Program : **B.Tech**

Subject Name: **Data Structure**

Subject Code: **CS-303**

Semester: **3rd**

Data Structure Lecture Notes

Unit -5

SORTING:

INTRODUCTION:

Sorting is any process of arranging items systematically, and has two common, yet distinct meanings:

1. **ordering:** arranging items in a sequence ordered by some criterion;
2. **Categorizing**: grouping items with similar properties.

Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted. For example, suppose we have a record of students, every such record will have the following data:

- Roll No.
- Name
- Age
- Class

Here Student roll no. can be taken as key for sorting the records in ascending or descending order. Now suppose we have to search a Student with roll no. 15, we don't need to search the complete record we will simply search between the Students with roll no. 10 to 20.

Need for sorting:

The most common uses of sorted sequences are:

- Making lookup or search efficient.
- Making merging of sequences efficient.
- Enable processing of data in a defined order.

Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios –

- **Telephone Directory –** The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary –** The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

SORT METHODS:

There are many types of sorting techniques, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next sections.

- Bubble sort
- Insertion sort
- Selection sort
- Quick sort
- Merge sort
- Heap sort
- Shell sort
- Radix sort

BUBBLE SORT:

Bubble sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Bubble sort compares all the element one by one and sort them based on their values. If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will swap both the elements, and then move on to compare the second and the third element, and so on.

If we have total n elements, then we need to repeat this process for n-1 times. It is known as bubble sort, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Example:
First Pass:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

QUICK SORT:

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.
Quick sort partitions an array and then calls itself recursively twice to sort the two resulting sub arrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of o(n^2), where n is the number of items.
Quick sort is a divide-and-conquer sorting algorithm in which division is dynamically carried out.

The steps of quick sort are as follows:
Divide: rearrange the elements and split the array into two sub arrays and an element in between such that so that each element in the left subarray is less than or equal the middle element and each element in the right subarray is greater than the middle element.
Conquer: recursively sort the two subarrays.

Algorithm:

```
Quicksort0 (A, p, r)
{
        if (p< r)
        {
                q = Partition(A, p, r);
                Quicksort0 (A, p, q − 1);
                Quicksort0 (A, q + 1, r);
        }
}

Partition(A, p, r)
{
        x = A[r] ;
        i ← p – 1;
        for j ← p to r − 1 do
        {
                if (A[j] ≤ x)
                {
                i ← i + 1 ;
                Exchange {A[i] and A[j] }
                }
        Exchange (A[i + 1] and A[r])
        }
        return i + 1;
}
```

Given a subarray A[p .. r] such that p ≤ r − 1, this subroutine rearranges the input subarray into two subarrays, A[p .. q − 1] and A[q + 1 .. r], so that

- each element in A[p .. q − 1] is less than or equal to A[q] and
- each element in A[q + 1 .. r] is greater than or equal to A[q]

Then the subroutine outputs the value of q. Use the initial value of A[r] as the “pivot,” in the sense that the keys are compared against it. Scan the keys A[p .. r − 1] from left to right and flush to the left all the keys that are greater than or equal to the pivot.

During the for-loop i + 1 is the position at which the next key that is greater than or equal to the pivot should go to.

Example:

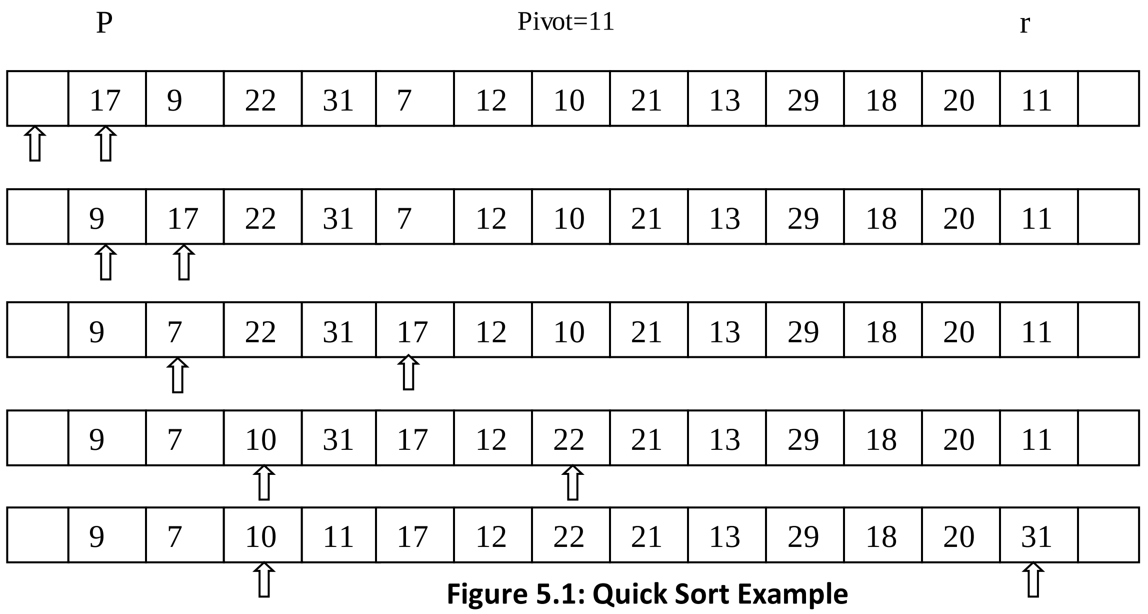

Figure 5.1: Quick Sort Example

SELECTION SORT:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.
This algorithm is not suitable for large data sets as its average and worst case complexities are of O(n^2), where **n** is the number of items.

