Introduction to Data Structures

Unit – 1

Content

Data Structures –Definition, Classification and Operations on Data Structures, Pseudo code, Algorithm analysis, Time and Space Complexity.
Searching: Linear search, Binary search.
Sorting: Insertion Sort, Selection Sort, Exchange (Bubble Sort, Quick Sort), merging (Merge) sort, distribution (Radix Sort) algorithms.

Definition:

- ✓ A ***data structure*** is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.
- ✓ Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables.

Data structures are widely applied in the following areas:

- ✓ Compiler design
- ✓ Operating system
- ✓ Statistical analysis package
- ✓ DBMS
- ✓ Numerical analysis
- ✓ Simulation
- ✓ Artificial intelligence
- ✓ Graphics

Elementary Data Structure Organization

- ✓ Data structures are building blocks of a program.
- ✓ A program built using improper data structures may not work as expected.
- ✓ So as a programmer it is mandatory to choose most appropriate data structures for a program.
- ✓ The term *data* means a value or set of values.
- ✓ A ***record*** is a collection of data items.
- ✓ For example, the name, address, course, and marks obtained are individual data items.
- ✓ But all these data items can be grouped together to form a record.
- ✓ A ***file*** is a collection of related records.
- ✓ For example, if there are 60 students in a class, then there are 60 records of the students.
- ✓ All these related records are stored in a file.
- ✓ Similarly, we can have a file of all the employees working in an organization, a file of all the customers of a company, a file of all the suppliers, and so on.
- ✓ It specifies either the value of a variable or a constant (e.g., marks of students, name of an employee, address of a customer, value of *pi*, etc.).

2. CLASSIFICATION OF DATA STRUCTURES

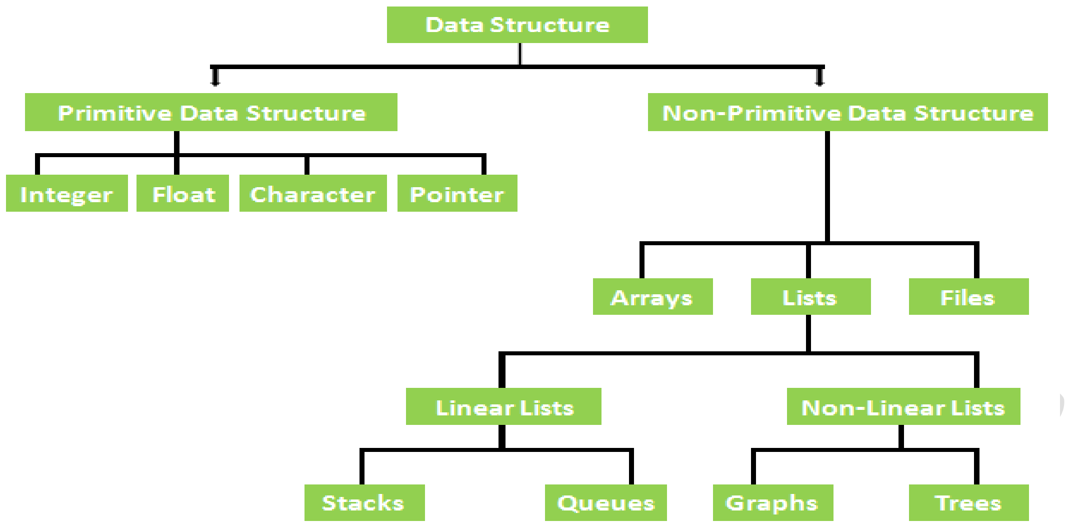
Primitive data structures are the fundamental data types which are supported by a programming language.

- ✓ Some basic data types are integer, real, character, and boolean.
- ✓ The terms 'data type', 'basic data type', and 'primitive data type' are often used interchangeably.

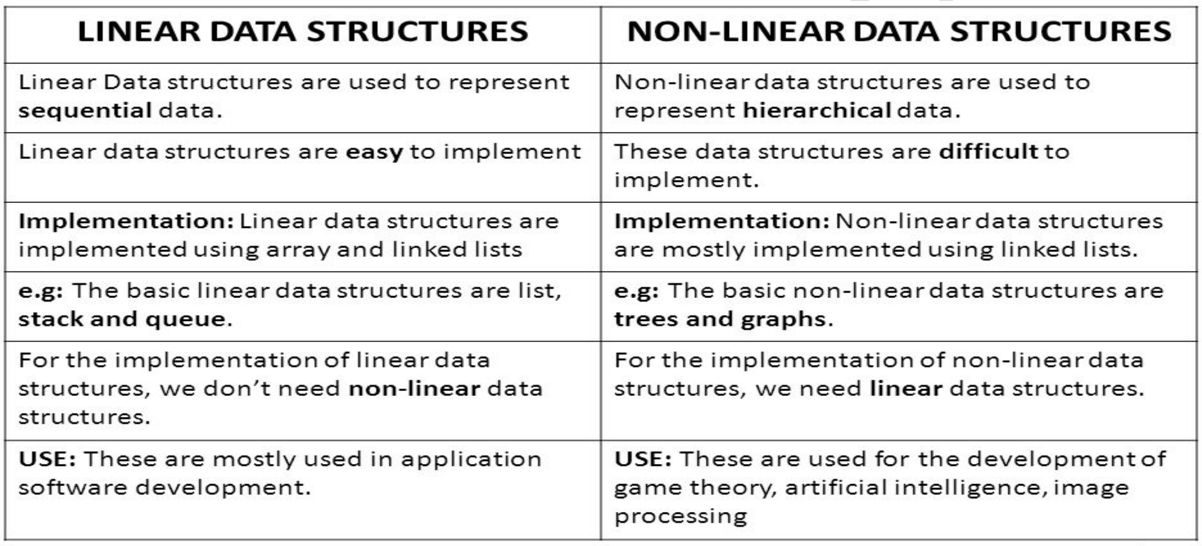
Non-primitive data structures are those data structures which are created using primitive data structures.

- ✓ Examples of such data structures include linked lists, stacks, trees, and graphs.

- ✓ Non-primitive data structures can further be classified into two categories: *linear* and *non-linear* data structures



Difference in Linear and Non-Linear Data Structures:



LINEAR DATA STRUCTURES	NON-LINEAR DATA STRUCTURES
Linear Data structures are used to represent **sequential** data.	Non-linear data structures are used to represent **hierarchical** data.
Linear data structures are **easy** to implement	These data structures are **difficult** to implement.
Implementation: Linear data structures are implemented using array and linked lists	**Implementation:** Non-linear data structures are mostly implemented using linked lists.
e.g: The basic linear data structures are list, **stack and queue**.	**e.g:** The basic non-linear data structures are **trees and graphs**.
For the implementation of linear data structures, we don't need **non-linear** data structures.	For the implementation of non-linear data structures, we need **linear** data structures.
USE: These are mostly used in application software development.	**USE:** These are used for the development of game theory, artificial intelligence, image processing

Different Data Structures

a. Arrays

- ✓ An array is a collection of similar data elements (same data type).
- ✓ The elements of the array are stored in consecutive memory locations and are referenced by an *index* (also known as the *subscript*).
- ✓ In C, arrays are declared using the following syntax:

type name[size];

- ✓ Example: int marks[10];
- ✓ The above statement declares an array marks that contains 10 elements.
- ✓ In C, the array index starts from zero.
- ✓ This means that the array marks will contain 10 elements in all.

