

# Session Objectives

- To understand the concepts of
  - Different Sorting Techniques
  - Different Searching Techniques

# Sorting Algorithms

# Sorting Algorithms

- ◆ A sorting algorithm is an algorithm that puts elements of a list in a certain order
- ◆ The most used orders are numerical order
- ◆ Efficient sorting is important to optimize the use of other algorithms that require sorted lists to work correctly

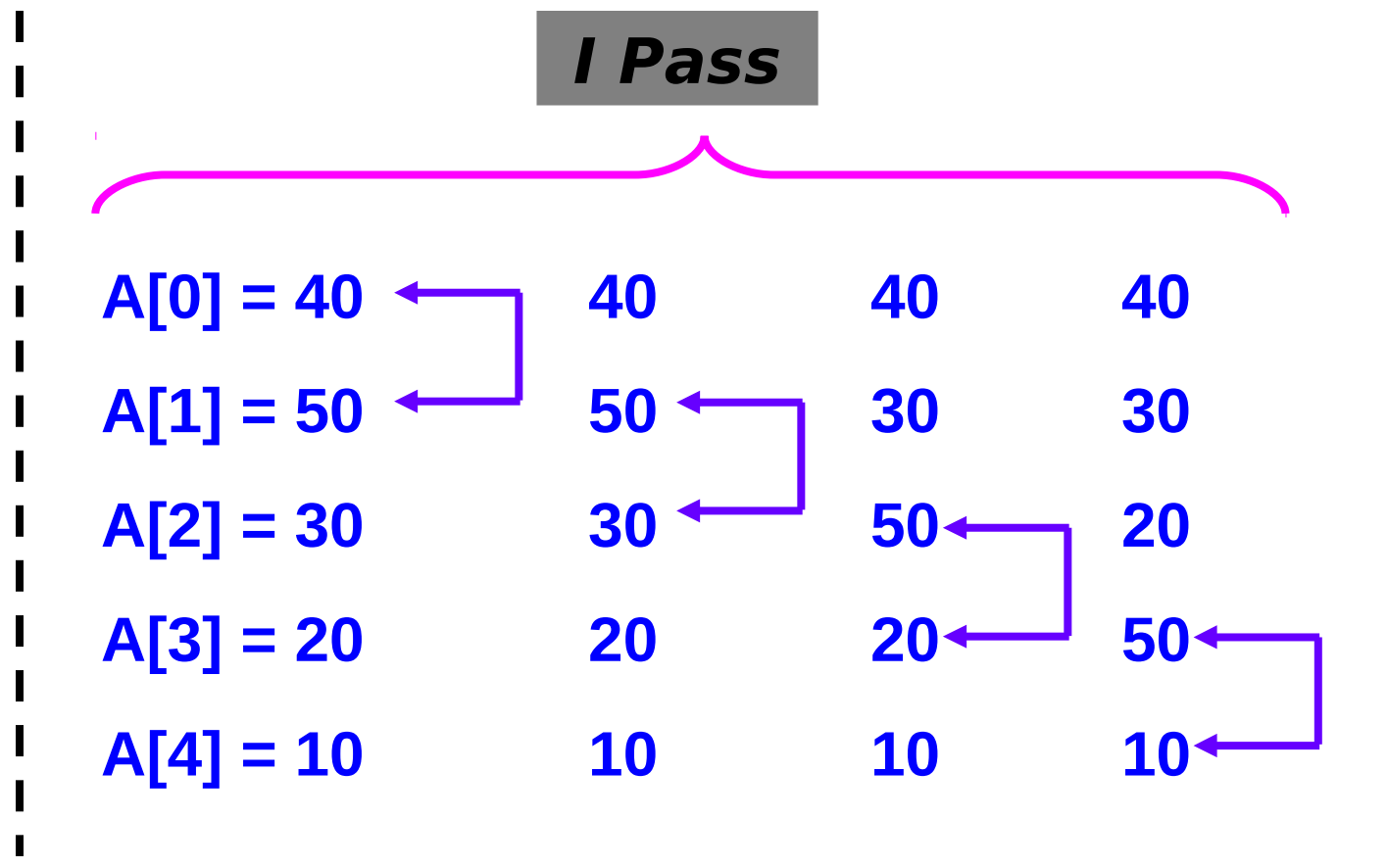
# List of Various Sorting Algorithms

- ◆ Bubble Sort
- ◆ Selection Sort
- ◆ Merge Sort
- ◆ Quick Sort
- ◆ Insertion Sort
- ◆ Shell Sort

