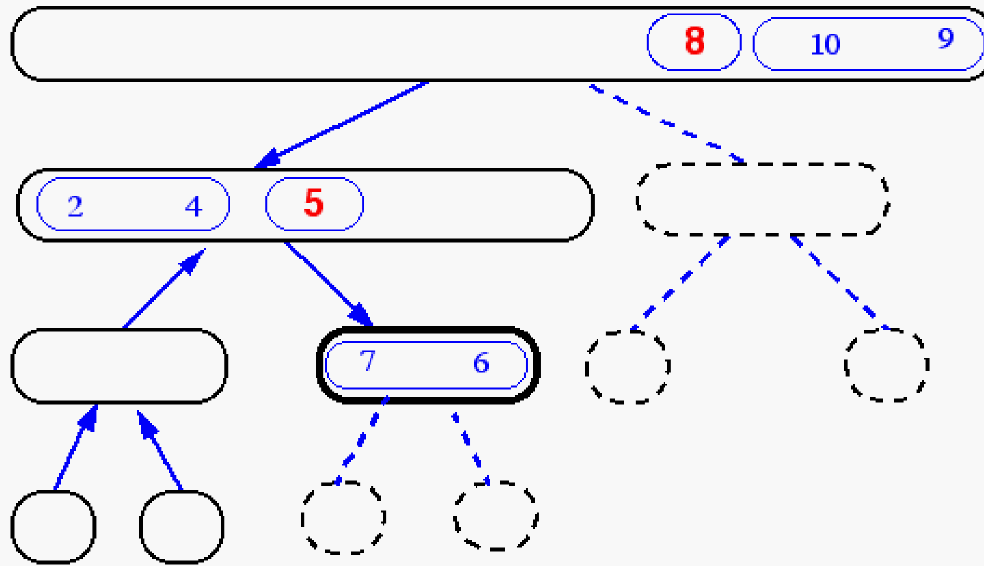


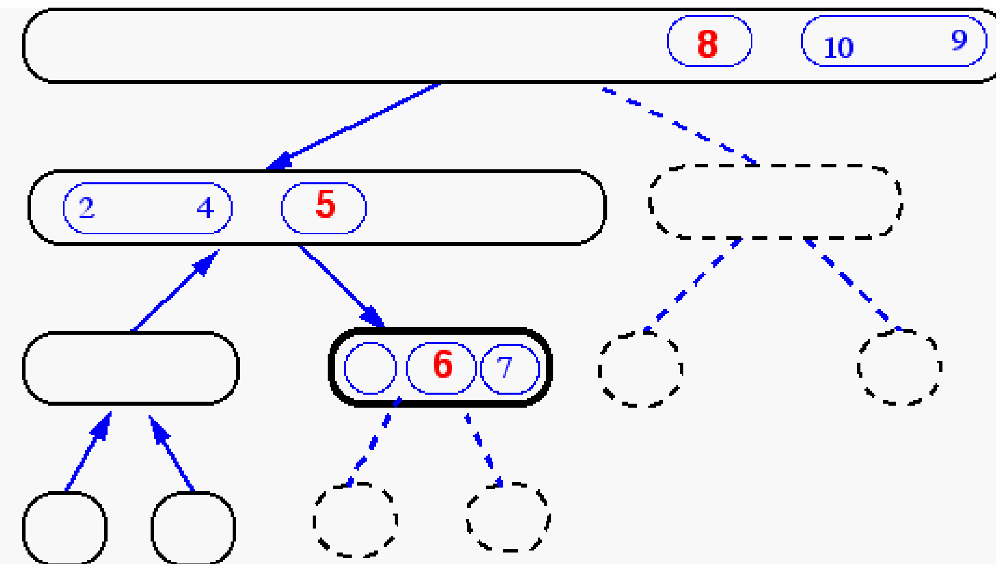
Quick-Sort Tree



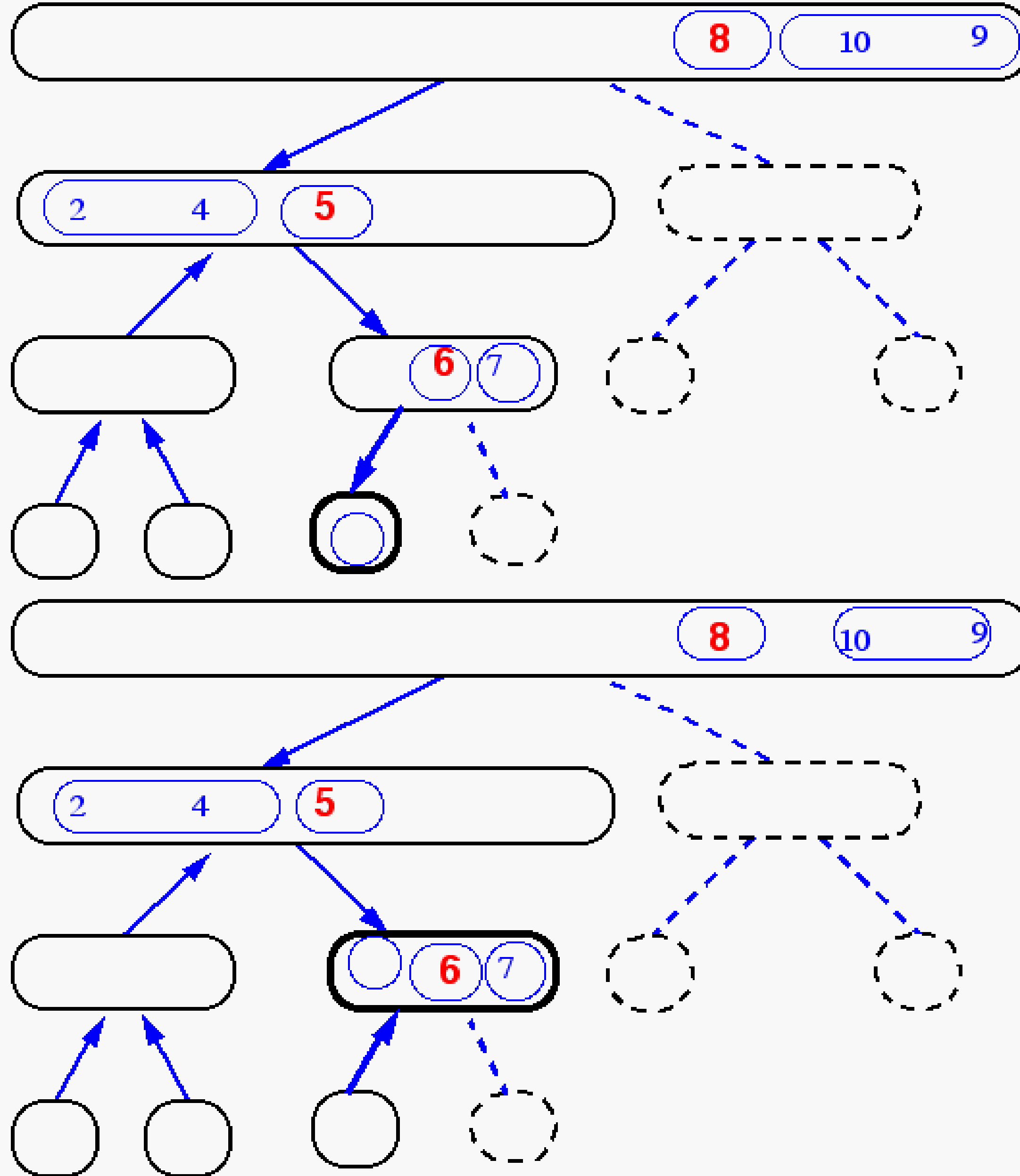


Quick-Sort Tree

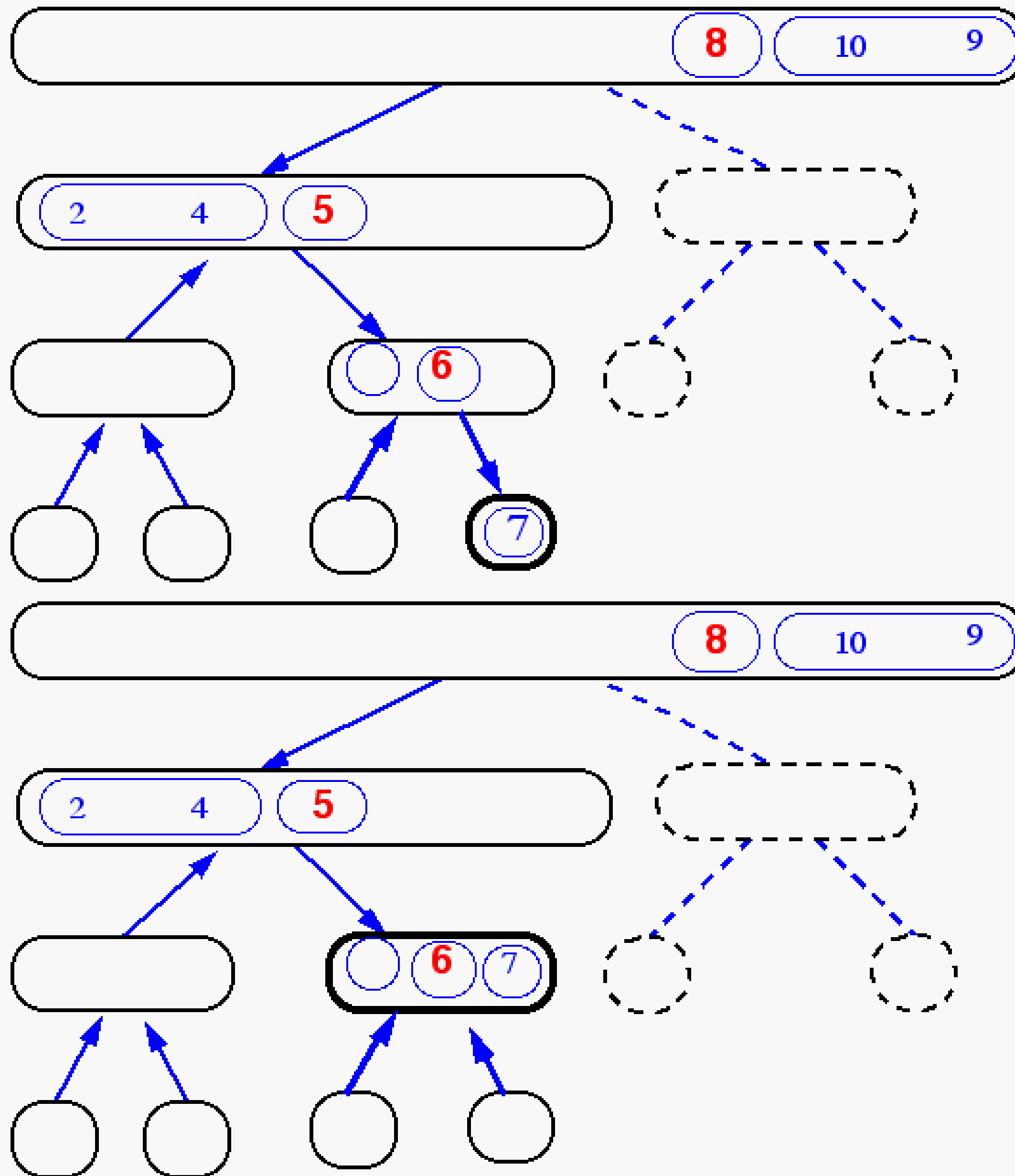




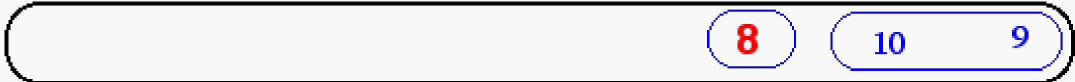