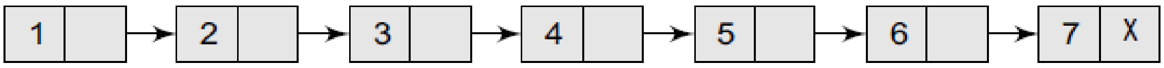
1st element	2nd element	3rd element	4th element	5th element	6th element	7th element	8th element	9th element	10th element
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]	marks[8]	marks[9]

- ✓ The first element will be stored in marks[0], second element in marks[1], so on and so forth.

- ✓ Therefore, the last element, that is the 10th element, will be stored in marks[9].
- ✓ Arrays are generally used when we want to store large amount of similar type of data.
- ✓ But they have the following limitations:
- Arrays are of fixed size.
- Data elements are stored in contiguous memory locations which may not be always available.
- Insertion and deletion of elements can be problematic because of shifting of elements from their positions.

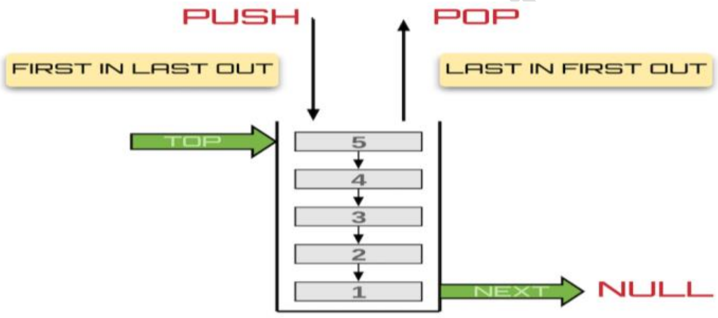
b. Linked Lists:

- ✓ A linked list is a very flexible, dynamic data structure in which elements called *nodes,* form a sequential list.
- ✓ In linked list, every node contains the following two types of data:
- The value of the node or any other data that corresponds to that node
- A pointer or link to the next node in the list



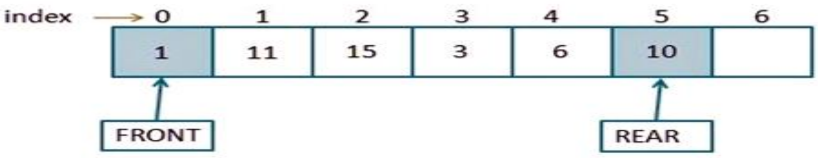
c. Stack

- ✓ A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack.
- ✓ Stack is called a Last-In, First-Out (LIFO) structure because the last element which is added to the stack is the first element to deleted from the stack.



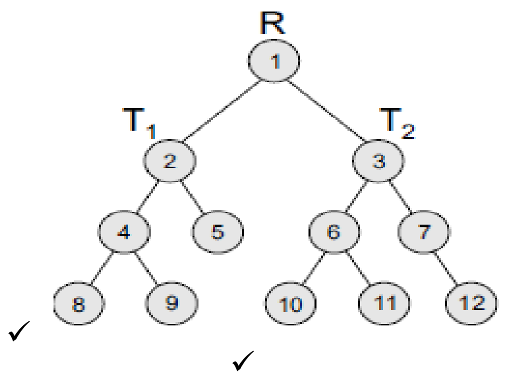
d. Queue

- ✓ A queue is a First-In, First-Out (FIFO) data structure in which the element that is inserted first is the first one to be taken out.
- ✓ The elements in a queue are added at one end called the rear and removed from the other end called the front.
- ✓ Stacks, queues can be implemented by using either arrays or linked lists.



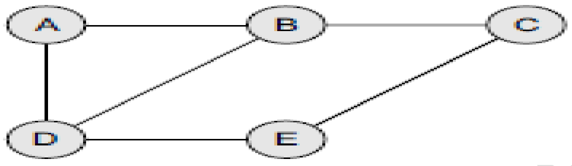
e. Trees

- ✓ A tree is a non-linear data structure which consists of a collection of nodes or *vertices* and *edges* that arranged in a hierarchical order.
- ✓ One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root.
- ✓ The root element is the topmost node which is pointed by a ‘root’ pointer.
- ✓ If root = NULL then the tree is empty.



f. Graphs

- ✓ A graph is a non-linear data structure which is a collection of *vertices* (also called *nodes*) and *edges* that connect these vertices.
- ✓ A graph is often viewed as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships between the nodes can exist.



3. Operations on Data Structures

1. ***Traversing:*** It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.
2. ***Searching:*** It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in mathematics.
3. ***Inserting:*** It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.
4. ***Deleting:*** It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.
5. ***Sorting:*** Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.
6. ***Merging:*** Lists of two sorted data items can be combined to form a single list of sorted data items.

4. Pseudo code

- ✓ Pseudocode is one of the tools that can be used to write a preliminary plan that can be developed into a computer program.
- ✓ Pseudocode is a generic way of describing an algorithm without use of any specific programming language syntax.
- ✓ **Pseudocode is a "text-based" detail (algorithmic design tool).**
- ✓ Pseudo code can be broken down into five components.
  - ➢ **Variables**
  - ➢ **Assignment**
  - ➢ **Input/output**
  - ➢ **Selection**
  - ➢ **Repetition**

Example:

```
Step 1: Begin
Step 2: read grade
Step 3: If student's grade is greater than or equal to 60
                Print "passed"
        else
                Print "failed“
Step 4: End
```

5. Algorithm:

- ✓ An algorithm is a set of well-defined instructions in sequence to solve a problem.
- ✓ It is considered to be an effective procedure for solving a problem in finite number of steps.
- ✓ A well-defined algorithm always provides an answer and is guaranteed to terminate.
- ✓ Algorithms are mainly used to achieve *software reuse*, once we have an idea or a blueprint of a solution.

Characteristics of an Algorithm

- ➢ **Input:** An algorithm must have 0 or well defined inputs.
- ➢ **Output:** An algorithm must have 1 or well defined outputs, and should match with the desired output.
- ➢ **Finiteness:** An algorithm must be terminated after the finite number of steps.
- ➢ **Independent:** An algorithm must have step-by-step directions which is independent of any programming code.
- ➢ **Definiteness:** An algorithm must be unambiguous and clear. Each of their steps and input/outputs must be clear and meaningful.
- ➢

Example:

Write an algorithm to add two numbers entered by the user

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
        sum←num1+num2
Step 5: Display sum
Step 6: Stop
```

Write an algorithm to find the largest among three different numbers entered by the user.