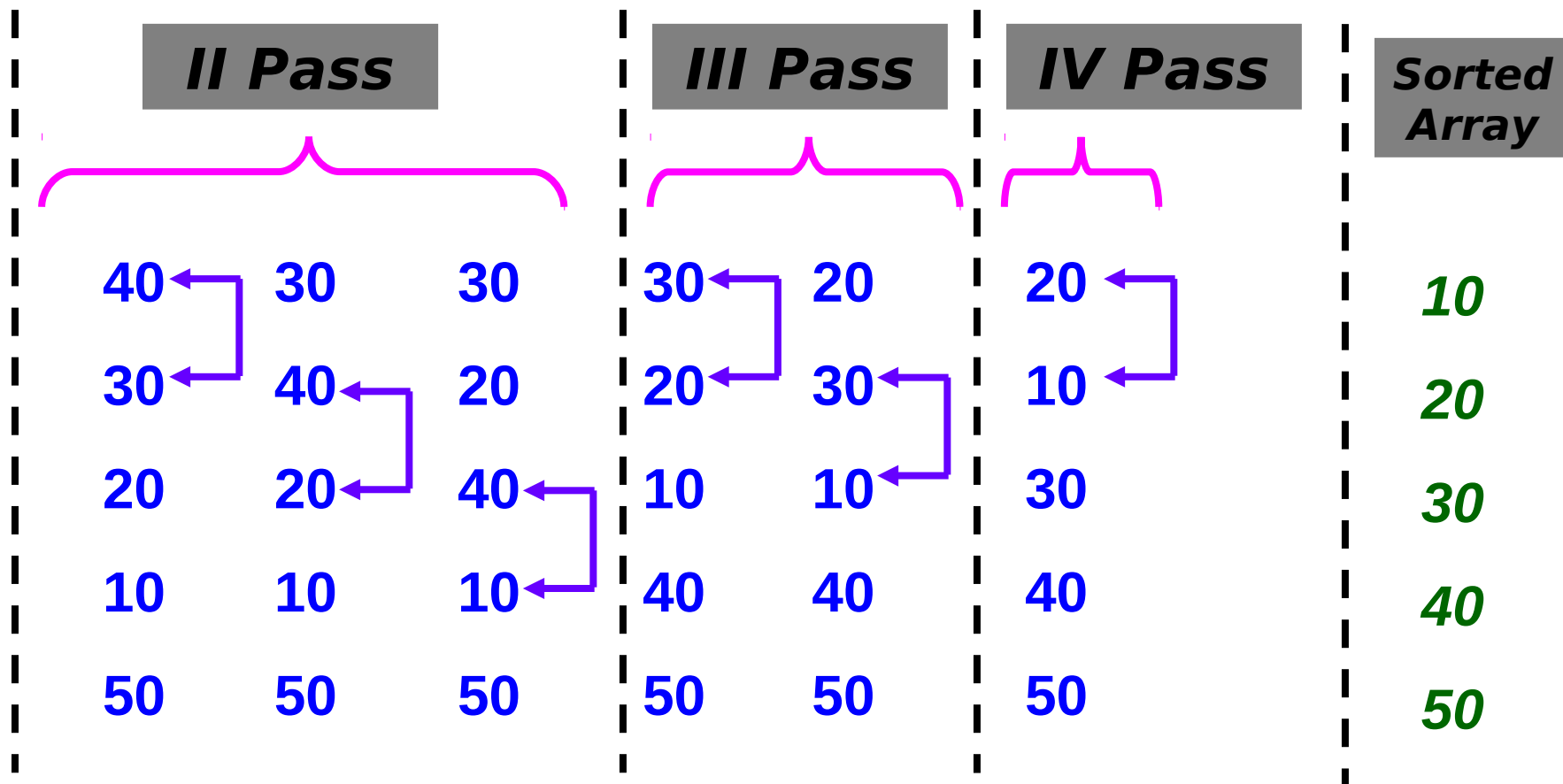
# Bubble Sort

- ◆ It is also known as *exchange sort*.
- ◆ It is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order.
- ◆ The pass through the list is repeated until no swaps are needed, which means the list is sorted.
- ◆ The algorithm gets its name from the way smaller elements "bubble" to the top (i.e., the beginning) of the list via the swaps.
- ◆ Because it only uses comparisons to operate on elements, it is a comparison sort. This is the easiest comparison sort to implement.

# Trace of a Bubble Sort



# contd...Trace of a Bubble Sort



# Worst Case Performance

- ◆ Bubble sort has worst-case complexity  $O(n^2)$  on lists of size  $n$ .
- ◆ Note that each element is moved no more than one step each time.
- ◆ No element can be more than a distance of  $n - 1$  away from its final sorted position, so we use at most  $n - 1 = O(n)$  operations to move an element to its final sorted position, and use no more than  $(n - 1)^2 = O(n^2)$  operations in the worst case.
- ◆ On a list where the smallest element is at the bottom, each pass through the list will only move it up by one step, so we will take  $n - 1$  passes to move it to its final sorted position.
- ◆ As each pass traverses the whole list a pass will take  $n - 1 = O(n)$  operations. Thus the number of operations in the worst case is also  $O(n^2)$ .



# Best Case Performance

- ◆ When a list is already sorted, bubble sort will pass through the list once, and find that it does not need to swap any elements. This means the list is already sorted.
- ◆ Thus bubble sort will take  $O(n)$  time when the list is completely sorted.
- ◆ It will also use considerably less time if the elements in the list are not too far from their sorted places.

# Selection Sort

- ◆ Selection sort is a sorting algorithm, specifically an in-place comparison sort.
- ◆ Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.
- ◆ It works as follows:
  1. Find the minimum value in the list
  2. Swap it with the value in the first position
  3. Repeat the steps above for remainder of the list (starting at the second position)

# Trace of a Selection Sort

Passes →	I	II	III	IV	V	VI
A[0] = 45	05	05	05	05	05	05
A[1] = 20	20	10	10	10	10	10
A[2] = 40	40	40	15	15	15	15
A[3] = 05	45	45	45	20	20	20
A[4] = 15	15	15	40	40	25	25
A[5] = 25	25	25	25	25	40	30
A[6] = 50	50	50	50	50	50	50
A[7] = 35	35	35	35	35	35	35
A[8] = 30	30	30	30	30	30	40
A[9] = 10	10	20	20	45	45	45

# contd....Trace of a Selection Sort

				<b><i>Sorted Array</i></b>	
VII	VIII	IX			
05	05	05		05	
10	10	10		10	
15	15	15		15	
20	20	20		20	
25	25	25		25	
30	30	30		30	
35	35	35		35	
50	40	40		40	
40	50	45		45	
45	45	50		50	

# Analysis

- ◆ Selection sort is very easy to analyze since none of the loops depend on the data in the array.
- ◆ Selecting the lowest element requires scanning all  $n$  elements (this takes  $n - 1$  comparisons) and then swapping it into the first position.
- ◆ Finding the next lowest element requires scanning the remaining  $n - 1$  elements and so on, for a total of  $(n - 1) + (n - 2) + \dots + 2 + 1 = O(n^2)$  comparisons.
- ◆ Each of these scans requires one swap for a total of  $n - 1$  swaps (the final element is already in place).
- ◆ Thus, the comparisons dominate the running time, which is  $O(n^2)$ .

# Insertion Sort

- ◆ Insertion sort is a simple sorting algorithm, a comparison sort in which the sorted array (or list) is built one entry at a time.
- ◆ Simple to implement.
- ◆ Efficient on small data sets.
- ◆ Efficient on data sets which are already substantially sorted
- ◆ Runs in  $O(n + d)$  time, where  $d$  is the number of inversions
- ◆ Stable (does not change the relative order of elements with equal keys)
- ◆ In-place (only requires a constant amount  $O(1)$  of extra memory space)
- ◆ It is an online algorithm, in that it can sort a list as it receives it.