Quick-Sort Tree

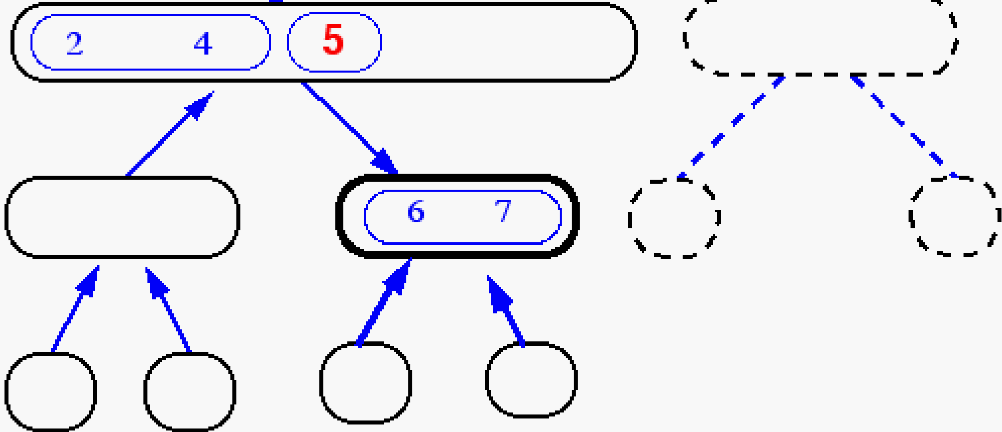


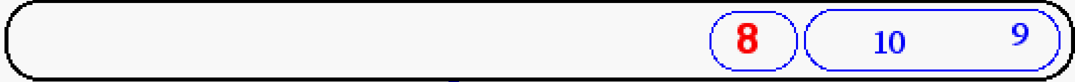
Quick-Sort Tree

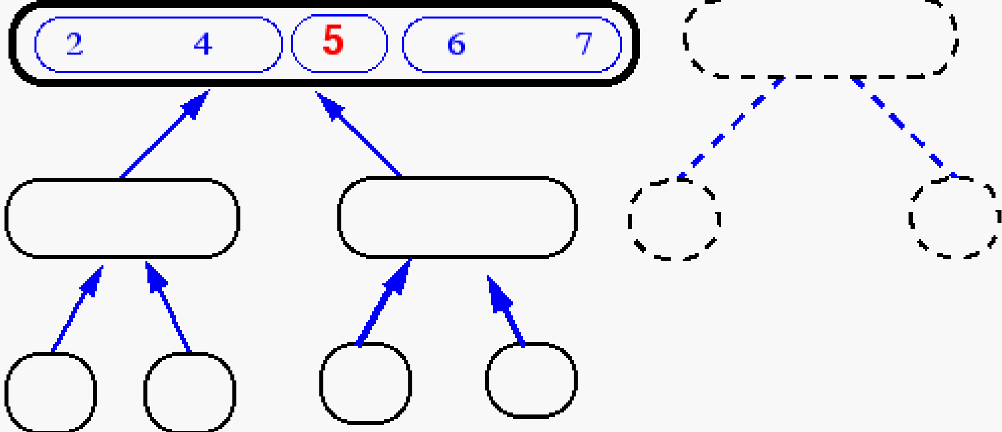


Quick-Sort Tree

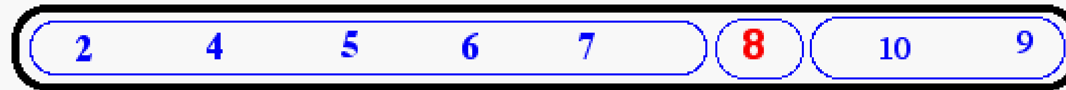