Example:

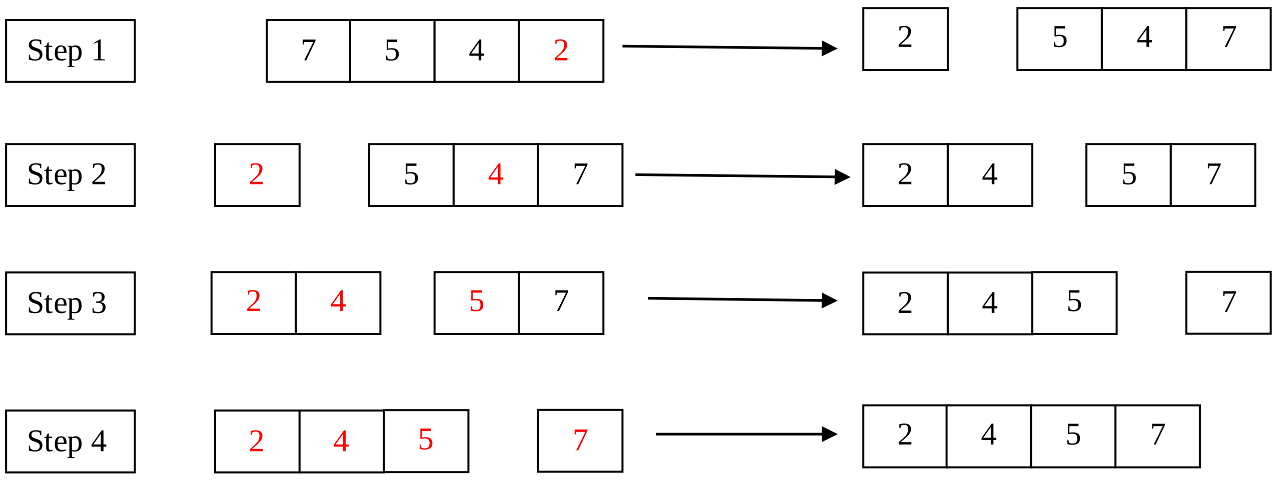

Figure 5.2: Selection Sort

Algorithm:
Step 1 – Set MIN to location 0
Step 2 – Search the minimum element in the list
Step 3 – Swap with value at location MIN
Step 4 – Increment MIN to point to next element
Step 5 – Repeat until list is sorted

HEAP SORT:
Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

Binary Heap:

- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
- A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

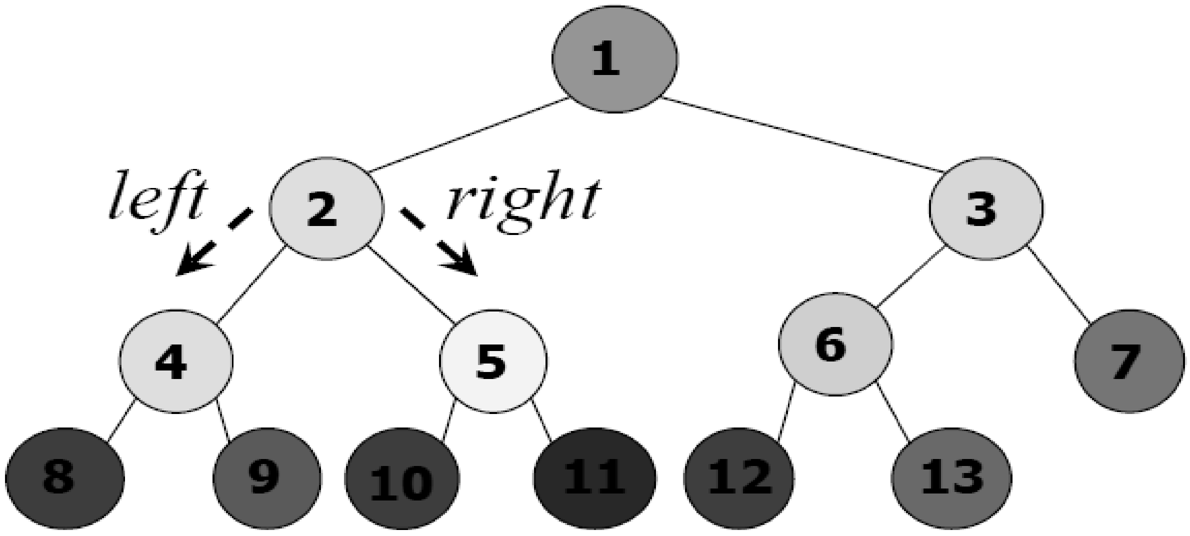

Figure 5.3: Binary Heap

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the
heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

Representation of Binary Heaps:

- An array *A* that represents a heap is an object with two attributes:
- *length*[*A*], which is the number of elements in the array
- *heap-size*[*A*], the number of elements in the heap stored within array *A*.