# Trace: Insertion Sort

$a[0] = 20$

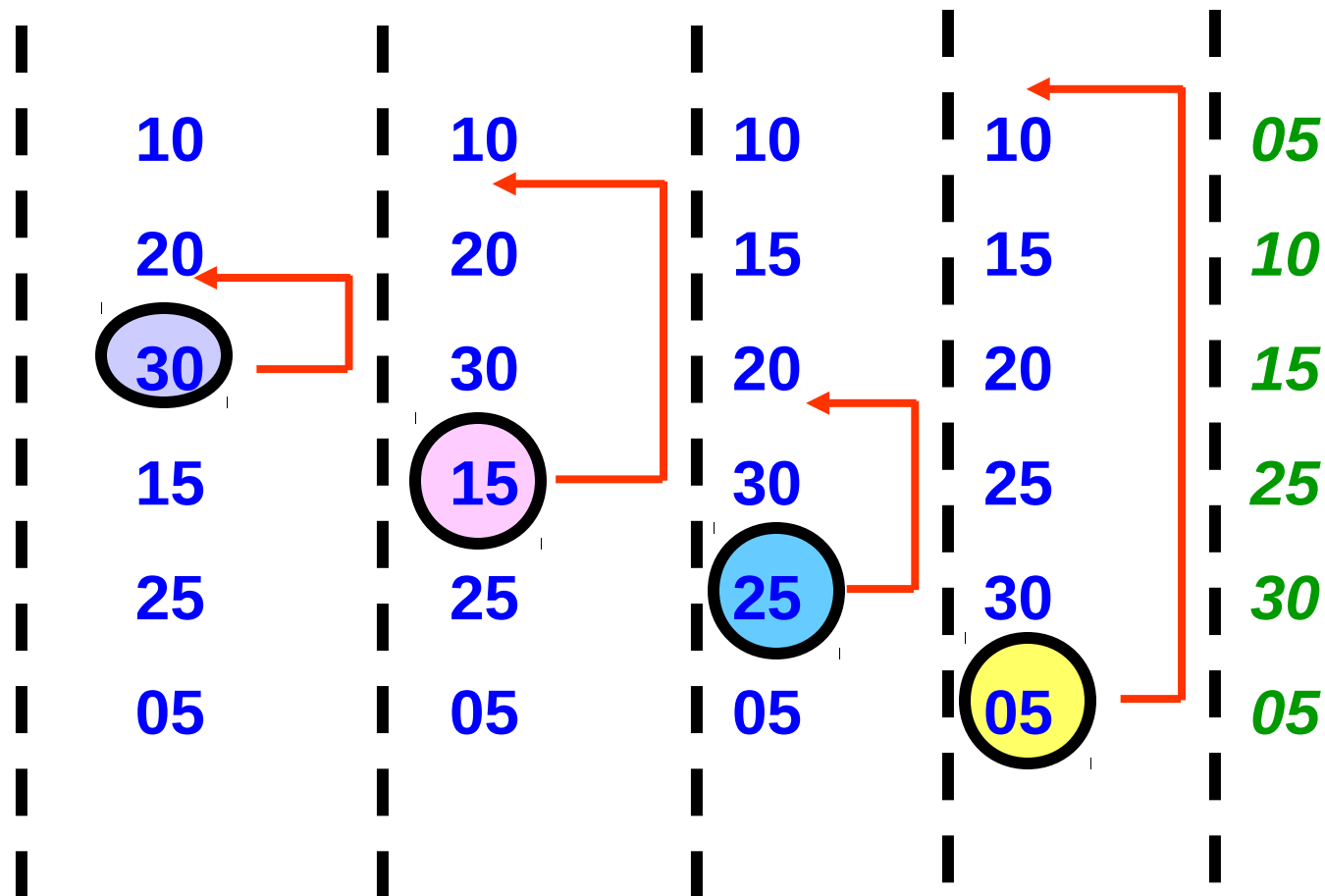
$a[1] = 10$

$a[2] = 30$

$a[3] = 15$

$a[4] = 25$

$a[5] = 05$



# Best Case: Insertion Sort

- ◆ In a sorted array, the implementation of insertion sort takes  $O(n)$  time.
- ◆ In each iteration, the first remaining element of the input is only compared with the last element of the sorted subsection of the array.
- ◆ Thus, if an array is sorted or nearly sorted, insertion sort will significantly outperform quick sort.



# Worst Case: Insertion Sort

- ◆ The worst case is an array sorted in reverse order, as every execution of the inner loop will have to scan and shift the entire sorted section of the array before inserting the next element.
- ◆ Insertion sort takes  $O(n^2)$  time in the worst case as well as in the average case, which makes it impractical for sorting large numbers of elements.
- ◆ However, insertion sort's inner loop is very fast, which often makes it one of the fastest algorithms for sorting small numbers of elements, typically less than 10 or so.

# Shell Sort

- ◆ Shell sort is a sorting algorithm which requires  $O(n^2)$  comparisons and exchanges in the worst case.
- ◆ Shell sort is a generalization of insertion sort, with two important observations:
  - Insertion sort is efficient if the input is "almost sorted"
  - Insertion sort is inefficient, on average, because it moves values just one position at a times
- ◆ Shell sort improves insertion sort by comparing elements separated by a gap of several positions.
- ◆ This lets an element take "bigger steps" toward its expected position. Multiple passes over the data are taken with smaller and smaller gap sizes.
- ◆ The last step of Shell sort is a plain insertion sort, but by then, the array of data is guaranteed to be almost sorted.

# Trace: Shell Sort

0      1      2      3      4      5      6      7      8      9      10      11      12

**45   36   75   20   05   90   80   65   30   50   10   75   85**

The distance between the elements to be compared = 3

The sub files generated with the distance of 3 are as follows:

Subfile 1                      a[0]      a[3]      a[6]      a[9]      a[12]

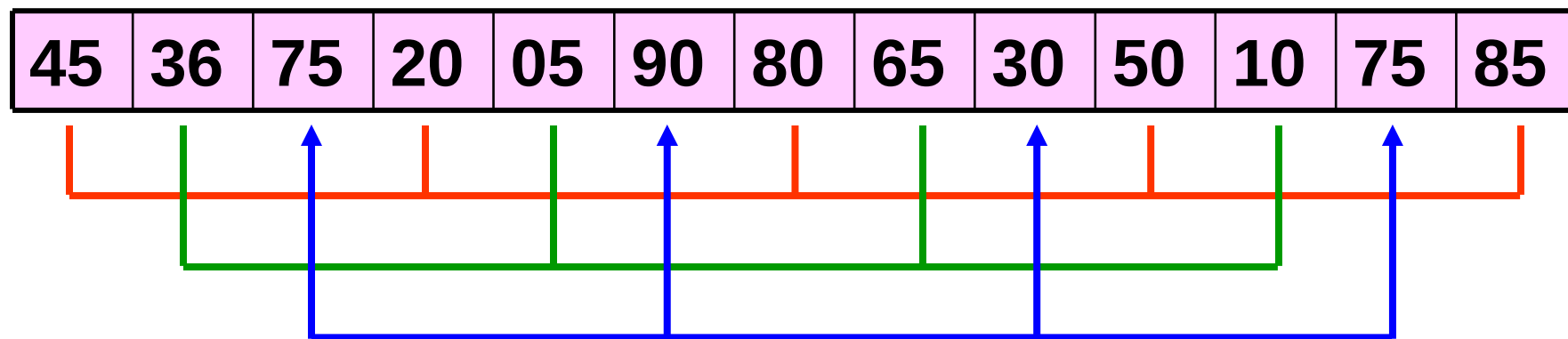
Subfile 2                      a[1]      a[4]      a[7]      a[10]

Subfile 3                      a[2]      a[5]      a[8]      a[11]

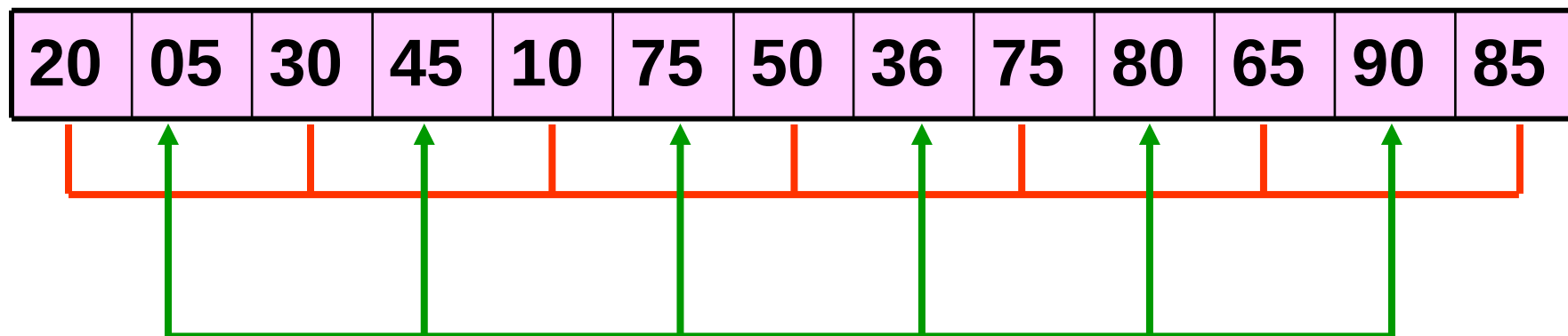
3                      3                      3