```
Step 1: Start
Step 2: Declare variables a, b and c.
Step 3: Read variables a, b and c.
Step 4: If a > b
                If a > c        Display a is the largest number.
                Else    Display c is the largest number.
        Else
                If b > c        Display b is the largest number.
                Else    Display c is the greatest number.
Step 5: Stop.
```

5. Analysis of Algorithms - Time and Space Complexity

- ✓ Analysing an algorithm means determining the amount of resources (such as time and memory) needed to execute it.
- ✓ Algorithms are generally designed to work with an arbitrary number of inputs, so the efficiency or complexity of an algorithm is stated in terms of time and space complexity.
- ✓ The ***time complexity*** of an algorithm is basically the running time of a program as a function of the input size.
- ✓ The ***space complexity*** of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.

a. Space Complexity:

The space needed by a program depends on the following two parts:

- ➢ *Fixed part: A fixed part that is a space required to store certain data and variables (i.e. simple variables and constants, program size etc.), that are not dependent of the size of the problem.*
- ➢ *Variable part: A variable part is a space required by variables, whose size is totally dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation etc.*

b. Time Complexity

1.Worst-case running time:

- ✓ This denotes the behaviour of an algorithm with respect to the worst possible case of the input instance.
- ✓ The worst-case running time of an algorithm is an upper bound on the running time for any input.
- ✓ Therefore, having the knowledge of worst-case running time gives us an assurance that the algorithm will never go beyond this time limit.

2. Average-case running time

- ✓ The average-case running time of an algorithm is an estimate of the running time for an ‘average’ input.
- ✓ It specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution.
- ✓ Average-case running time assumes that all inputs of a given size are equally likely.

3. Best-case running time

- ✓ The term ‘best-case performance’ is used to analyse an algorithm under optimal conditions.
- ✓ For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list.
- ✓ However, while developing and choosing an algorithm to solve a problem, we hardly base our decision on the best-case performance.
- ✓ It is always recommended to improve the average performance and the worst-case performance of an algorithm.

4. Amortized running time

- ✓ Amortized running time refers to the time required to perform a sequence of (related) operations averaged over all the operations performed.
- ✓ Amortized analysis guarantees the average performance of each operation in the worst case.

Algorithm Efficiency:

- ✓ *A function is linear (without any loops or recursions), the efficiency of that algorithm or the running time of that algorithm can be given as the number of instructions it contains.*

- ✓ If an algorithm contains *loops*, then the efficiency of that algorithm may *vary* depending on the number of loops and the running time of each loop in the algorithm.

Types:

- *Linear Loops*
- *Logarithmic Loops*
- *Nested Loops*

Asymptotic notations:

- ✓ When it comes to analysing the complexity of any algorithm in terms of time and space, we can never provide an exact number to define the time required and the space required by the algorithm, instead we express it using some standard notations, also known as **Asymptotic Notations**.
- ✓ We use three types of asymptotic notations to represent the growth of any algorithm, as input increases:
- Big Theta (Θ)
- Big Oh(O)
- Big Omega (Ω)

a. Big O notation:

The Big O notation, where O stands for 'Order of', is concerned with what happens for very large values of n.

For example, A constant-time function/method is "order 1" : O(1)

A linear-time function/method is "order N" : O(N)

A quadratic-time function is "order N squared" : $O(N^2)$

Definition: Let g and f be functions from the set of natural numbers. The function f is said to be O(g) (read big-oh of g), if there is a constant $c > 0$ and a natural number n_0 such that $f(n) \leq c*g(n)$ for all $n >= n_0$.

Categories of Algorithms

- ✓ According to the Big O notation, we have five different categories of algorithms:
- ▪ Constant time algorithm: running time complexity given as O(1)
- ▪ Linear time algorithm: running time complexity given as O(n)
- ▪ Logarithmic time algorithm: running time complexity given as O(log n)
- ▪ Polynomial time algorithm: running time complexity given as O(nk) where k > 1
- ▪ Exponential time algorithm: running time complexity given as O(2n)`

Limitations of Big O Notations:

There are certain limitations with the Big O notation of expressing the complexity of algorithms. These limitations are as follows:

- ➢ Many algorithms are too hard to analyse mathematically.
- ➢ There may not be sufficient information to calculate the behaviour of the algorithm in the average case.
- ➢ Big O analysis only tells us how the algorithm grows with the size of the problem, not *how efficient it is,* as it does not consider the programming effort.
- ➢ It ignores important constants. For example, if one algorithm takes $O(n^2)$ time to execute and the other takes $O(100000n^2)$ time to execute, then as per Big O, both algorithm have equal time complexity. In real-time systems, this may be a serious consideration.

b. OMEGA NOTATION (Ω)

- ➢ The Omega notation provides a tight lower bound for f(n).
- ➢ It means that the function can never do better than the specified value but it may do worst.
- ➢ Ω notation is simply written as, $f(n) \in \Omega(g(n))$, where n is the problem size and $\Omega(g(n)) = \{h(n): \exists$ positive constants $c > 0, n_0$ such that $\mathbf{0 \leq cg(n) \leq h(n)}, \forall\ n \geq n0\}$.

➢ Hence, we can say that Ω(g(n)) comprises a set of all the functions h(n) that are greater than or equal to cg(n) for all values of $n \geq n_0$.

Summarize:

- ✓ Best case Ω describes a lower bound for all combinations of input. This implies that the function can never get any better than the specified value. For example, when sorting an array the best case is when the array is already correctly sorted.
- ✓ Worst case Ω describes a lower bound for worst case input combinations. It is possibly greater than best case. For example, when sorting an array the worst case is when the array is sorted in reverse order. If we simply write Ω, it means same as best case Ω.

Example: f(n)= 8n+7 and g(n)=n.

We know that => f(n)>=c*g(n)

8n+7 >= c*n

8n+7 >= n where c=1, no>=1

And we can write the above statement

<u>8n+7 >= Ω (n).</u>

b. THETA Notation (θ):

- ✓ Theta notation provides an asymptotically tight bound for f(n).
- ✓ Θ notation is simply written as, f(n) ∈ Θ(g(n)), where n is the problem size and Θ(g(n)) = {h(n): ∃ positive constants c_1, c_2, and n_0 such that $\mathbf{0 \leq c_1 g(n) \leq h(n) \leq c_2 g(n)}$, ∀ n ≥ n0}.
- ✓ Hence, we can say that Θ(g(n)) comprises a set of all the functions h(n) that are between $c_1g(n)$ and $c_2g(n)$ for all values of $n \geq n_0$.

Example: f(n)= 8n+7 and g(n)=n.

We know that => $\mathbf{c_1 g(n) \leq f(n) \leq c_2 g(n)}$

Separate the above statement in two statements:

$f(n) <= c_2g(n)$ and $f(n) >= c_1g(n)$

8n+7 <= 5*n 8n+7 <= n

Where c_2 is taken as 5 and n0 >=0 where c_1 is taken as 1 and n0 >=0

To summarize:

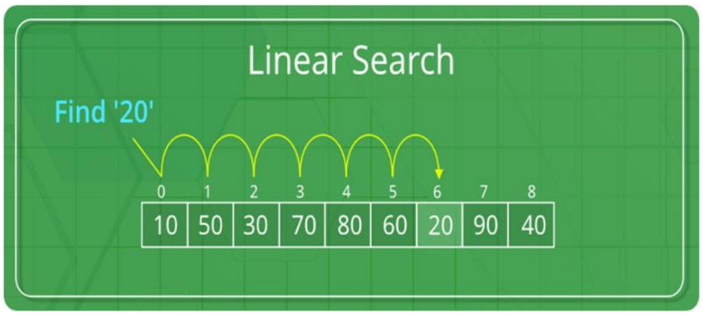
- ✓ The best case in Θ notation is not used.
- ✓ Worst case Θ describes asymptotic bounds for worst case combination of input values.
- ✓ If we simply write Θ, it means same as worst case Θ.

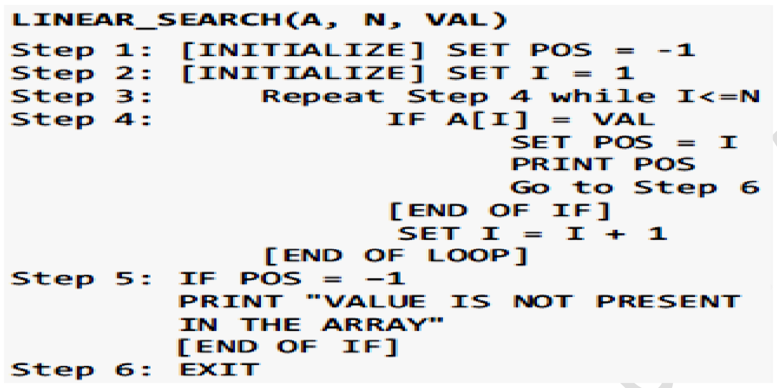
7. Searching:

- ✓ Searching means to *find whether a particular value is present in an array or not*.
- ✓ If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array.
- ✓ However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.
- ✓ There are two popular methods for searching the array elements: *linear search* and *binary search*.

A. Linear search:

- ✓ Linear search, also called as *sequential search,* is a very simple method used for searching an array for a particular value.
- ✓ It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.
- ✓ Linear search is mostly used to search an unordered list of elements.