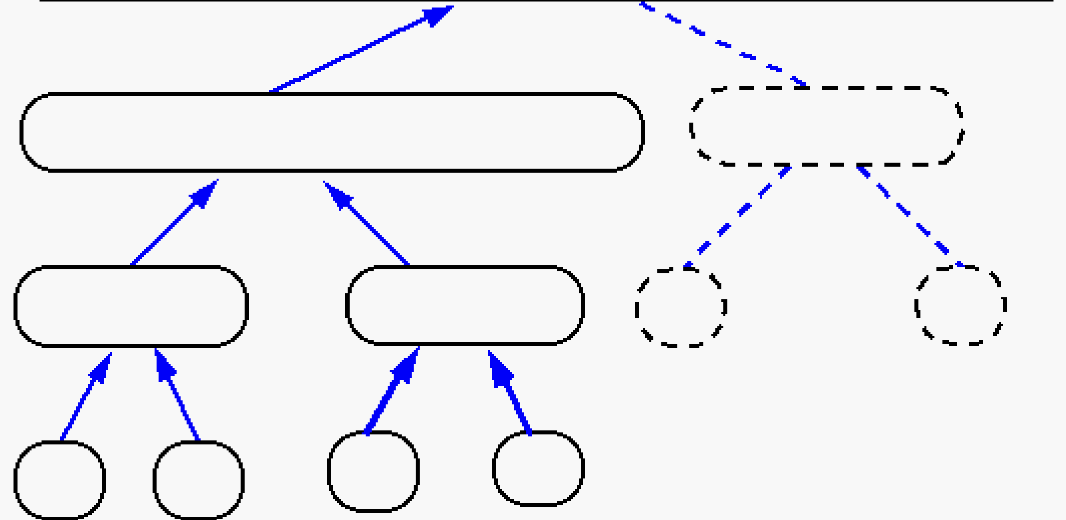


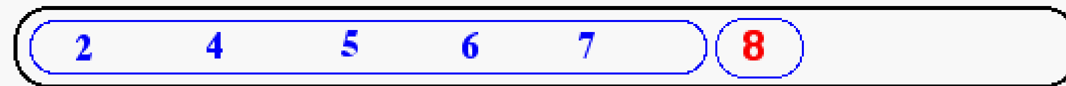


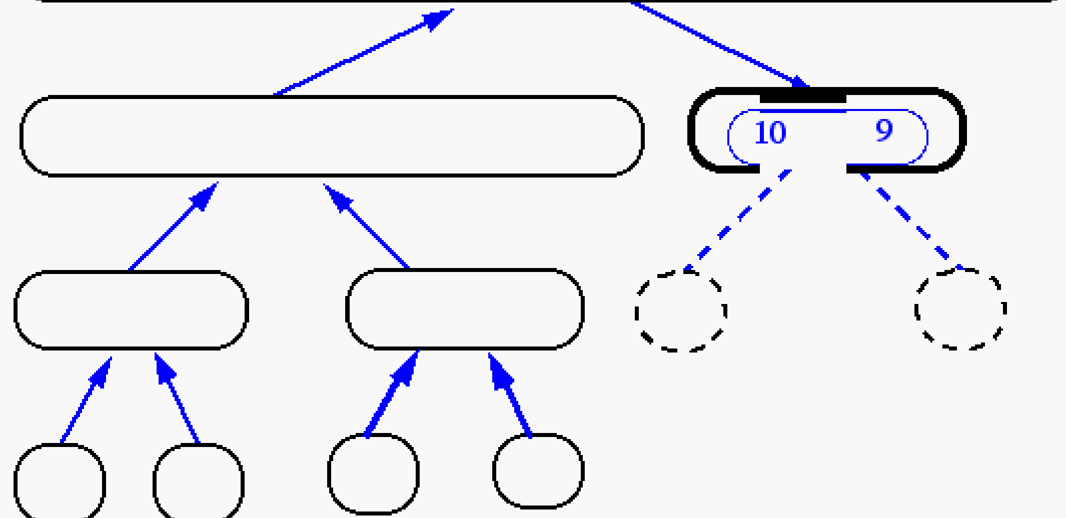


Quick-Sort Tree

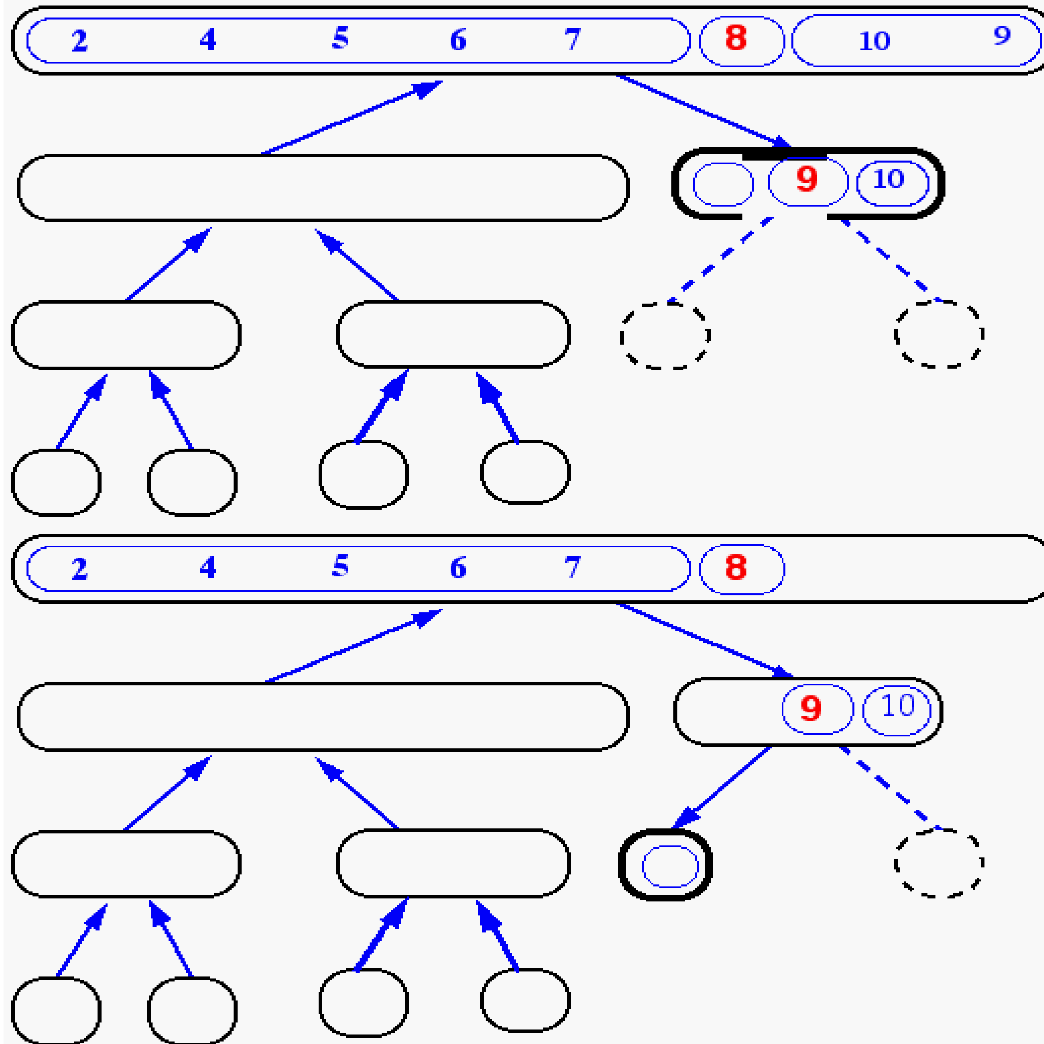






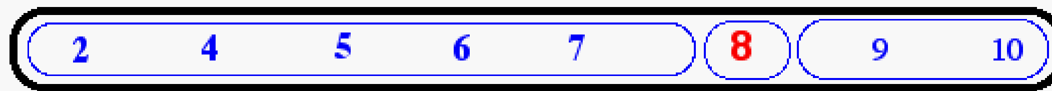