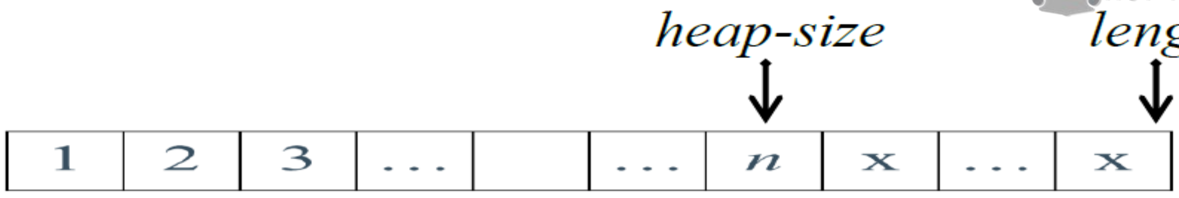

Figure 5.4: Representation of Binary Heap

Properties of Binary Heaps

• If a heap contains *n* elements, its height is $\lg_2 n$.
• In a max-heaps
For every non-root node *i*, *A*[PARENT(*i*)+ ≥ *A*[*i*]
• In a min-heaps
For every non-root node *i*, *A*[PARENT(*i*)+ ≤ *A*[*i*]

Example:

Max Heap

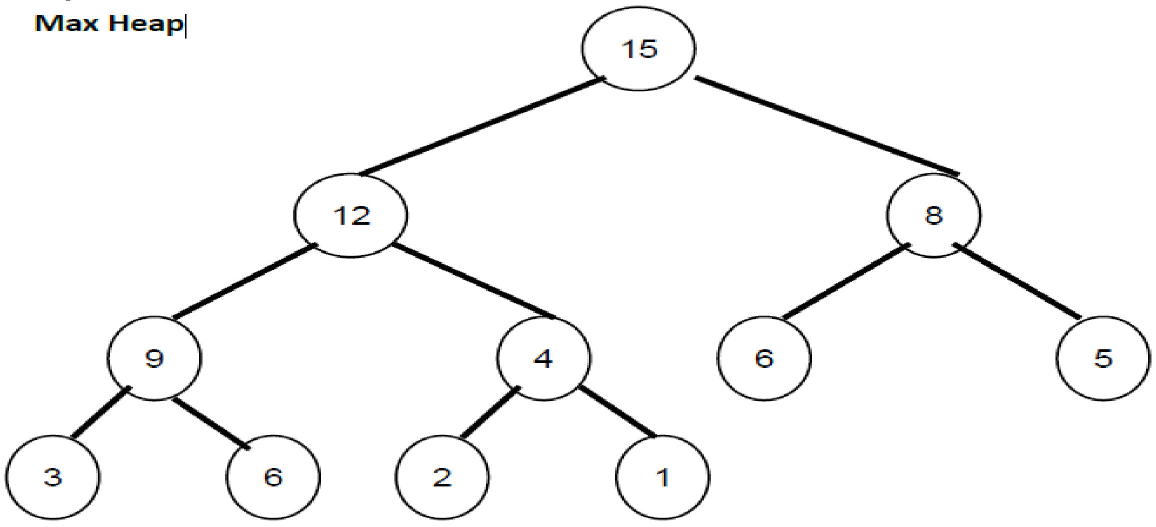

Figure 5.5: Max Heap

Min Heap:

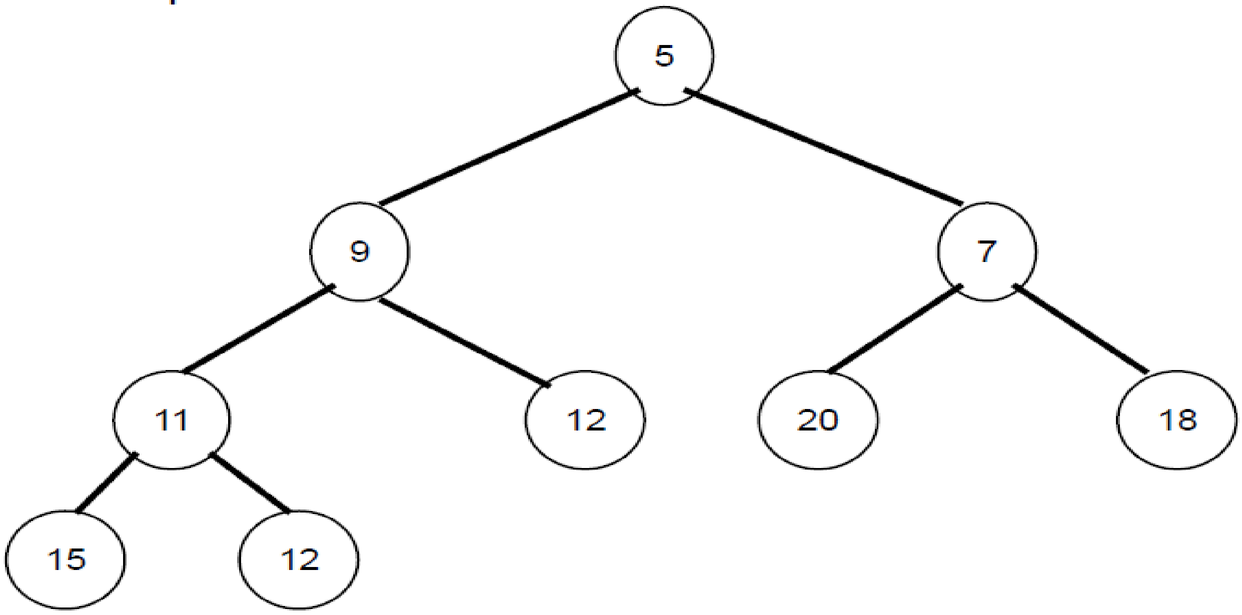

Figure 5.6: Min Heap

INSERTION SORT:

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less. Like Bubble Sorting, insertion sort also requires a single additional memory space.
6. It is a Stable sorting, as it does not change the relative order of elements with equal keys.

How Insertion Sorting Works:

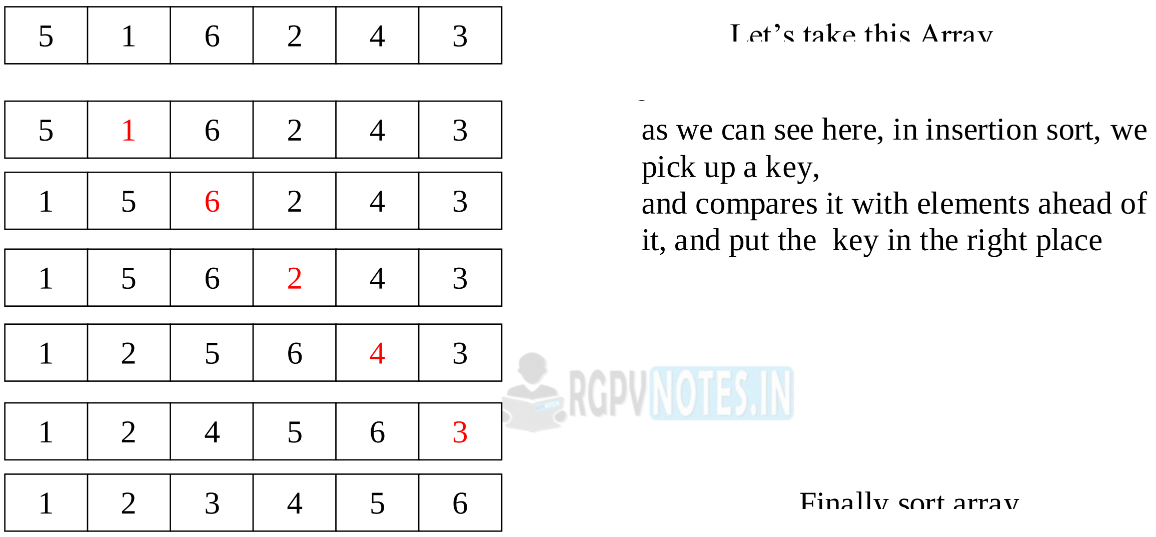
5 1 6 2 4 3
Let's take this Array
5 1 6 2 4 3
as we can see here, in insertion sort, we pick up a key,
and compares it with elements ahead of it, and put the key in the right place
1 5 6 2 4 3
1 5 6 2 4 3
1 2 5 6 4 3
1 2 4 5 6 3
1 2 3 4 5 6
Finally sort array

Figure 5.7: Example of Insertion Sort

SHELL SORT:

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

The shell sort, sometimes called the “diminishing increment sort,” improves on the insertion sort by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort. The unique way that these sublists are chosen is the key to the shell sort.

This algorithm is quite efficient for medium-sized data sets as its average and worst case complexity are of O(n), where n is the number of items.

The unique way that these sublists are chosen is the key to the shell sort. Instead of breaking the list into sublists of contiguous items, the shell sort uses an increment i, sometimes called the gap, to create a sublist by choosing all items that are i items apart.

Example: In the Figure 3.3 This list has nine items. If we use an increment of three, there are three sublists, each of which can be sorted by an insertion sort. After completing these sorts, we get the list shown in Figure 3.4 Although this list is not completely sorted, something very interesting has happened. By sorting the sublists, we have moved the items closer to where they actually belong.