# Trace: Shell Sort

Input to Pass 1 with distance = 3



Output of Pass 1 is input to Pass 2 and distance = 2



## Trace: Shell Sort

Output of Pass 2 is input to Pass 3 and distance = 1

10	05	20	36	30	45	50	75	65	80	75	90	85

*Output of Pass 3*

05	10	20	30	36	45	50	65	75	75	80	85	90
----	----	----	----	----	----	----	----	----	----	----	----	----

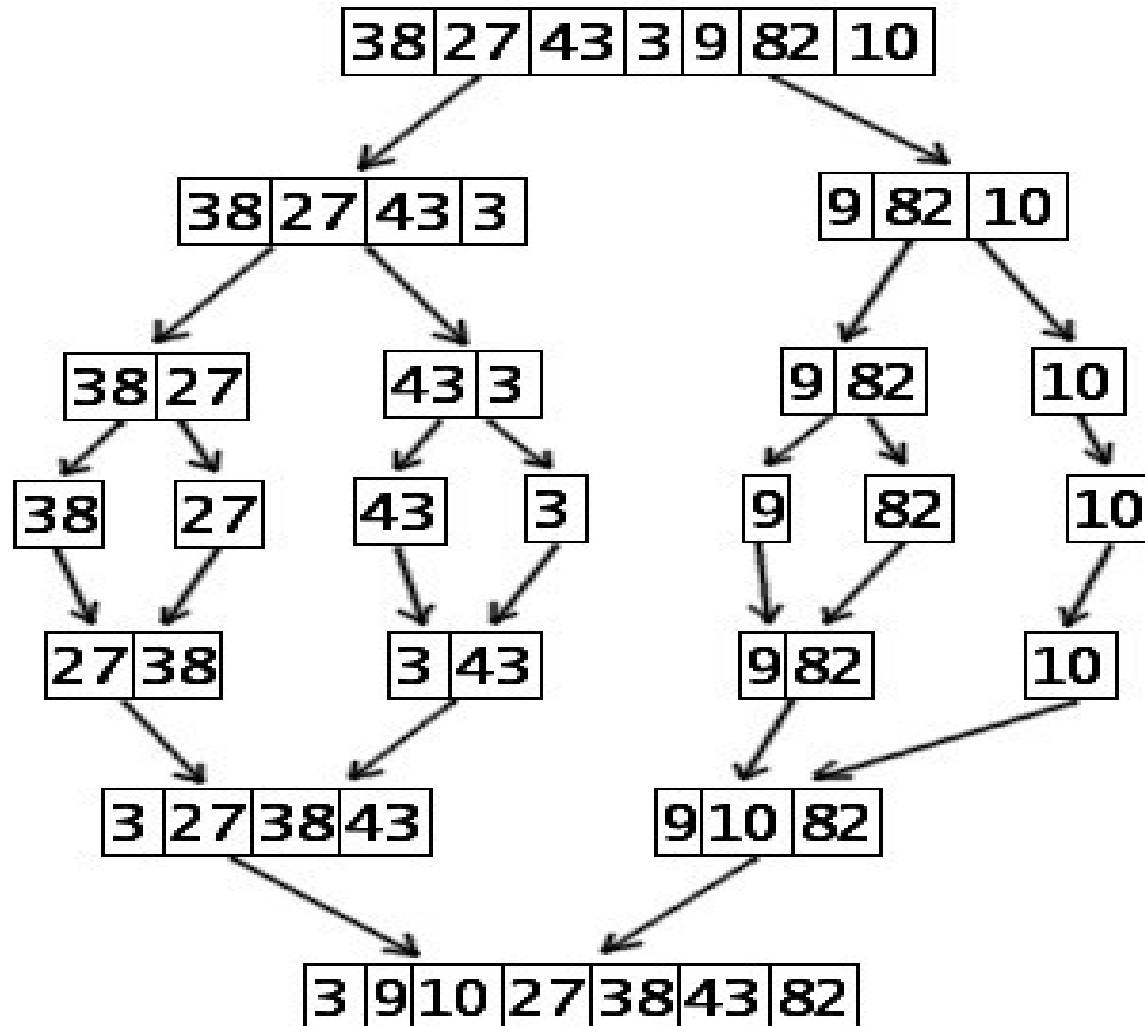
# Merge Sort

- ◆ Merge sort is a  $O(n \log n)$  sorting algorithm
- ◆ It is easy to implement merge sort such that it is stable, meaning it preserves the input order of equal elements in the sorted output
- ◆ It is an example of the *divide and conquer* algorithmic paradigm and are usually *recursive* in nature
- ◆ It is a comparison sort
- ◆ Space complexity is the main problem

# Merge Sort: Algorithm

- ◆ Divide the array into two equal parts.
- ◆ Recursively sort the left part of the array
- ◆ Recursively sort the right part of the array
- ◆ Merge the sorted left and the right part into a single sorted vector using the concept of Simple Merge.

# Trace of a Merge Sort





# Quick Sort

- ◆ Quick sort is a well-known sorting algorithm
- ◆ It is also called as *Partition Exchange Sort*
- ◆ On an average, makes  $O(n \log n)$  comparisons to sort  $n$  items
- ◆ However, in the worst case, it makes  $O(n^2)$  comparisons
- ◆ Typically, quick sort is significantly faster in practice than other  $O(n \log n)$  algorithms
  - because its inner loop can be efficiently implemented on most architectures, and
  - in most real-world data it is possible to make design choices which minimize the possibility of requiring time
- ◆ Quick sort is a comparison sort and, in efficient implementations, is not a stable sort


# Trace of a Quick Sort

low	i							high, j		
42	37	11	98	36	72	65	10	88	78	42>37 so, i++
		i								
42	37	11	98	36	72	65	10	88	78	42>11 so, i++
			i							
42	37	11	98	36	72	65	10	88	78	42>98 stop, i++, & compare 42 with a[j] = 78
				i						
42	37	11	98	36	72	65	10	88	78	42<78 so, j--