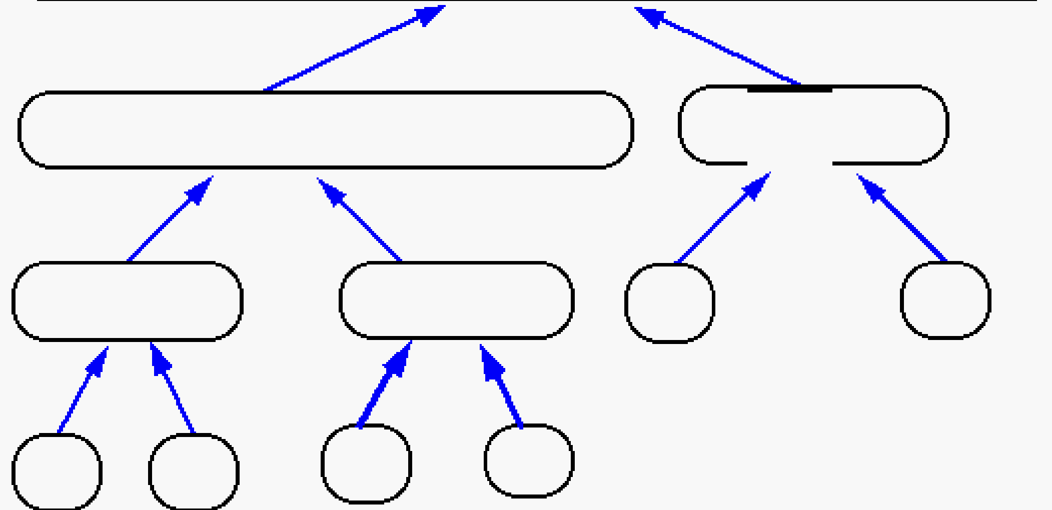
Quick-Sort Tree

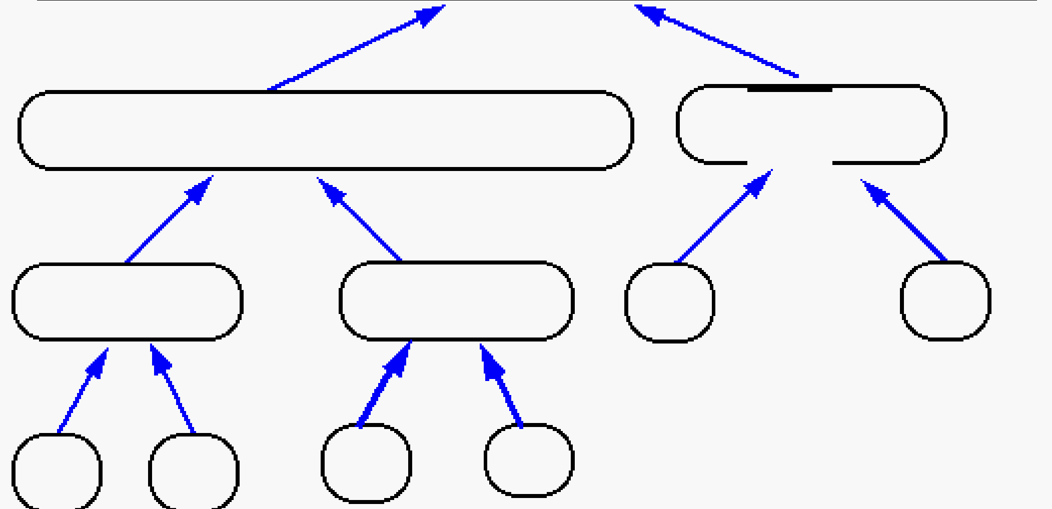


Skipping ...

... Finally







Binary Search

Presentation by:
S.V.Jansi Rani
AP/CSE
SSN College of Engineering

Searching Arrays

- Linear search

  small arrays

  unsorted arrays

- Binary search

  large arrays

  sorted arrays

Linear Search Algorithm

Start at first element of array.

Compare value to value (key) for which you are searching

Continue with next element of the array until you find a match or reach the last element in the array.

Note: On the average you will have to compare the search key with half the elements in the array.

Binary Search Algorithm

May only be used on a sorted array.
Eliminates one half of the elements after each comparison.

Locate the middle of the array

Compare the value at that location with the search key.

If they are equal - done!
Otherwise, decide 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.

