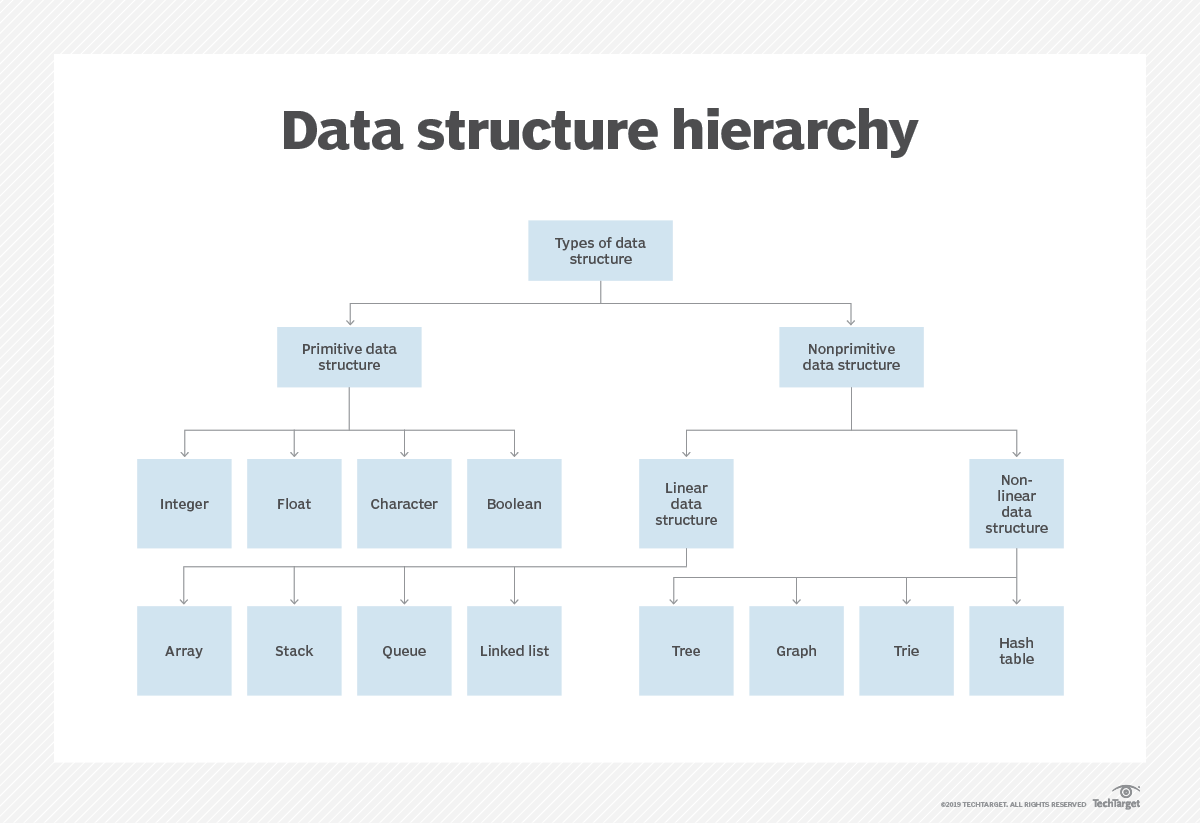
**DATA STRUCTURE**

In computer science, a **data structure** is a data organization, management, and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by a pointer—a bit string, representing a memory address, that can be itself stored in memory and manipulated by the program. Thus, the array and record data structures are based on computing the addresses of data items with arithmetic operations, while the linked data structures are based on storing addresses of data items within the structure itself.

There are number of types in data structure:



**TYPES IN DATA STRUCTURE:**

**Primitive Data Types:** A primitive data type is pre-defined by the programming language. The size and type of variable values are specified, and it has no additional methods.

**Non-Primitive Data Types:** These data types are not actually defined by the programming language but are created by the programmer. They are also called “reference variables” or “object references” since they reference a memory location which stores the data.

**SEARCHING AND SORTING IN DATA STRUCTURE:**

**What is Searching**?

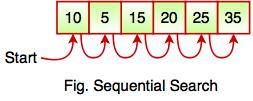
* Searching is the process of finding a given value position in a list of values.
* It decides whether a search key is present in the data or not.
* It is the algorithmic process of finding a particular item in a collection of items.
* It can be done on internal data structure or on external data structure.

Searching Techniques

**To search an element in a given array, it can be done in following ways:**  
  
1. Sequential Search  
2. Binary Search

1. **Sequential Search:**

* Sequential search is also called as Linear Search.
* Sequential search starts at the beginning of the list and checks every element of the list.
* It is a basic and simple search algorithm.
* Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.