```
LINEAR_SEARCH(A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:      Repeat Step 4 while I<=N
Step 4:               IF A[I] = VAL
                            SET POS = I
                            PRINT POS
                            Go to Step 6
                      [END OF IF]
                       SET I = I + 1
             [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Complexity of Linear Search Algorithm

- ✓ Linear search executes in O(n) time where n is the number of elements in the array.
- ✓ The best case of linear search is when VAL is equal to the first element of the array.
- ✓ Only one comparison will be made every time.
- ✓ The worst case will happen when either VAL is not present in the array, or it is at the last location of the array.
- ✓ In both the cases, n comparisons will have to be made.
- ✓ However, the performance of the linear search algorithm can be improved by using a sorted array.

B. Binary search:

- ✓ Binary search is a searching algorithm that works efficiently with a sorted list.
- ✓ The mechanism of binary search can be better understood by an analogy of a telephone directory.
- ✓ When we are searching for a particular name in a directory,
- ✓ First open the directory from the middle and then decide whether to look for the name in the first part of the directory, or in the second part of the directory.
- ✓ Again, we open some page in the middle and the whole process is repeated until, we find the right name.

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
         END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:             SET MID = (BEG + END)/2
Step 4:             IF A[MID] = VAL
                         SET POS = MID
                         PRINT POS
                         Go to Step 6
                    ELSE IF A[MID] > VAL
                         SET END = MID - 1
                    ELSE
                         SET BEG = MID + 1
                    [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
             PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Complexity of Binary search Algorithm:

- ✓ The complexity of the binary search algorithm can be expressed as f(n), where n is the number of elements in the array.
- ✓ The complexity of the algorithm is calculated depending on the number of comparisons that are made.
- ✓ In the binary search algorithm, for each comparison, the size of the segment *"where search has to be made is reduced to half"*.
- ✓ The time complexity of the binary search algorithm is O(log n).
- ✓ The best-case time complexity would be O(1) when the central index would directly match the desired value.

8. Sorting:

- ✓ Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending.
- ✓ A sorting algorithm is defined as an algorithm that puts the elements of a list in a certain order, which can be either numerical order, lexicographical order, or any user-defined order.
- ✓ There are two types of sorting:
- ➢ ***Internal sorting*** which deals with sorting the data stored in the computer's memory.
- ➢ ***External sorting*** which deals with sorting the data stored in files. External sorting is applied when there is voluminous data that cannot be stored in the memory.

a. Bubble Sorting:

- ✓ Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment.
- ✓ In *bubble sorting*, consecutive adjacent pairs of elements in the array are compared with each other.
- ✓ If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one.
- ✓ This process will continue till the list of unsorted elements arranged in an order.

BUBBLE_SORT(A, N)

```
Step 1: Repeat Step 2 for I =  0 to N-1
Step 2:      Repeat for J = 0 to N - I
Step 3:           IF A[J] > A[J + 1]
                       SWAP A[J] and A[J+1]
                  [END OF INNER LOOP]
             [END OF OUTER LOOP]
```

Step 4: EXIT

Example:

A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:
(a) Compare 30 and 52. Since 30 < 52, no swapping is done.
(b) Compare 52 and 29. Since 52 > 29, swapping is done.
30, 29, 52, 87, 63, 27, 19, 54
(c) Compare 52 and 87. Since 52 < 87, no swapping is done.
(d) Compare 87 and 63. Since 87 > 63, swapping is done.
30, 29, 52, 63, 87, 27, 19, 54
(e) Compare 87 and 27. Since 87 > 27, swapping is done.
30, 29, 52, 63, 27, 87, 19, 54
(f) Compare 87 and 19. Since 87 > 19, swapping is done.
30, 29, 52, 63, 27, 19, 87, 54
(g) Compare 87 and 54. Since 87 > 54, swapping is done.
30, 29, 52, 63, 27, 19, 54, 87
Pass 2:
(a) Compare 30 and 29. Since 30 > 29, swapping is done.
29, 30, 52, 63, 27, 19, 54, 87
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 63. Since 52 < 63, no swapping is done.
(d) Compare 63 and 27. Since 63 > 27, swapping is done.
29, 30, 52, 27, 63, 19, 54, 87
(e) Compare 63 and 19. Since 63 > 19, swapping is done.
29, 30, 52, 27, 19, 63, 54, 87
(f) Compare 63 and 54. Since 63 > 54, swapping is done.
29, 30, 52, 27, 19, 54, 63, 87
Pass 3:
(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 52. Since 30 < 52, no swapping is done.
(c) Compare 52 and 27. Since 52 > 27, swapping is done.
29, 30, 27, 52, 19, 54, 63, 87
(d) Compare 52 and 19. Since 52 > 19, swapping is done.
29, 30, 27, 19, 52, 54, 63, 87
(e) Compare 52 and 54. Since 52 < 54, no swapping is done.
Pass 4:
(a) Compare 29 and 30. Since 29 < 30, no swapping is done.
(b) Compare 30 and 27. Since 30 > 27, swapping is done.
29, 27, 30, 19, 52, 54, 63, 87
(c) Compare 30 and 19. Since 30 > 19, swapping is done.
29, 27, 19, 30, 52, 54, 63, 87
(d) Compare 30 and 52. Since 30 < 52, no swapping is done.
Pass 5:
(a) Compare 29 and 27. Since 29 > 27, swapping is done.
27, 29, 19, 30, 52, 54, 63, 87
(b) Compare 29 and 19. Since 29 > 19, swapping is done.
27, 19, 29, 30, 52, 54, 63, 87
(c) Compare 29 and 30. Since 29 < 30, no swapping is done.
Pass 6:
(a) Compare 27 and 19. Since 27 > 19, swapping is done.

19, 27, 29, 30, 52, 54, 63, 87

(b) Compare 27 and 29. Since 27 < 29, no swapping is done.