SSN

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

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

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

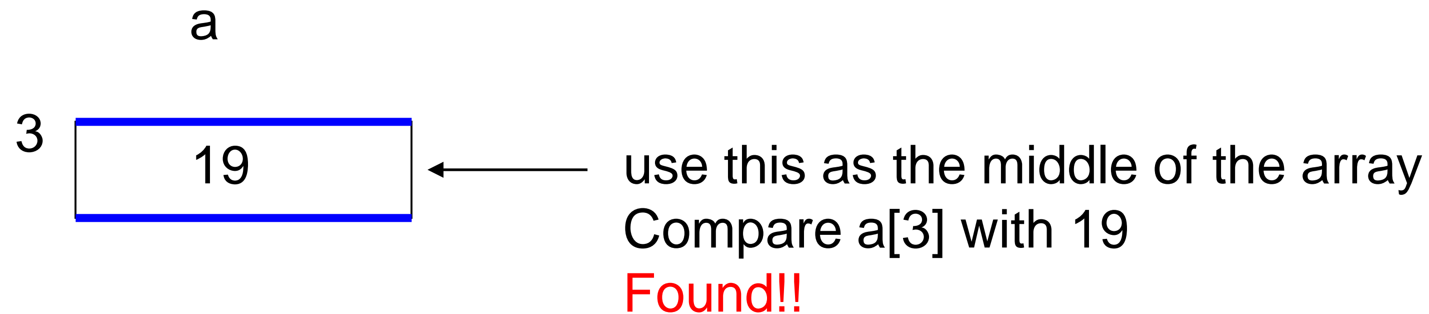
a

3	19
4	25
5	27

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



Binary Search Example

a

	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

a

3	19
4	25
5	27

use this as the middle of the array
Compare a[4] with 18

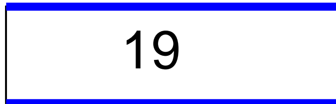
25 is bigger than 18 so use the bottom half

Binary Search Pass 4

search key = 18

a

3



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

```
Algorithm: Binary_Search( A, low, high, x)
Input: sorted array A[ ], low, high, key
Output: Index  or 0 if element not found
if( high == low)
    { if (x == a[low])  return low ;
      else              return 0;
    }
else
{  mid = (low+high)/2
   if (x==a[mid])
      return mid
   else if (x<a[mid])
      return Binary_Search( A,low,mid-1,x)
   else
      return Binary_Search( A,mid+1,high,x)
}
```

Example

-15	-6	0	7	9	23	54	82	101	112	125	131	142	151
1	2	3	4	5	6	7	8	9	10	11	12	13	14

X	low	high	mid
142			

Example

-15	-6	0	7	9	23	54	82	101	112	125	131	142	151
1	2	3	4	5	6	7	8	9	10	11	12	13	14

X	**low**	**high**	**mid**
142	1	14	7
	8	14	11
	12	14	13 found

Example

-15	-6	0	7	9	23	54	82	101	112	125	131	142	151
1	2	3	4	5	6	7	8	9	10	11	12	13	14

X	low	high	mid
151			

Example

-15	-6	0	7	9	23	54	82	101	112	125	131	142	151
1	2	3	4	5	6	7	8	9	10	11	12	13	14

X	low	high	mid
151	1	14	7
	8	14	11
	12	14	13
	14	14	14 found

Example

-15	-6	0	7	9	23	54	82	101	112	125	131	142	151
1	2	3	4	5	6	7	8	9	10	11	12	13	14

X	low	high	mid
-14			

Example

-15	-6	0	7	9	23	54	82	101	112	125	131	142	151
1	2	3	4	5	6	7	8	9	10	11	12	13	14

X	low	high	mid
-14	1	14	7
	1	6	3
	1	2	1
	1	2	1
	2	2	2 [Return 0]
	2	1	1
	2	1	1
	2	1	1

Algorithm does not halt

What happens if (l==r) terminating condition is not there

Example

-15	-6	0	7	9	23	54	82	101	112	125	131	142	151
1	2	3	4	5	6	7	8	9	10	11	12	13	14

X	low	high	mid
149	1	14	7
	8	14	11
	12	14	13
	14	14	14 return 0
	14	13	13
	14	13	13
	14	13	13
	..	..	..

Algorithm does not halt

What happens if (l==r) terminating condition is not there

Theorem

If n is in the range $[2^{k-1}, 2^k]$, then binary search makes atmost k element comparisons for a successful search and either k-1 or k comparisons for an unsuccessful search.

Time Complexity of Binary Search

Worst Case:

1. Arrays without search key
2. Search element in the first / last position

WORK OUT ??????

Time Complexity of Binary Search

Worst Case:

1. Arrays without search key
2. Search element in the first / last position

WORK OUT ??????

$T(n) = \log_2 n + 1$

Average Case:

$T(n) = \log_2 n$

Best Case:

$T(n) = 1$

Efficiency

Searching an array of 1024 elements will take at most 10 passes to find a match or determine that the element does not exist in the array.
512, 256, 128, 64, 32, 16, 8, 4, 2, 1

An array of one billion elements takes a maximum of 30 comparisons.

The bigger the array the better a binary search is as compared to a linear search

THANK YOU

THANK YOU