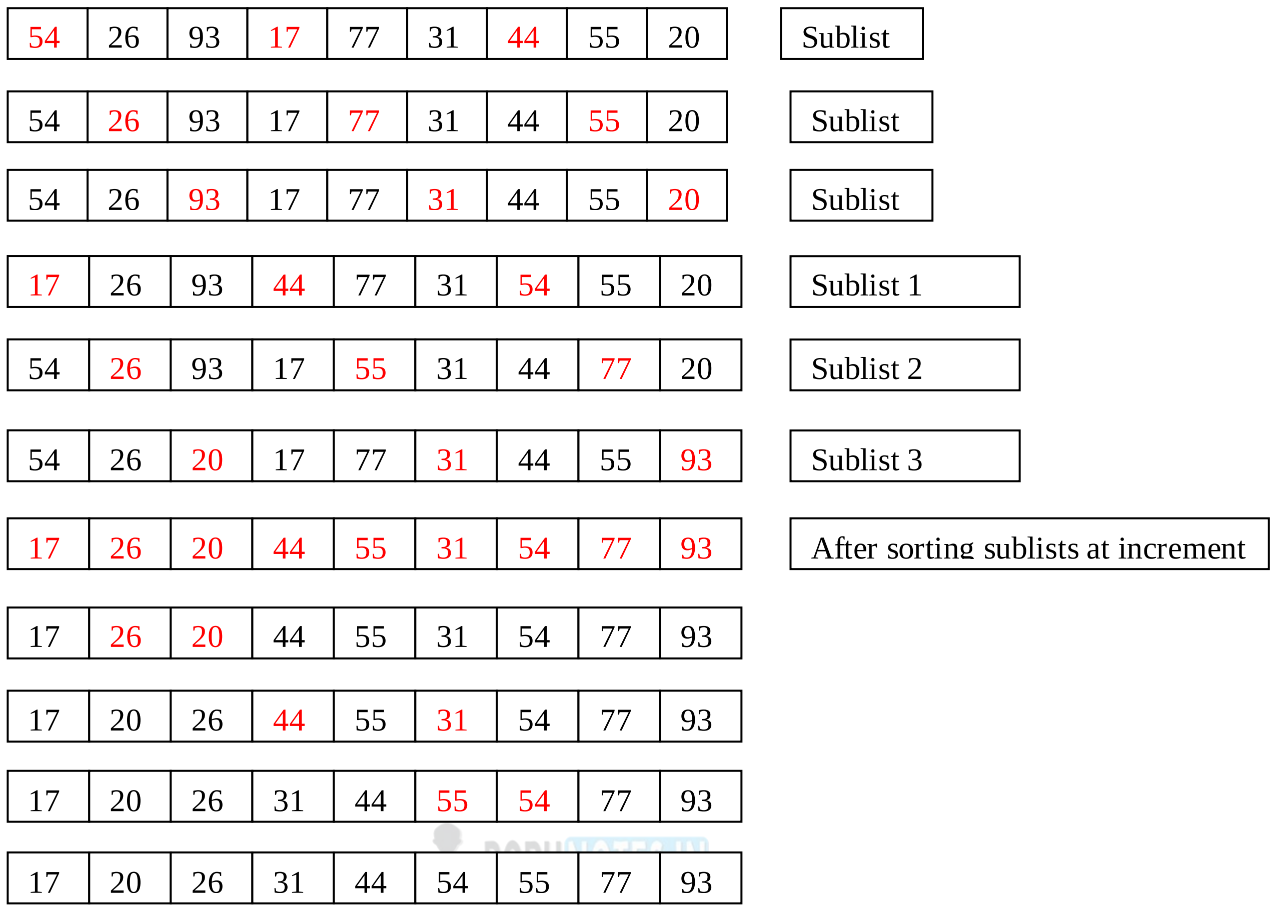

Figure 5.8: Example of Shell Sort

Figure 5.8 shows a final insertion sort using an increment of one; in other words, a standard insertion sort. Note that by performing the earlier sublist sorts, we have now reduced the total number of shifting operations necessary to put the list in its final order. For this case, we need only four more shifts to complete the process.

Here are some key points of shell sort algorithm –

- Shell Sort is a comparison based sorting.
- Time complexity of Shell Sort depends on gap sequence . Its best case time complexity is O(n* logn) and worst case is O(n* log^{2}n). Time complexity of Shell sort is generally assumed to be near to O(n) and less than O(n^2) as determining its time complexity is still an open problem.
- The best case in shell sort is when the array is already sorted. The number of comparisons is less.
- It is an in-place sorting algorithm as it requires no additional scratch space.
- Shell Sort is unstable sort as relative order of elements with equal values may change.
- It is been observed that shell sort is 5 times faster than bubble sort and twice faster than insertion sort its closest competitor.
- There are various increment sequences or gap sequences in shell sort which produce various complexity between O(n) and O(n^2).