**Code for sequential/linear search:**

import java.lang.\*;

import java.util.\*;

public class LinearSearch {

public static void main(String[] args) {

Scanner sc= new Scanner(System.in);

int array[]=new int[10],i,searched\_element;

//Input the element to be searched

System.out.println("Enter the elements to be searched: ");

searched\_element=sc.nextInt();

//Input the array

System.out.println("Enter the elements in the array: ");

for(i=0;i<6;i++)

{

array[i]=sc.nextInt();

}

// Searching for the element in the array

for(i=0;i<6;i++)

{

if(array[i]==searched\_element)

{

System.out.println("Found the element at:"+(i+1)+" position");

break;

}

}

}

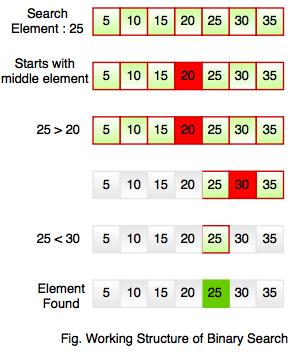
}

**2. Binary Search**

* Binary search works on the principle of divide and conquer.
* Binary Search is used for searching an element in a sorted array.
* It is a fast search algorithm with run-time complexity of O(log n).
* This searching technique looks for a particular element by comparing the middle most element of the collection.
* It is useful when there are large number of elements in an array.

binary array

* The above array is sorted in ascending order. As we know binary search is applied on sorted lists only for fast searching.

**For example,** if searching an element 25 in the 7-element array, following figure shows how binary search works:  
  


**Code for Binary Search:**

**import** java.util.Scanner;

**public** **class** BinarySearch {

**public** **static** **void** main(String[] args) {

Scanner sc= **new** Scanner(System.***in***);

**int** array[]=**new** **int**[10],i,search\_element,mid,start,end;

//Input the element to be searched

System.***out***.println("Enter the elements to be searched: ");

search\_element=sc.nextInt();

//Input the array

System.***out***.println("Enter the elements in the array: ");

**for**(i=0;i<6;i++)

{

array[i]=sc.nextInt();

}

//Binary Search

start=0;

end=array.length-1;

**while**(start<=end)

{

mid=start+(end-1)/2;

//Check if search element is present at middle of the array

**if**(array[mid]==search\_element)

{

System.***out***.println("Element found at:"+(mid+1)+" position");

**break**;

}

//Check if element is greater, then ignore left part of the array

**else** **if**(array[mid]<search\_element)

{

start=mid+1;

**continue**;

}

//Else ignore right part of the array

**else** **if**(array[mid]>search\_element)

{

end=mid-1;

**continue**;

}

//If the element is not present in the array

**else**

{

System.***out***.println("Element is not present in the array");

**break**;

}

}

}

}

**What is Sorting?**

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats

There are various types of sorting techniques:

* Bubble Sort.
* Selection Sort.
* Insertion Sort.
* Quick Sort.
* Merge Sort.
* Heap Sort.

1. **BUBBLE SORT:**

**How Bubble Sort Works?**

We take an unsorted array for our example. Bubble sort takes Ο(n2) time so we're keeping it short and precise.

Bubble Sort

Bubble sort starts with very first two elements, comparing them to check which one is greater.

Bubble Sort

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare with 27.

Bubble Sort

We find that 27 is smaller than 33 and these two values must be swapped.

Bubble Sort

The new array should look like this −

Bubble Sort

Next we compare 33 and 35. We find that both are in already sorted positions.

Bubble Sort

Then we move to the next two values, 35 and 10.

Bubble Sort

We know then that 10 is smaller 35. Hence they are not sorted.

Bubble Sort

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

Bubble Sort

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Bubble Sort

Notice that after each iteration, at least one value moves at the end.

Bubble Sort

And when there's no swap required, bubble sorts learn that an array is completely sorted.

Bubble Sort

Now we should look into some practical aspects of bubble sort.

**CODE FOR BUBBLE SORT:**

**import** java.util.Scanner;

**public** **class** BubbleSort {

**public** **static** **void** main(String[] args) {

Scanner sc= **new** Scanner(System.***in***);

**int** array[]=**new** **int**[10],i,j,temp,elements;

//Input the no.of elements

System.***out***.println("Enter the no.of elements to be added in the array: ");

elements=sc.nextInt();

//Input the array

System.***out***.println("Enter the elements in the array: ");

**for**(i=0;i<elements;i++)

{

array[i]=sc.nextInt();

}

//Bubble Sort

**for**(i=1;i<elements;i++)

{

**for**(j=0;j<(elements-i);j++)

{

**if**(array[j]>array[j+1])

{

temp=array[j];

array[j]=array[j+1];

array[j+1]=temp;

}

}

}

**for**(i=0;i<elements;i++)

{

System.***out***.print(array[i]+" ");

}

}

}

1. **SELECTION SORT:**

**How Selection Sort Works?**

Consider the following depicted array as an example.