# Trace of a Quick Sort

			i					j		
42	37	11	98	36	72	65	10	88	78	42 < 88 so, j--

			i							
42	37	11	98	36	72	65	10	88	78	42<10 stop, j--



*Since  $i < j$ , exchange  $a[i]$  with  $a[j]$  and repeat the process*

			i					j		
42	37	11	10	36	72	65	98	88	78	42 > 10 so, i++

				i						
42	37	11	10	36	72	65	98	88	78	42>36 so, i++

# Trace of a Quick Sort

					i		j		
42	37	11	10	36	72	65	98	88	78

42 > 36 so, i++

					i		j		
42	37	11	10	36	72	65	98	88	78

42 > 72 stop,  
i++, &  
compare 42  
with a[j] = 98

					i		j		
42	37	11	10	36	72	65	98	88	78

42 < 98 so, j--

					i		j		
42	37	11	10	36	72	65	98	88	78

42 < 65 so, j--

# Trace of a Quick Sort

$i, j$

42	37	11	10	36	72	65	98	88	78
----	----	----	----	----	----	----	----	----	----

42 < 72 so, j--

$j \quad i$

42	37	11	10	36	72	65	98	88	78
----	----	----	----	----	----	----	----	----	----

42 < 36 FALSE

*Since  $i$  exceeds  $j$ , exchange  $a[\text{low}]$  with  $a[j]$ .*

$\text{low} \quad j \quad i$

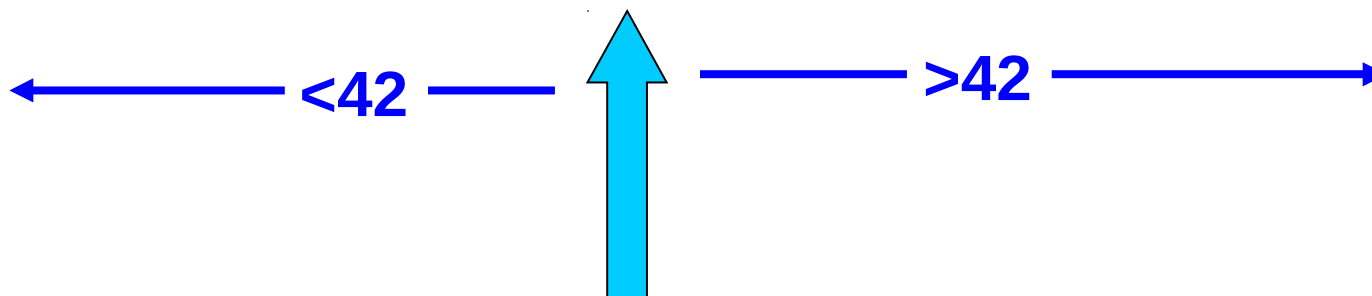
42	37	11	10	36	72	65	98	88	78
----	----	----	----	----	----	----	----	----	----



# Trace of a Quick Sort

*The sorted array after partition*

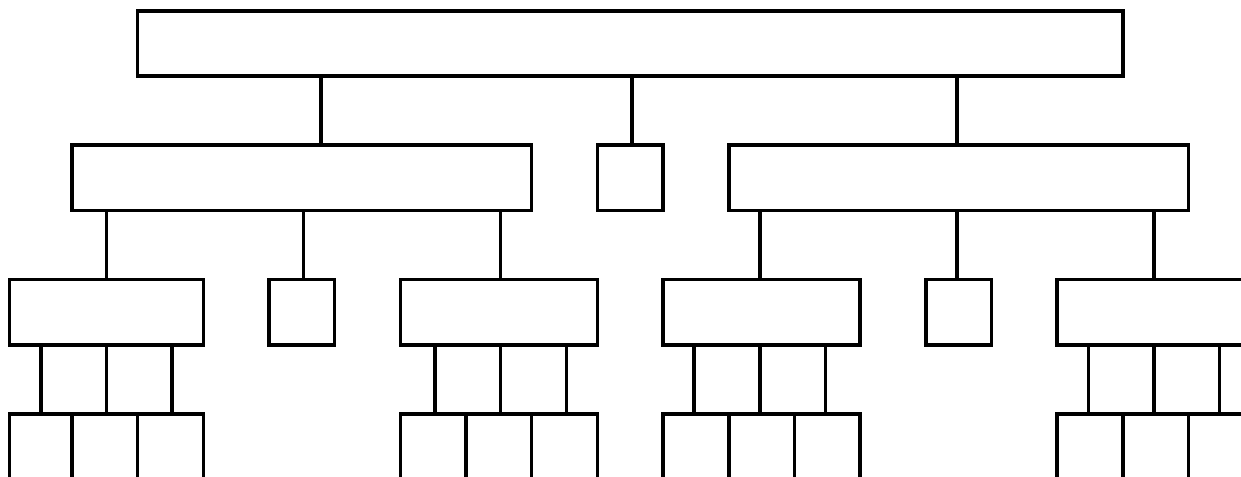
36	37	11	10	42	72	65	98	88	78
----	----	----	----	----	----	----	----	----	----



*Pivot Element*

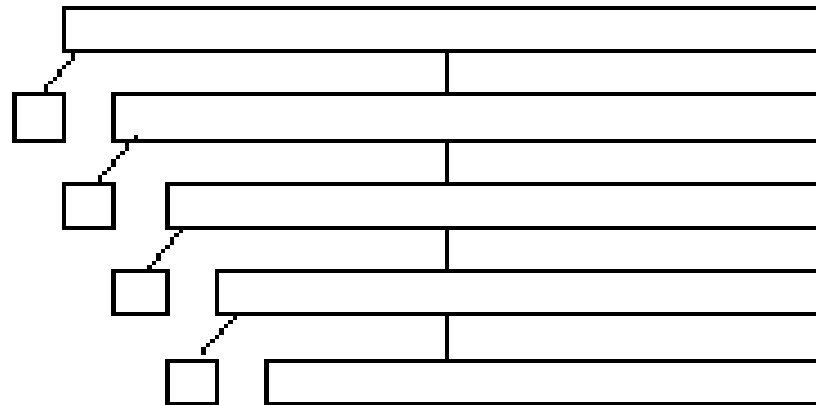
# Best Case for Quick Sort

- ◆ The recursion tree for the best case is as shown below
- ◆ The total partitioning on each level is  $O(n)$ , and it takes levels of perfect partitions to get to single element sub problems.
- ◆ When we are down to single elements, the problems are sorted. Thus the total time in the best case is  $O(n \log n)$ .



# Worst Case for Quick Sort

- ◆ Instead of  $n/2$  elements in the smaller half, we get zero, meaning that the pivot element is the biggest or smallest element in the array.





# Time Complexities

Algorithm	Best	Average	Worst
Bubble Sort	$O(n)$	--	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n + d)$	$O(n^2)$
Shell Sort	--	--	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

# Searching Techniques

# Linear Search

- ◆ Linear search is a search algorithm, also known as **sequential search**, that is suitable for searching a set of data for a particular value.
- ◆ It operates by checking every element of a list one at a time in sequence until a match is found.
- ◆ Linear search runs in  $O(N)$ . If the data is distributed randomly, on average  $N/2$  comparisons will be needed.
- ◆ The best case is that the value is equal to the first element tested, in which case only 1 comparison is needed.
- ◆ The worst case is that the value is not in the list (or is the last item in the list), in which case  $N$  comparisons are needed.