MERGE SORT:

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

- Divide the unsorted list into NN sublists, each containing 11 element.
- Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. NN will now convert into N/2N/2 lists of size 2.
- Repeat the process till a single sorted list of obtained.

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following –

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

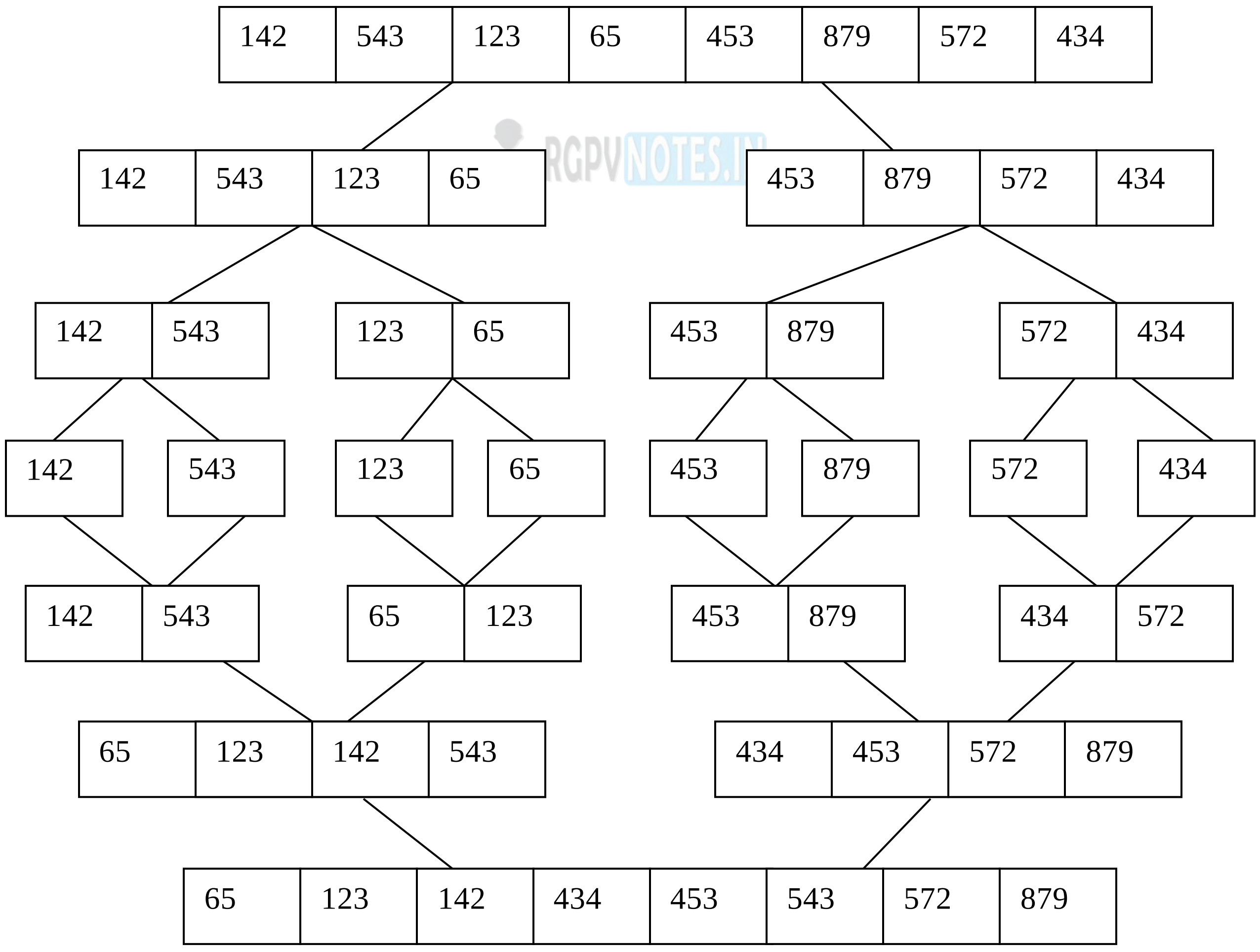

Figure 5.9: Merge Sort

RADIX SORT:

Radix Sort is a clever and intuitive little sorting algorithm. Radix Sort puts the elements in order by comparing the digits of the numbers.

We sort the numbers from the least significant digit to most significant digit.

Consider the following 9 numbers:

493, 812, 715, 710, 195, 437 ,582, 340, 385

We should start sorting by comparing and ordering the **one's** digits:

Digit	Sublist
0	340 710
1	
2	812 582
3	493
4	
5	715 195 385
6	
7	437
8	
9	

Figure 5.10: Radix Sort Example (comparing based on one's digits)

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sublists above. Now, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again:

340 710 812 582 493 715 195 385 437

Note: The **order** in which we divide and reassemble the list is **extremely important**, as this is one of the foundations of this algorithm.