Pass 7:

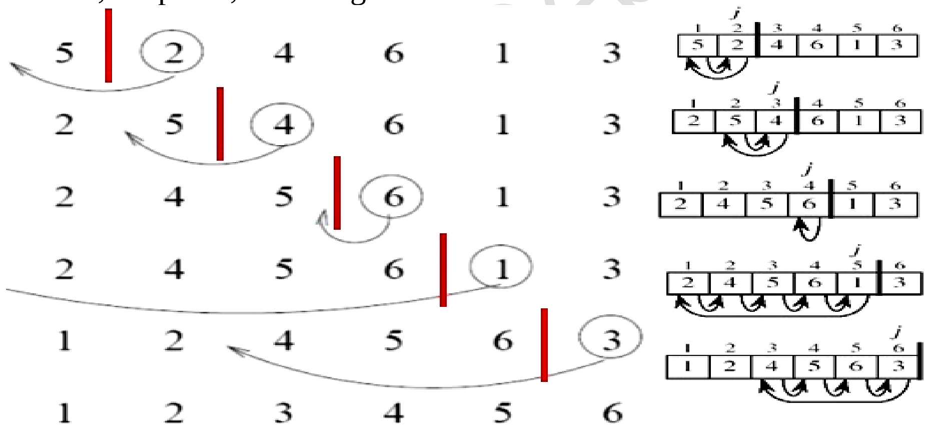
(a) Compare 19 and 27. Since 19 < 27, no swapping is done.

Complexity of Bubble Sort:

- ✓ The complexity of the bubble sort algorithm depends on the number of comparisons.
- ✓ In bubble sort, total number of passes are N-1.
- ✓ In the first pass N-1 comparisons are made to place the highest value in its position.
- ✓ In the second pass N-2 comparisons are made to place the next highest value in its position.
- ✓ $f(n) = (n-1)+ (n-2)+ (n-3)+\ldots\ldots+3+2+1 \quad = n(n-1)/2 \quad =n^2/2 +O(n) \; = O(n^2)$.

b. ***Insertion Sorting:***

- ✓ Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time.
- ✓ The main idea behind insertion sort is that it inserts each item into its proper place in the final list.
- ✓ To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place.
- ✓ Insertion sort is less efficient as compared to other more advanced algorithms such as quick sort, heap sort, and merge sort.



```
INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N - 1
Step 2:      SET TEMP = ARR[K]
Step 3:      SET J = K - 1
Step 4:      Repeat while TEMP <= ARR[J]
                    SET ARR[J + 1] = ARR[J]
                    SET J = J - 1
             [END OF INNER LOOP]
Step 5:      SET ARR[J + 1] = TEMP
        [END OF LOOP]
Step 6: EXIT
```

Advantages of Insertion Sorting:

The advantages of this sorting algorithm are as follows:

- ✓ It is easy to implement and efficient to use on small sets of data.

- ✓ It can be efficiently implemented on data sets that are already substantially sorted.
- ✓ It performs better than algorithms like selection sort and bubble sort.
- ✓ Insertion sort algorithm is simpler than shell sort, with only a small trade-off in efficiency.
- ✓ It is over twice as fast as the bubble sort and almost 40 percent faster than the selection sort.
- ✓ It requires less memory space (O(1) of additional memory space).

c. *Selection Sort:*

- ✓ Selection sort is a sorting algorithm that has a quadratic running time complexity of O(n^2), thereby making it inefficient to be used on large lists.
- ✓ Selection sort performs worse than insertion sort algorithm, it is noted for its simplicity and also has performance advantages over more complicated algorithms in certain situations.
- ✓ Selection sort is generally used for sorting files with very large objects (records) and small keys.

Advantages of Selection Sort

- ➢ It is simple and easy to implement.
- ➢ It can be used for small data sets.
- ➢ It is 60 per cent more efficient than bubble sort.

39	9	81	45	90	27	72	18

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

```
SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N-1
            IF SMALL > ARR[J]
                SET SMALL = ARR[J]
                SET POS = J
            [END OF IF]
        [END OF LOOP]
Step 4: RETURN POS
```

```
SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1
        to N-1
Step 2:     CALL SMALLEST(ARR, K, N, POS)
Step 3:     SWAP A[K] with ARR[POS]
        [END OF LOOP]
Step 4: EXIT
```

Complexity of Selection Sort

- ✓ Selection sort is not difficult to analyze compared to other sorting algorithms.
- ✓ The total number of comparisons are:
  $f(n) = (n-1)+(n-2)+\ldots.+3+2+1$

$=1/2\ n(n-1)\ \ =1/2\ (n^2-n)$.

- ✓ which is of complexity $\mathbf{O(n^2)}$ in terms of number of comparisons.

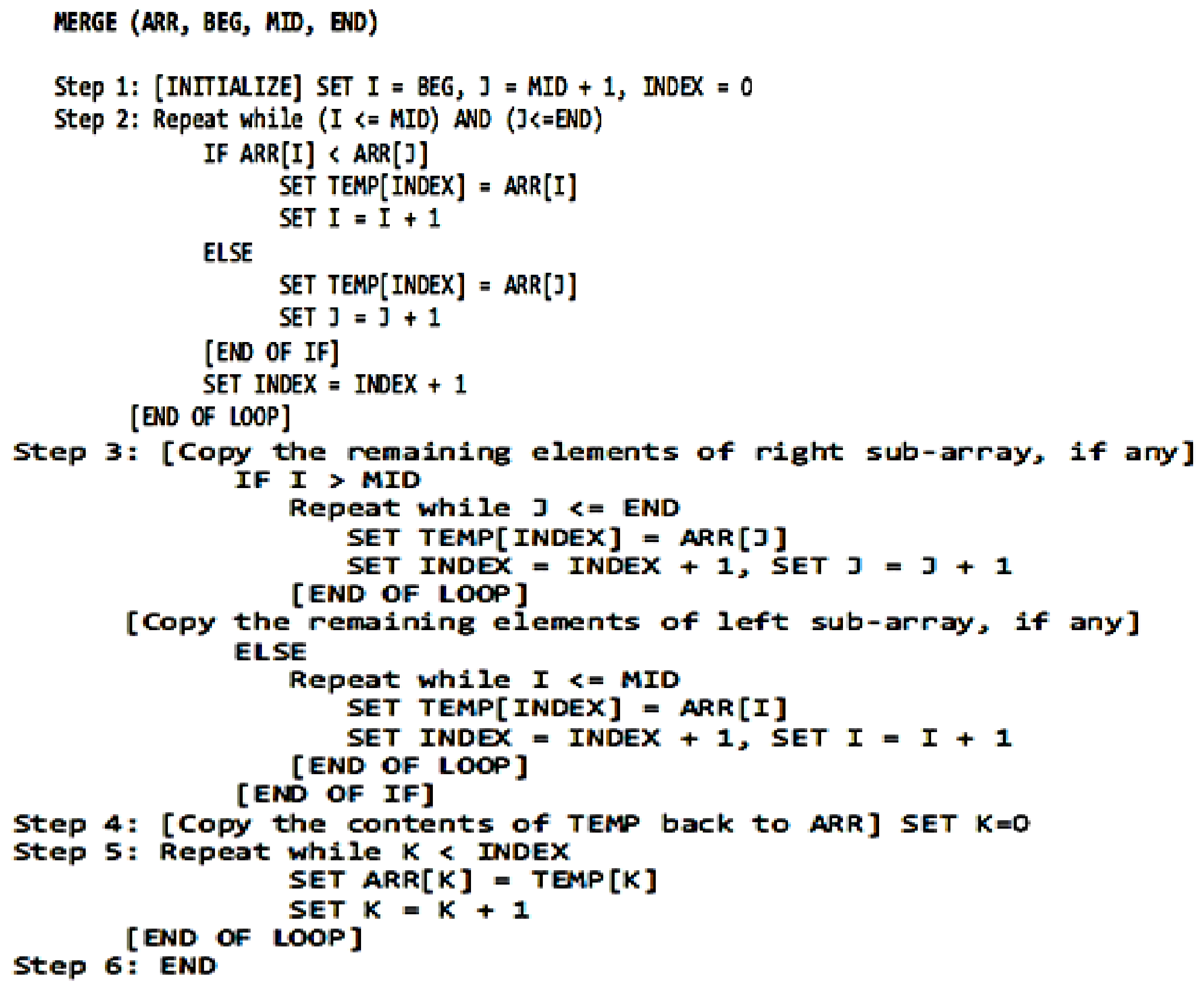
d. Merge Sort:

- ✓ Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm.
- ✓ ***Divide*** means partitioning the n-element array to be sorted into two sub-arrays of n/2 elements. If A is an array containing zero or one element, then it is already sorted. If there are more elements in the array, divide A into two sub-arrays, A1 and A2, each containing about half of the elements of A.
- ✓ ***Conquer*** means sorting the two sub-arrays recursively using merge sort.
- ✓ ***Combine*** means merging the two sorted sub-arrays of size n/2 to produce the sorted array of n elements.

MERGE_SORT(ARR, BEG, END)

```
Step 1: IF BEG < END
            SET MID = (BEG + END)/2
            CALL MERGE_SORT (ARR, BEG, MID)
            CALL MERGE_SORT (ARR, MID + 1, END)
            MERGE (ARR, BEG, MID, END)
    [END OF IF]
Step 2: END
```

Merge Sort Algorithm:



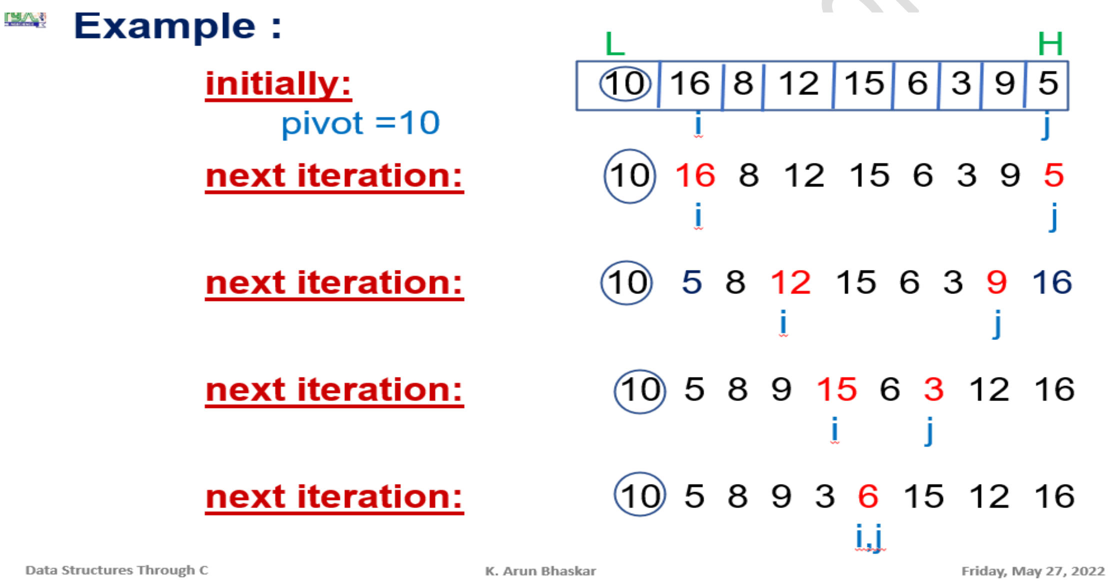
```
 MERGE (ARR, BEG, MID, END)

 Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0
 Step 2: Repeat while (I <= MID) AND (J<=END)
             IF ARR[I] < ARR[J]
                 SET TEMP[INDEX] = ARR[I]
                 SET I = I + 1
             ELSE
                 SET TEMP[INDEX] = ARR[J]
                 SET J = J + 1
             [END OF IF]
             SET INDEX = INDEX + 1
       [END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
             IF I > MID
                 Repeat while J <= END
                     SET TEMP[INDEX] = ARR[J]
                     SET INDEX = INDEX + 1, SET J = J + 1
                 [END OF LOOP]
        [Copy the remaining elements of left sub-array, if any]
             ELSE
                 Repeat while I <= MID
                     SET TEMP[INDEX] = ARR[I]
                     SET INDEX = INDEX + 1, SET I = I + 1
                 [END OF LOOP]
             [END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K=0
Step 5: Repeat while K < INDEX
                 SET ARR[K] = TEMP[K]
                 SET K = K + 1
        [END OF LOOP]
Step 6: END
```

e. Quick Sort

- ✓ Quick sort is a widely used sorting algorithm developed by C. A. R. Hoare that makes O(nlog n) comparisons in the average case to sort an array of n elements.
- ✓ In the worst case, it has a quadratic running time given as $O(n^2)$.
- ✓ The quick sort algorithm is faster than other O(nlog n) algorithms, because its efficient implementation can minimize the probability of requiring quadratic time.
- ✓ Quick sort is also known as partition exchange sort.

The quick sort algorithm works as follows:

- ✓ Select an element pivot from the array elements.
- ✓ Rearrange the elements in the array in such a way that all elements that are less than the pivot appear before the pivot and all elements greater than the pivot element come after it (equal values can go either way).
- ✓ After such a partitioning, the pivot is placed in its final position.
- ✓ It is called the *partition* operation.
- ✓ Recursively sort the two sub-arrays thus obtained. (One with sub-list of values smaller than that of the pivot element and the other having higher value elements.)



Example :

initially: L 10 16 8 12 15 6 3 9 5 H, i j
pivot =10

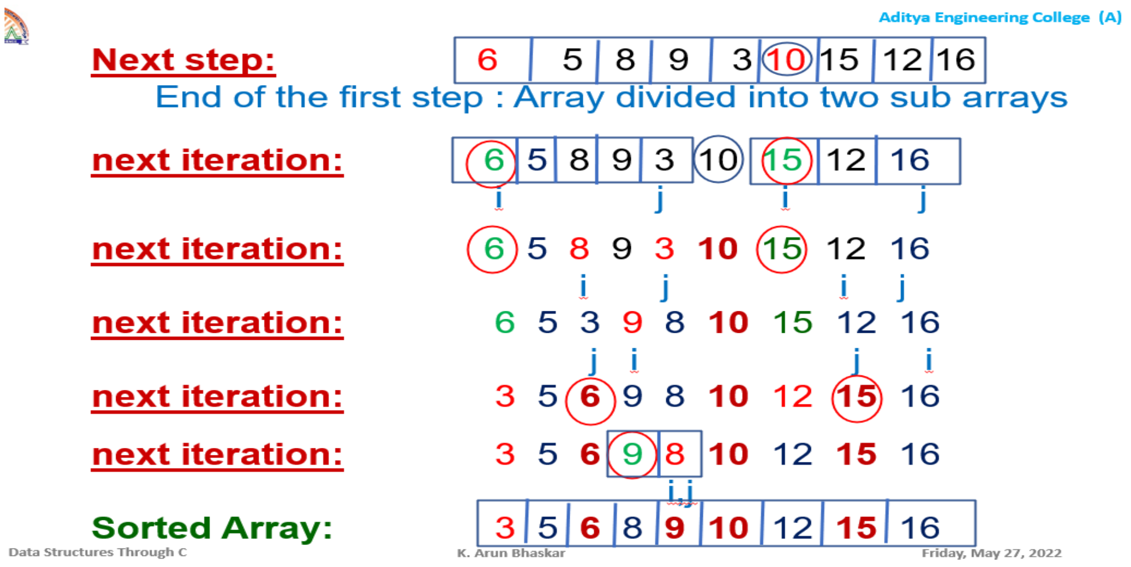
next iteration: 10 16 8 12 15 6 3 9 5, i j