# Binary Search

- ◆ A binary search algorithm is a technique for finding a particular value in a linear array, by ruling out half of the data at each step.
- ◆ A binary search finds the median, makes a comparison to determine whether the desired value comes before or after it, and then searches the remaining half in the same manner.
- ◆ A binary search is an example of a divide and conquer algorithm

# Binary Search Trees

# Binary Search Trees

A Binary Search Tree (BST) is

A binary tree storing keys (or key-value entries) at its internal nodes

And satisfying the following **BST Property**:

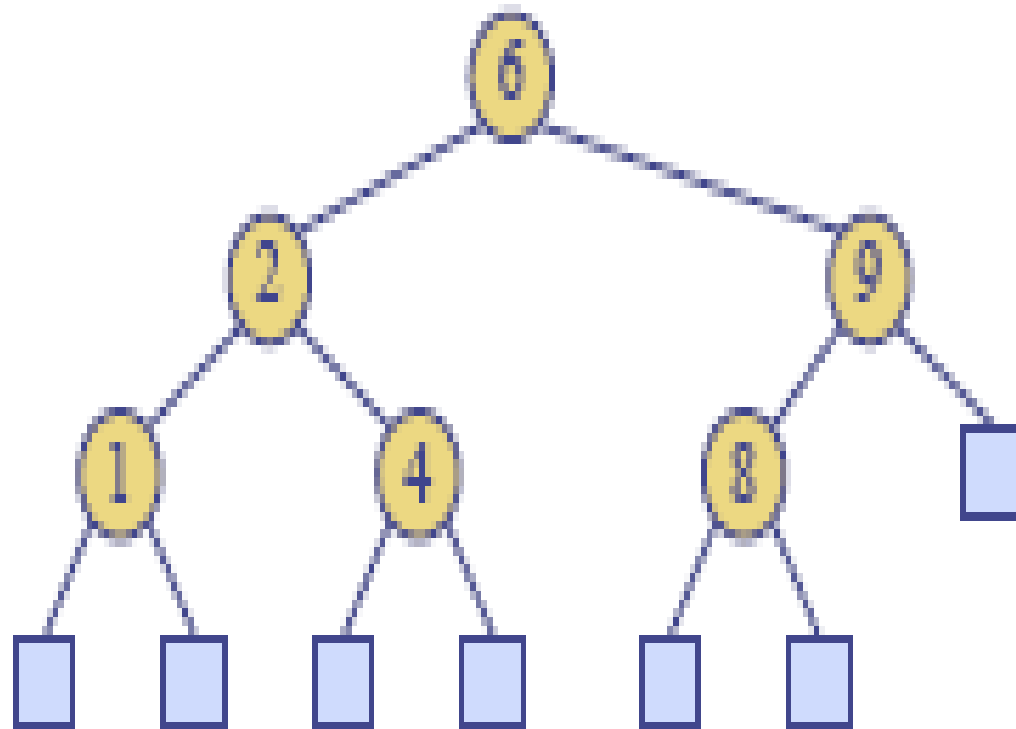
Let  $u$ ,  $v$ , and  $w$  be three nodes such that

- $u$  is in the left subtree of  $v$
- $w$  is in the right subtree of  $v$
- Then,  $\text{key}(u) \leq \text{key}(v) \leq \text{key}(w)$

External nodes do not store items

An in-order traversal of a binary search trees visits the keys in increasing order

# Illustration



# Search

To search for a key  $k$

- Trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of  $k$  with the key of the current node
- If there is match declare success
- If a leaf is reached and the key is not found return null
- TreeSearch algorithm; see next slide



# TreeSearch Algorithm

**Algorithm** *TreeSearch*(*k*, *v*)

**if** *T.isExternal* (*v*)

**return** *v*

**if** *k* < *key*(*v*)

**return** *TreeSearch*(*k*, *T.left*(*v*))

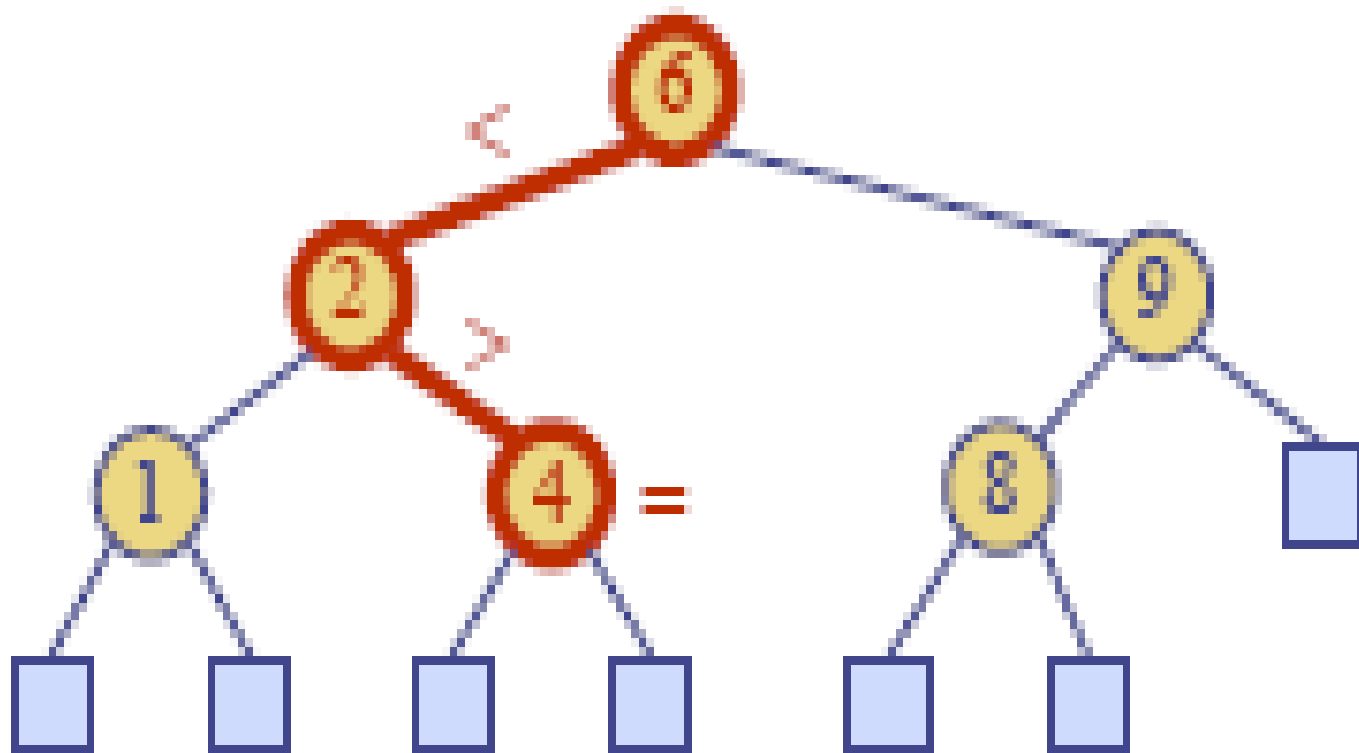
**else if** *k* = *key*(*v*)

