Terminology

Data: It is a collection of facts, such as numbers, **words**, measurements, observations or just descriptions of things.

Data Items: Single unit of values of data.

Entity: An **entity** is an object about which **data** is to be captured. (attributes)

Field: Field is a single elementary unit of information representing an attribute of an entity.

Or

The term "**fields**" refers to columns, or vertical categories of data.

Record: It is collection of field values of a given entity.

Or

The term "**records**" refers to rows, or horizontal groupings of unique **field** data.

File: It is the collection of records of the entities in a given entity set.

Or

A collection of data or information that has a name, called the filename. Almost all information stored in a computer must be in a **file**.

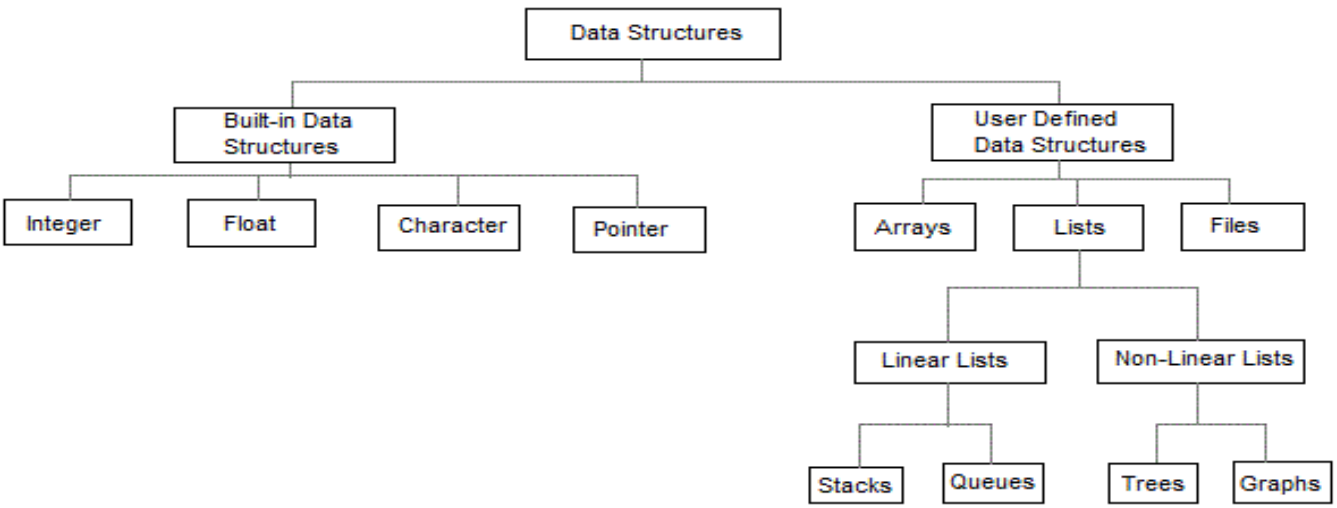
Data-Structure

Definition:

Data can be organized in various ways; Logical or Mathematical arrangement of data is Known as Data-Structure.

Data structure is a **data** organization, management, and storage format that enables efficient access and modification.

Classification of Data-Structure:



Terms:

- **Primitive D.S\Built-in:** The basic **data structures** that directly operate upon the machine instructions and supported by High Level Language (pointer, integer, float, char etc.)
- **Non-Primitive D.S\User Defined:** They are called as derived data type; it means it is derived from fundamental data structures. They are also called "reference variables" or "object references" since they reference a memory location which stores the **data**.

Non-Primitive:

1. **Linear Data-Structure:** Data arrangement in this is in linear sequence.
    - There are two ways to represent data in memory:
        1. Sequent memory allocation
        2. Using pointer or links

    Ex: Array, Queues, Link list, Stacks

2. **Non-Linear Data-Structure:** Data arrangement in this is in non- linear sequence.

    Ex: Tree, Graph

Algorithm

Definition:

Step by step approach/method/procedure to solve a given problem. It describes what the program is going to perform on execution, also it states the order in which these actions have to be perform.