Now, the sublists are created again, this time based on the **ten's** digit:

Digit	Sublist
0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Figure 5.11: Radix Sort Example (comparing based on ten's digits)

Now the sublists are gathered in order from 0 to 9:

710 812 715 437 340 582 385 493 195

Finally, the sublists are created according to the **hundred's** digit:

Digit	Sublist
0	
1	195
2	
3	340 385
4	437 493
5	582
6	
7	710 715
8	812
9	

Figure 5.12: Radix Sort Example(comparing based on hundred's digits)

At last, the list is gathered up again:

195 340 385 437 493 582 710 715 812

And now we have a fully sorted array! Radix Sort is very simple, and a computer can do it fast. When it is programmed properly, Radix Sort is in fact **one of the fastest sorting algorithms** for numbers or strings of letters.

COMPARISON OF VARIOUS SORTING TECHNIQUES

	Time					
Sort	Average	Best	Worst	Space	Stability	Remarks
Bubble sort	O(n^2)	O(n^2)	O(n^2)	Constant	Stable	Always use a modified bubble sort
Modified Bubble sort	O(n^2)	O(n)	O(n^2)	Constant	Stable	Stops after reaching a sorted array
Selection Sort	O(n^2)	O(n^2)	O(n^2)	Constant	Stable	Even a perfectly sorted input requires scanning the entire array
Insertion Sort	O(n^2)	O(n)	O(n^2)	Constant	Stable	In the best case (already sorted), every insert requires constant time
Heap Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Constant	Instable	By using input array as storage for the heap, it is possible to achieve constant space
Merge Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Depends	Stable	On arrays, merge sort requires O(n) space; on linked lists, merge sort requires constant space
Quick Sort	O(n*log(n))	O(n*log(n))	O(n^2)	Constant	Stable	Randomly picking a pivot value (or shuffling the array prior to sorting) can help avoid worst case scenarios such as a perfectly sorted array.

Table 5.1: Sorting Comparison Table

SEARCHING:

Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in the data structure is listed below:

- Linear Search or Sequential Search
- Binary Search

BASIC SEARCH TECHNIQUES:

SEQUENTIAL SEARCH OR LINEAR SEARCH:

This is the simplest method for searching. In this technique of searching, the element to be found in searching the elements to be found is searched sequentially in the list. This method can be performed on a sorted or an unsorted list (usually arrays). In case of a sorted list searching starts from 0th element and continues until the element is found from the list or the element whose value is greater than (assuming the list is sorted in ascending order), the value being searched is reached.

As against this, searching in case of unsorted list also begins from the 0th element and continues until the element or the end of the list is reached.

A simple approach is to do **linear search**, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.

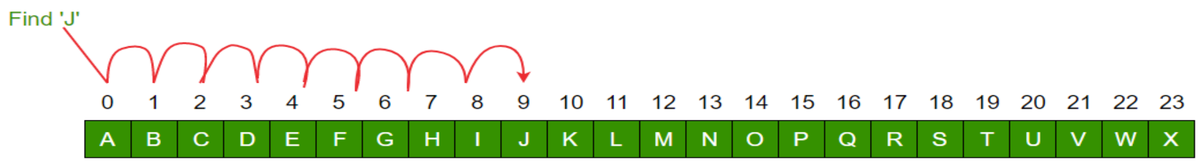

Figure 5.13: Linear Search

BINARY SEARCH:

The Binary search technique is a search technique which is based on Divide & Conquer strategy. The entered array must be sorted for the searching, then we calculate the location of mid element by using formula mid= (Beg + End)/2, here Beg and End represent the initial and last position of array. In this technique we compare the Key element to mid element. So there May be three cases:-

- If array[mid] = = Key (Element found and Location is Mid)
- If array[mid] > Key ,Then set End = mid-1.(continue the process)
- If array [mid] < Key, Then set Beg=Mid+1. (Continue the process)

Binary Search Algorithm

1. [Initialize segment variable] set beg=LB,End=UB and Mid=int(beg+end)/2.
2. Repeat step 3 and 4 while beg<=end and Data[mid] != item.
3. If item< data[mid] then set end=mid-1
1. Else if Item>data[mid] then set beg=mid+1[end of if structure]

4. Set mid= int(beg+end)/2.[End of step 2 loop]
5. If data[mid]=item then set Loc= Mid.
2. Else set loc=null[end of if structure]
6. Exit.

COMPARISON OF SEARCH METHODS:

- The major difference between linear search and binary search is that binary search takes less time to search an element from the sorted list of elements. So it is inferred that efficiency of binary search method is greater than linear search.
- Another difference between the two is that there is a prerequisite for the binary search, i.e., the elements must be sorted while in linear search there is no such prerequisite.

HASHING & INDEXING:

Hashing is the process of indexing and retrieving element (data) in a data structure to provide faster way of finding the element using the hash key. Here, hash key is a value which provides the index value where the actual data is likely to store in the data structure.