**return** *v*

**else** { *k* > *key*(*v*) }

**return** *TreeSearch*(*k*, *T.right*(*v*))

## Call TreeSearch(4, root)



# Insertion

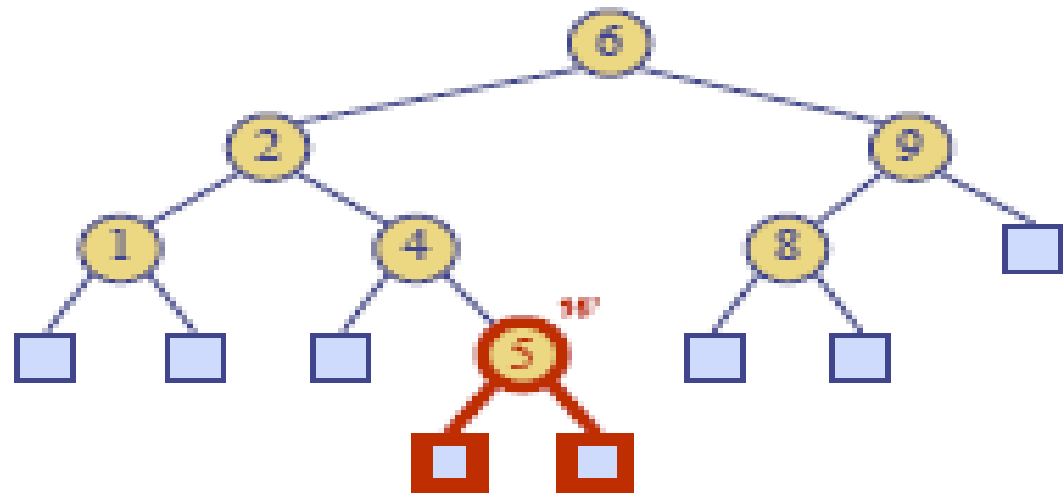
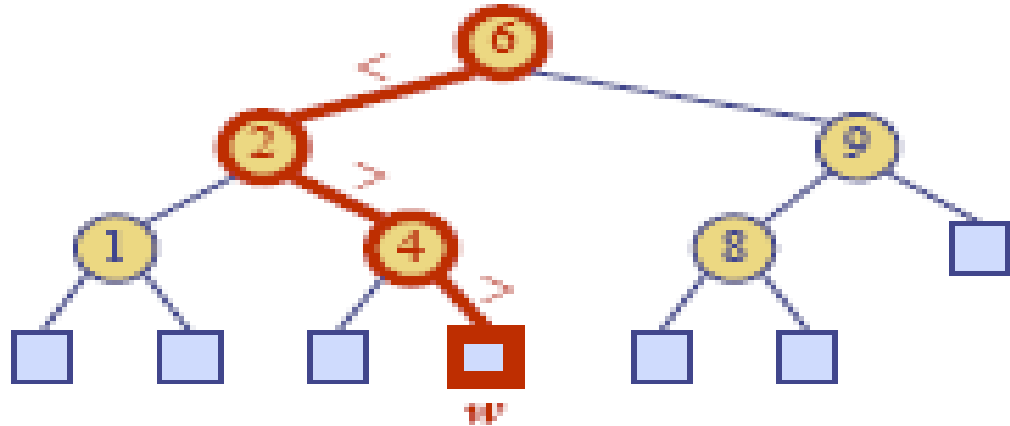
**Insert**(k, o), we search for key k (using TreeSearch)

Assume k is not already in the tree, and let w be the leaf reached by the search

We insert k at node w and expand w into an internal node

# Insert Example

Example: insert (5)



# Deletion

To perform operation  $\text{remove}(k)$ , we search for key  $k$

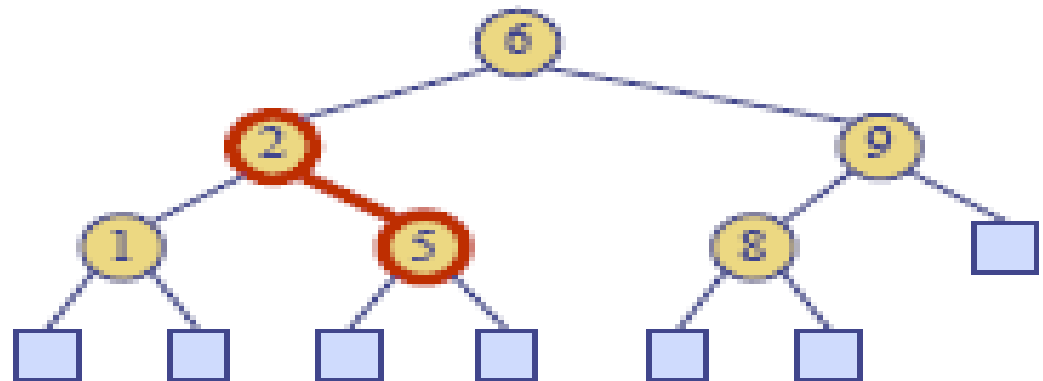
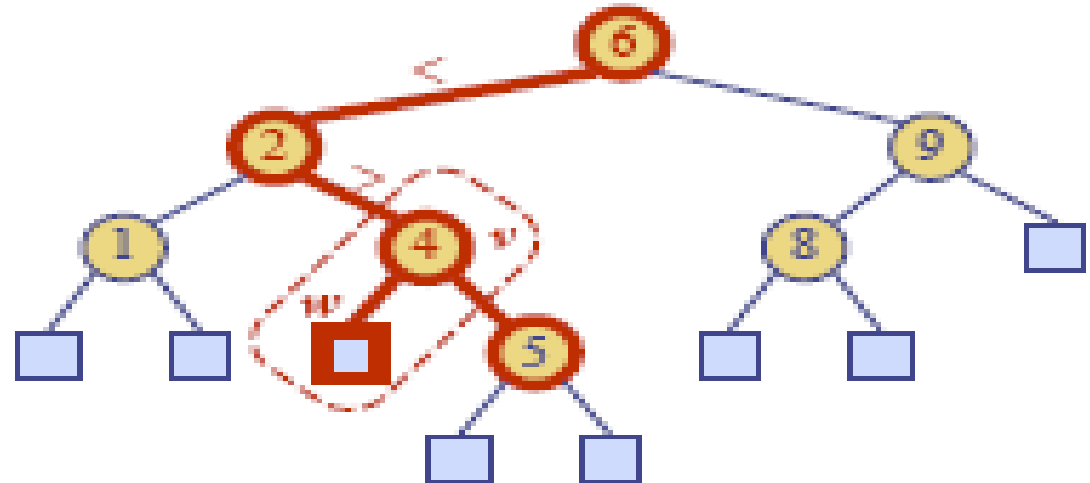
Assume key  $k$  is in the tree, and let  $v$  be the node storing  $k$

If node  $v$  has a leaf child  $w$ , we remove  $v$  and  $w$  from the tree with operation

$\text{removeExternal}(w)$ , which removes  $w$  and its parent

# Deletion Example

Example:  
remove (4)



# Deletion

Consider the case where the key  $k$  to be removed is stored at a node  $v$  whose children are both internal