Unsorted Array

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

Selection Sort

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

Selection Sort

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

Selection Sort

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

Selection Sort

After two iterations, two least values are positioned at the beginning in a sorted manner.

Selection Sort

The same process is applied to the rest of the items in the array.

**Code for Selection Sort:**

**import** java.util.Scanner;

**public** **class** SelectionSort {

**public** **static** **void** main(String[] args) {

Scanner sc= **new** Scanner(System.***in***);

**int** array[]=**new** **int**[10],i,j,small,temp,position,elements;

//Input the no.of elements

System.***out***.println("Enter the no.of elements to be added in the array: ");

elements=sc.nextInt();

//Input the array

System.***out***.println("Enter the elements in the array: ");

**for**(i=0;i<elements;i++)

{

array[i]=sc.nextInt();

}

//Selection Sort

**for**(i=0;i<(elements-1);i++)

{

small=array[i];position=i;

**for**(j=i+1;j<elements;j++)

{

**if**(small>array[j])

{

small=array[j];

position=j;

}

}

temp=array[position];

array[position]=array[i];

array[i]=temp;

}

**for**(i=0;i<elements;i++)

{

System.***out***.print(array[i]+" ");

}

}

}

1. **INSERTION SORT:**

**How Insertion Sort Works?**

We take an unsorted array for our example.

Unsorted Array

Insertion sort compares the first two elements.

Insertion Sort

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion Sort

Insertion sort moves ahead and compares 33 with 27.

Insertion Sort

And finds that 33 is not in the correct position.

Insertion Sort

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

Insertion Sort

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

Insertion Sort

These values are not in a sorted order.

Insertion Sort

So we swap them.

Insertion Sort

However, swapping makes 27 and 10 unsorted.

Insertion Sort

Hence, we swap them too.

Insertion Sort

Again we find 14 and 10 in an unsorted order.

Insertion Sort

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

Insertion Sort

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

**Algorithm:**

**Step 1** − If it is the first element, it is already sorted. return 1;

**Step 2** − Pick next element

**Step 3** − Compare with all elements in the sorted sub-list

**Step 4** − Shift all the elements in the sorted sub-list that is greater than

the value to be sorted

**Step 5** − Insert the value

**Step 6** − Repeat until list is sorted

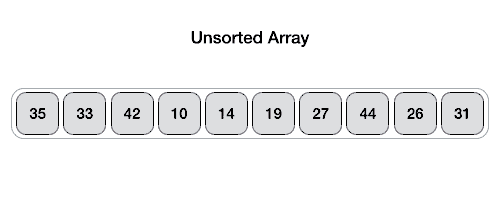
1. **QUICK SORT**

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.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(nLogn) and image.png(n2), respectively.

Partition in Quick Sort

Following animated representation explains how to find the pivot value in an array.



The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

**Quick Sort Pivot Algorithm**

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

**Step 1** − Choose the highest index value has pivot

**Step 2** − Take two variables to point left and right of the list excluding pivot

**Step 3** − left points to the low index

**Step 4** − right points to the high

**Step 5** − while value at left is less than pivot move right

**Step 6** − while value at right is greater than pivot move left

**Step 7** − if both step 5 and step 6 does not match swap left and right

**Step 8** − if left ≥ right, the point where they met is new pivot

1. **MERGE SORT**

## How Merge Sort Works?

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

Unsorted Array

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.

Merge Sort Division

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

Merge Sort Division

We further divide these arrays and we achieve atomic value which can no more be divided.

Merge Sort Division

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

Merge Sort Combine

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

Merge Sort Combine

After the final merging, the list should look like this −

Merge Sort

Now we should learn some programming aspects of merge sorting.