next iteration: 10 5 8 12 15 6 3 9 16, i j

next iteration: 10 5 8 9 15 6 3 12 16, i j

next iteration: 10 5 8 9 3 6 15 12 16, i,j

Data Structures Through C K. Arun Bhaskar Friday, May 27, 2022



Aditya Engineering College (A)

Next step: 6 5 8 9 3 10 15 12 16
End of the first step : Array divided into two sub arrays

next iteration: 6 5 8 9 3 10 15 12 16, i j i j

next iteration: 6 5 8 9 3 10 15 12 16, i j i j

next iteration: 6 5 3 9 8 10 15 12 16, j i j i

next iteration: 3 5 6 9 8 10 12 15 16

next iteration: 3 5 6 9 8 10 12 15 16, i,j

Sorted Array: 3 5 6 8 9 10 12 15 16

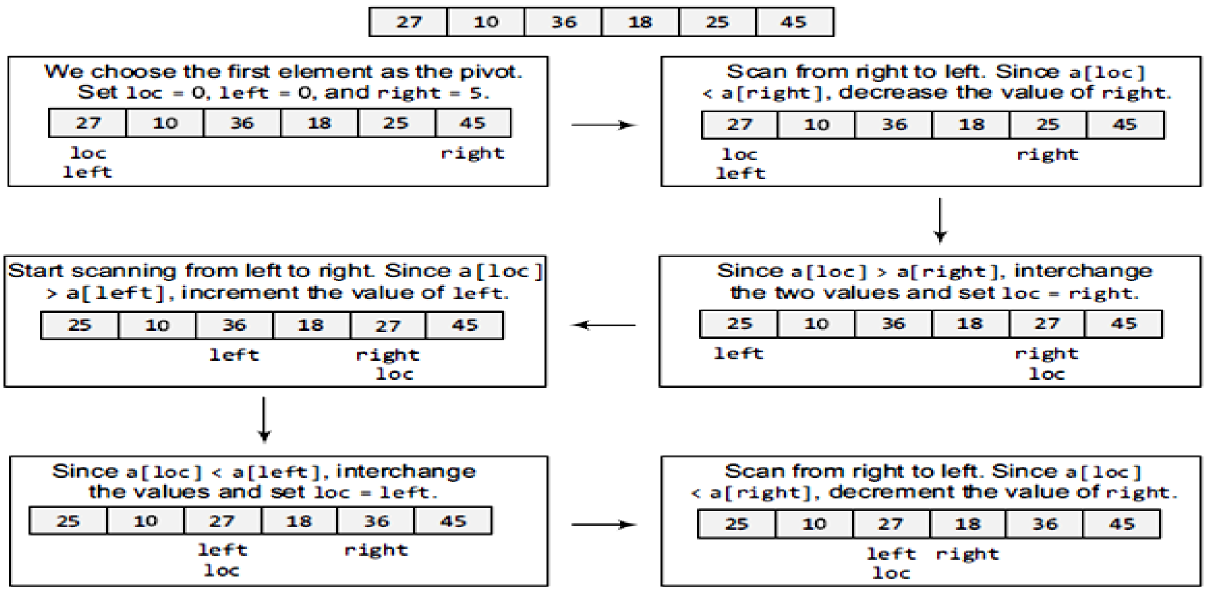
Data Structures Through C K. Arun Bhaskar Friday, May 27, 2022

Quick Sort Algorithm:

```
QUICK_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)
              CALL PARTITION (ARR, BEG, END, LOC)
              CALL QUICKSORT(ARR, BEG, LOC - 1)
              CALL QUICKSORT(ARR, LOC + 1, END)
        [END OF IF]
Step 2: END
```

```
PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG = 0
Step 2: Repeat Steps 3 to 6 while FLAG = 0
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND LOC != RIGHT
                SET RIGHT = RIGHT - 1
        [END OF LOOP]
Step 4: IF LOC = RIGHT
                SET FLAG = 1
        ELSE IF ARR[LOC] > ARR[RIGHT]
                SWAP ARR[LOC] with  ARR[RIGHT]
                SET LOC = RIGHT
        [END OF IF]
Step 5: IF FLAG = 0
                Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
                SET LEFT = LEFT + 1
                [END OF LOOP]
Step 6:         IF LOC = LEFT
                        SET FLAG = 1
                ELSE IF ARR[LOC] < ARR[LEFT]
                        SWAP ARR[LOC] with  ARR[LEFT]
                        SET LOC = LEFT
                [END OF IF]
        [END OF IF]
Step 7: [END OF LOOP]
Step 8: END
```

Example :


We choose the first element as the pivot. Set loc = 0, left = 0, and right = 5.

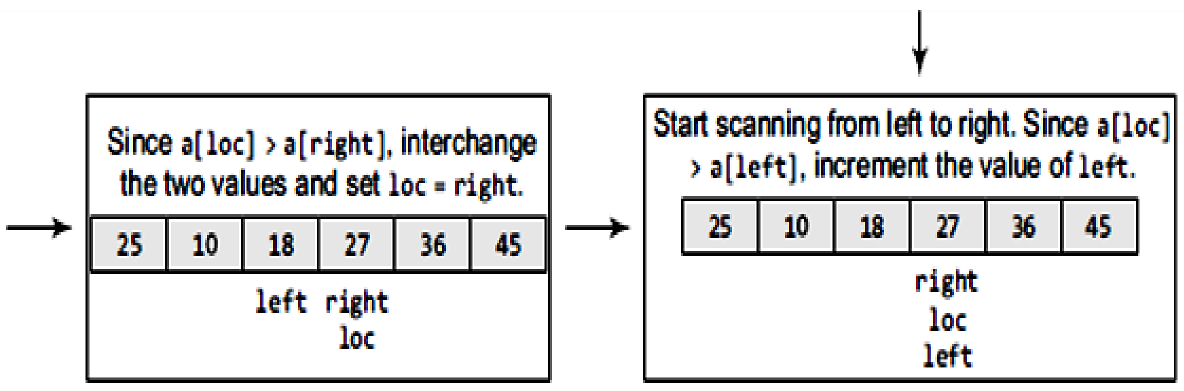
Scan from right to left. Since a[loc] < a[right], decrease the value of right.

Since a[loc] > a[right], interchange the two values and set loc = right.

Start scanning from left to right. Since a[loc] > a[left], increment the value of left.