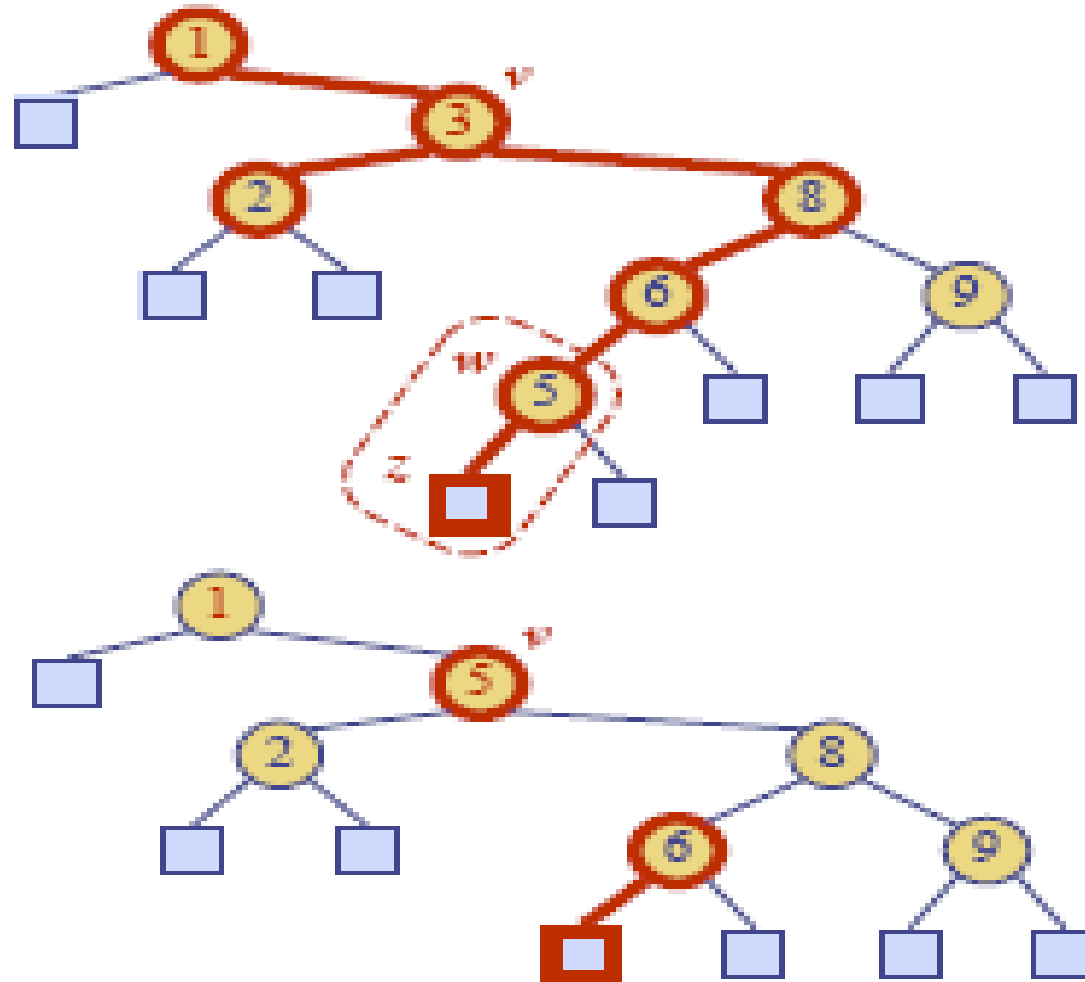
Find the internal node  $w$  that follows  $v$  in an inorder traversal

Copy  $key(w)$  into node  $v$

Remove node  $w$  and its left child  $z$  (which must be a leaf) by means of operation `removeExternal(z)`

# Deletion Example

Example:  
remove (3)





# Performance

Consider a dictionary with  $n$  items implemented by means of a binary search tree of height  $h$

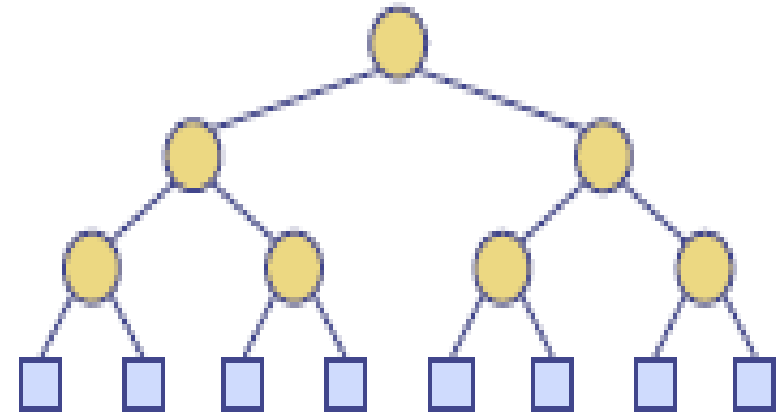
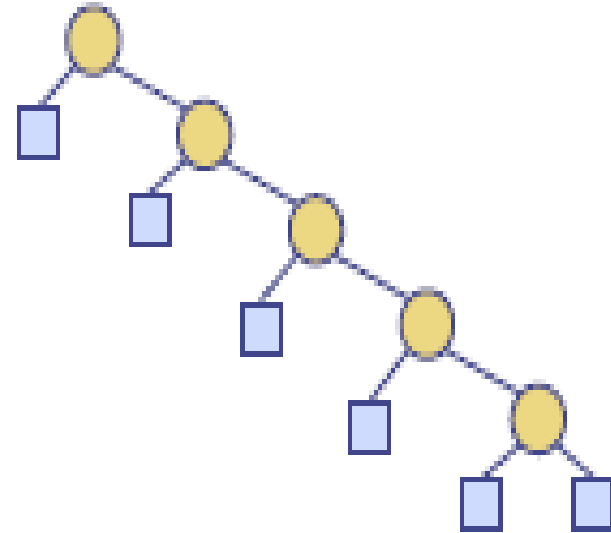
Space used is  $O(n)$

**find**, **insert** and **remove** take  $O(h)$  time

The height  $h$ :

$O(n)$  in the worst case

$O(\log n)$  in the best case



# Sorting using Binary Search Trees

- Consider a  $n$  keys which are comparable
- We can use a BST to *sort* the keys!
- How?
  - Insert the items one-by-one to form a BST
  - Now traverse the BST in-order
- What is the resultant sequence?
  - A (increasing) sorted sequence of the keys
- Performance
  - $O(n^2)$  in the worst case
  - $O(n \log n)$  in the best case

# Summary

A sorting algorithm is an algorithm that puts elements of a list in a certain order.

Bubble sort is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order.

Selection sort is a comparison sort in which the minimum element is exchanged with the first position repeatedly.

Insertion sort is a comparison sort in which the sorted array (or list) is built one entry at a time

Selection and Insertion sort algorithms can be implemented using Priority Queue ADT

Quick sort is the most used sorting algorithm. Its worst case performance is  $O(n^2)$ , its average case performance is  $O(n \log n)$ .

Quick sort is superior to other  $O(n \log n)$  algorithms in practice

# Summary

Linear search is a search algorithm, also known as sequential search, that is suitable for searching a set of data for a particular value.

A binary search algorithm is a technique for finding a particular value in a linear array, by ruling out half of the data at each step

Efficient Searching and Sorting algorithms can be designed and analysed using Binary Search Trees

The height of a Binary Search Tree is crucial for efficient implementation of algorithms using it