In this data structure, we use a concept called Hash table to store data. All the data values are inserted into the hash table based on the hash key value. Hash key value is used to map the data with index in the hash table. And the hash key is generated for every data using a hash function. That means every entry in the hash table is based on the key value generated using a hash function.
Hash table is just an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations can be performed with constant time complexity (i.e. O(1)).

Hash tables are used to perform the operations like insertion, deletion and search very quickly in a data structure. Using hash table concept insertion, deletion and search operations are accomplished in constant time. Generally, every hash table make use of a function, which we'll call the **hash function** to map the data into the hash table.
Hash function is a function which takes a piece of data (i.e. key) as input and outputs an integer (i.e. hash value) which maps the data to a particular index in the hash table.
Basic concept of hashing and hash table is shown in the following figure.

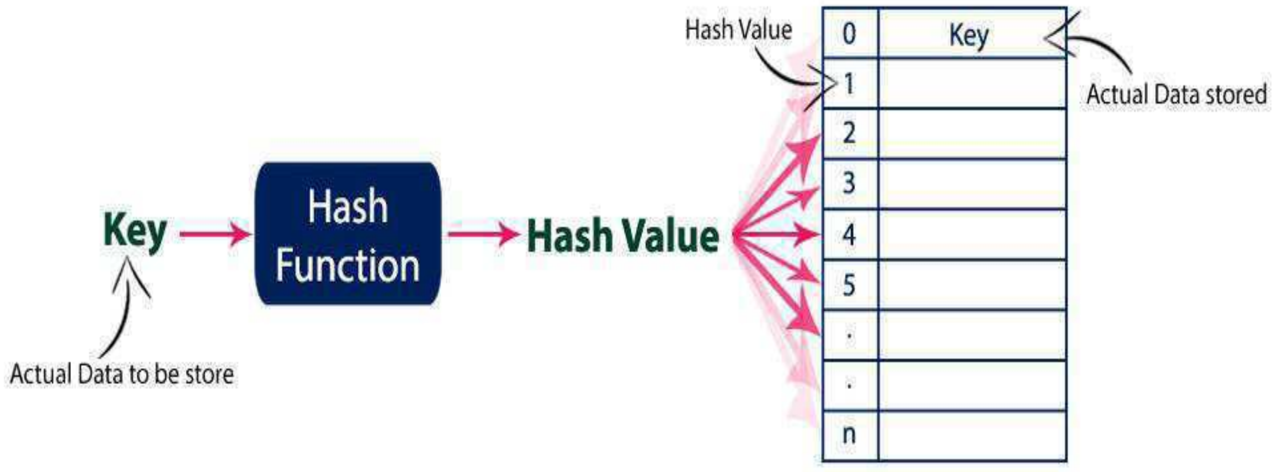

Figure 5.14: Hashing Concept

CASE STUDY:

APPLICATION OF VARIOUS DATA STRUCTURES IN OPERATING SYSTEM:

Some of the application of data structure associating in accomplishing operating System task like:

Stack:

- A stack is useful for the compiler/operating system to store local variables used inside a function block, so that they can be discarded once the control comes out of the function block.
- A stack can be used as an "undo" mechanism in text editors; this operation has been accomplished by keeping all text changes in a stack.
- Stacks are useful in backtracking, which is a process when you need to access the most recent data element in a series of elements.
- An important application of stacks is in parsing. For example, a compiler must parse arithmetic expressions written using infix notation: 1 + ((2 + 3) * 4 + 5)*6

Queue:

- Queues can be used to store the interrupts in the operating system.
- It is used by an application program to store the incoming data.
- Queue is used to process synchronization in Operating System
- Queues are used for CPU job scheduling and in disk scheduling.

Linked lists:

- Linked lists are used in dynamic Memory Management tasks like allocation and releasing memory at run time.
- Linked lists are used in Symbol Tables for balancing parenthesis and in representing Sparse Matrix.
- The cache in your browser that allows you to hit the BACK button where a linked list of URLs can be implemented.

Tree:

- An operating system maintains a disk's file system as a tree, where file folders act as tree nodes. The tree structure is useful because it easily accommodates the creation and deletion of folders and files.
- Trees are used to represent phrase structure of sentences, which is crucial to language processing programs. Java compiler checks the grammatical structures of Java program by reading the program's words and attempting to build the program's parse tree. If successfully constructed the parse tree is used as a guide to help the Java compiler in order to generate the byte code that one finds in program's class file.

Queue:

- The link structure of a website could be represented by a directed graph: the vertices are the web pages available at the website and a directed edge from page A to page B exists if and only if A contains a link to B.
- Simultaneous execution of jobs problem between set of processors and set of jobs can be easily solved with graphs.

APPLICATIONS OF DATA STRUCTURES FOR DATA BASE MANAGEMENT SYSTEM:

The data structures employed in a DBMS context are B-trees, buffer trees, quad trees, R-trees, interval trees, hashing etc. Data Structures for Query Processing high-level input query expressed in a declarative language called SQL the parser scans, parses, and validates the query.