### **Algorithm**

**Step 1** − if it is only one element in the list it is already sorted, return.

**Step 2** − divide the list recursively into two halves until it can no more be divided.

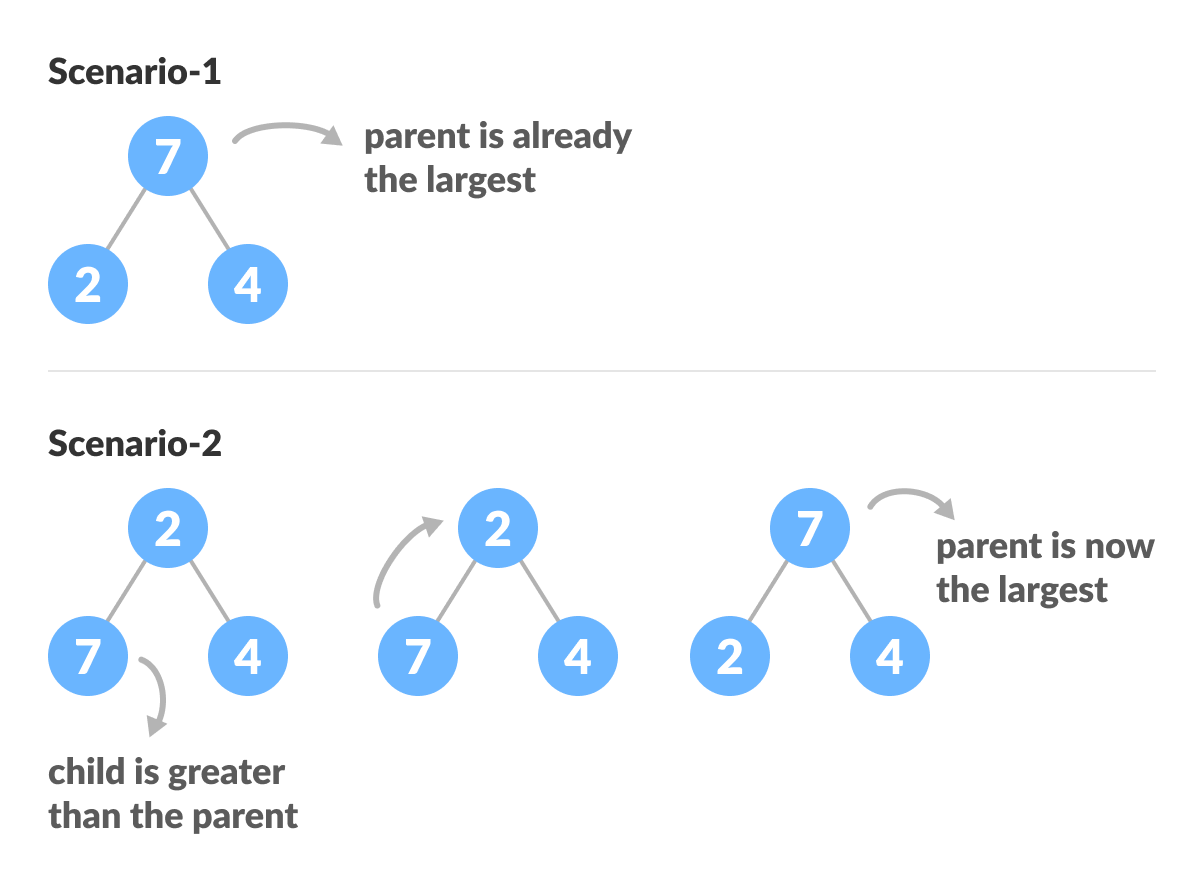
**Step 3** − merge the smaller lists into new list in sorted order.

1. **HEAP SORT**

**How to "heapify" a tree**

Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called heapify on all the non-leaf elements of the heap.

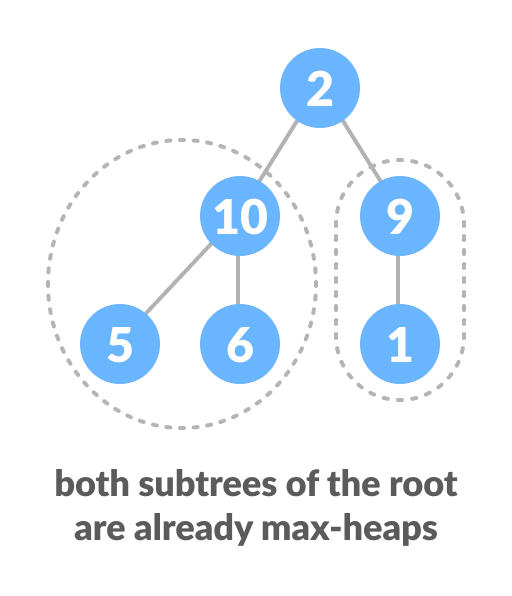
Since heapify uses recursion, it can be difficult to grasp. So let's first think about how you would heapify a tree with just three elements.



The example above shows two scenarios - one in which the root is the largest element and we don't need to do anything. And another in which the root had a larger element as a child and we needed to swap to maintain max-heap property.

If you're worked with recursive algorithms before, you've probably identified that this must be the base case.

Now let's think of another scenario in which there is more than one level.



The top element isn't a max-heap but all the sub-trees are max-heaps.

To maintain the max-heap property for the entire tree, we will have to keep pushing 2 downwards until it reaches its correct position.