Since a[loc] < a[left], interchange the values and set loc = left.

Scan from right to left. Since a[loc] < a[right], decrement the value of right.


Since a[loc] > a[right], interchange the two values and set loc = right.
25 10 18 27 36 45
left right
loc
Start scanning from left to right. Since a[loc] > a[left], increment the value of left.
25 10 18 27 36 45
right
loc
left

Now left = loc, so the procedure terminates, as the pivot element (the first element of the array, that is, 27) is placed in its correct position. All the elements smaller than 27 are placed before it and those greater than 27 are placed after it.

The left sub-array containing 25, 10, 18 and the right sub-array containing 36 and 45 are sorted in the same manner.

Complexity of Quick Sort:

- ✓ Quicksort is a divide-and-conquer algorithm.
- ✓ It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
- ✓ For this reason, it is sometimes called partition-exchange sort.
- ✓ The sub-arrays are then sorted recursively. So it require small additional amounts of memory to perform the sorting.
- ✓ Worst-case analysis : O(n^2)
- ✓ Best-case analysis : O(n log n)
- ✓ Average-case analysis : O(n log n)

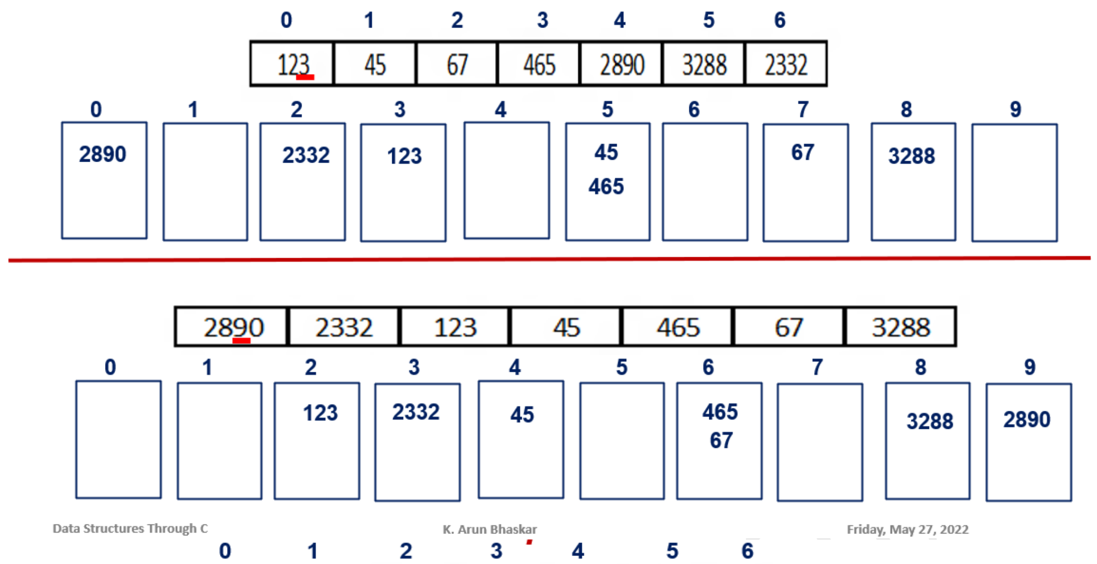
f. Radix Sort:

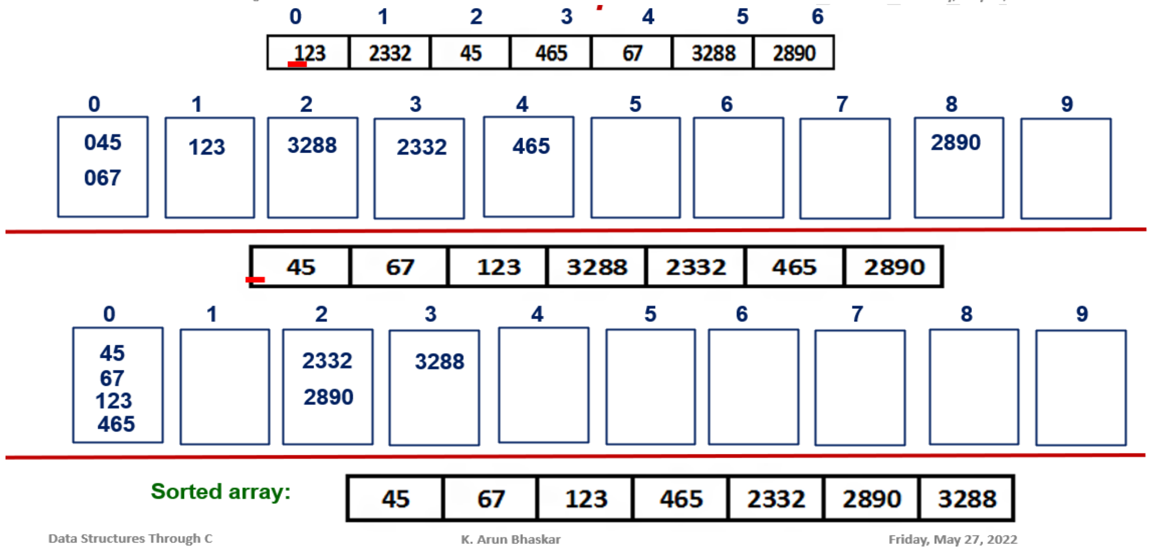
- ✓ Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order.
- ✓ When we have a list of sorted names, the *radix* is 26 (or 26 buckets) because there are 26 letters in the English alphabet.
- ✓ So, radix sort is also known as bucket sort.
- ✓ Observe that words are first sorted according to the first letter of the name.
- ✓ That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on.
- ✓ During the second pass, names are grouped according to the second letter.
- ✓ After the second pass, names are sorted on the first two letters.
- ✓ This process is continued till the nth pass, where n is the length of the name with maximum number of letters.
- ✓ Example:

123	45	67	465	2890	3288	2332

- ✓ Find the largest element from the input array.
- ✓ Calculate the total digits in that largest element.

- ✓ For example : in the above example largest value is 3288. it consist of four digits. Those are : in one’s place 8, ten’s place 8, hundred place 2 and thousand's place 3.
- ✓ Create 0 to 9 buckets .





Radix Sort Algorithm:

```
Algorithm for RadixSort (ARR, N)

Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS = 0
Step 4: Repeat Step 5 while PASS <= NOP-1
Step 5:            SET I = 0 and INITIALIZE buckets
Step 6:            Repeat Steps 7 to 9 while I<N-1
Step 7:                 SET DIGIT  = digit at PASSth place in A[I]
Step 8:                 Add A[I] to the bucket numbered DIGIT
Step 9:                 INCEREMENT bucket count for bucket numbered DIGIT
                   [END OF LOOP]
Step 10:           Collect the numbers in the bucket
         [END OF LOOP]
Step 11: END
```

Complexity of Radix Sort:

- ✓ Radix sort is a non-comparative algorithm, it has advantages over comparative sorting algorithms.
- ✓ Let there be d digits in input integers.
- ✓ Radix Sort takes O(d*(n+b)) time where b is the base for representing numbers, for example, for the decimal system, b is 10.
- ✓ All the three-case analysis : O(n+k) where ‘k’ is maximum possible value. (Best, Average and worst-case analysis)
- ✓ Space Complexity : O(max)