❖ ***Study of Algorithm:***

A. Design of Algorithm
B. Analysis of Algorithm
C. Algorithm Validation
D. Algorithm Testing

A. ***Method of design of Algorithm:***
    1. Divide and Conquer Method
    2. Incremental Programming
    3. Back Tracking
    4. Greedy Method Approach

B. ***Algorithm Validation:*** Checks the algorithm result for all legal set of input, whether it computes the correct and desired results or not.

C. ***Analysis of Algorithm:*** The analysis of algorithm involves the calculation of **Time complexity** and **Space complexity**.

I. ***Time Complexity:*** The amount of time taken by a program to complete its task depends on the number of steps in an algorithm. This time is not always same.

Types:

A. ***Compilation Time:*** The amount of time taken by a compiler to compile an algorithm is known as compilation time. This time is not the execution time it only calculates the declarative statements and checks only for the syntax and semantic errors. The time depends truly upon the architecture of compiler used.

B. ***Run Time:*** The Run time of a program depends upon the size of an algorithm which is only calculated for the execution statements only.

```
int main ()
{
        setbuf (stdout, NULL);
        cout << "First c++ Program." << endl << "The first c plus Experience"<< endl;
        cout << "It is easy then c.\n" << "love to write in it.";
        return 0;
}
```

Runtime: 3 UNIT; 3 Executable sentences

Cases:

A. *Worst Case:* Worst case is the longest time taken by an algorithm for producing a desired result.

B. *Average Case:* The average case is the average time taken by an algorithm for producing a desired output.

C. *Best Case:* The shortest time taken by an algorithm for producing a desired result.

II. Space Complexity: When an algorithm executes it requires some amount of memory to store temporary data known as space complexity.

Amount of Memory = Space occupied by the variables used in the program

A variable amount of memory occupied by the component variable where size is depends on the program being solved.

Types:

A. Instruction Space: When the program compiled that compiled instruction is stored in the memory known as Instruction Space and is independent of the size of the problem.

B. Data Space: The memory used, to hold the variables of an algorithm and other data elements and is related to the size of the problem.

C. Environmental Space: The memory used at the time of execution for each function call which creates a **RUN Time** stack used to hold the returning address of the previously called function.

```
int main ()
{
        int a;
        long b;
        float c;
        double d;
        return 0;
}
```

a – 2

b – 6

c – 4

d – 8

Total Amount of Memory = 2 + 6 + 4 + 8 =20 Unit

- ***Different ways to write an algorithm are:***
  1. English –Special reserved keywords are connected to each other to create an algorithm.
  2. Flowchart: Pictorial representation of the steps of solving problem.
  3. Pseudo-code: Different types of symbols used to create blocks and arithmetic expressions.

Asymptotic Notation

Definition:

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when input tends a Particular/Limiting Value.

Types [5]:

1. Big-O Notation (O-notation)
2. Big-Omega Notation (Ω-notation)
3. Big-Theta Notation (Θ-notation)
4. Little-O Notation (o-notation)
5. Little-Omega Notation (ω-notation)

1. Big-O Notation (O-notation): Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.
   - Let f(n) and g(n) be functions that map positive integers to positive real numbers. We say that f (n) is O(g(n)) (or f (n) ∈ O(g(n))) if there exists a real constant c > 0 and there exists an integer constant n0 ≥ 1 such that f (n) ≤ c ∗ g(n) for every integer n ≥ n0.

---

2. Big-Omega Notation (Ω-notation): Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides best case complexity of an algorithm.
   - Let f (n) and g(n) be functions that map positive integers to positive real numbers. We say that f (n) is Ω(g(n)) (or f (n) ∈ Ω(g(n))) if there exists a real constant c > 0 and there exists an integer constant n0 ≥ 1 such that f (n) ≥ c · g(n) for every integer n ≥ n0.

---

3. Big-Theta Notation (Θ-notation): Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average case complexity of an algorithm.
   - Let f (n) and g(n) be functions that map positive integers to positive real numbers. We say that f (n) is Θ(g(n)) (or f (n) ∈ Θ(g(n))) if and only if f (n) ∈ O(g(n)) and f (n) ∈ Ω(g(n)).

---

4. Little-o Notation (o-notation): Little o notation is used to describe an upper bound that cannot be tight. In other words, loose upper bound of f(n).
   - Let f (n) and g(n) be functions that map positive integers to positive real numbers. We say that f (n) is o(g(n)) (or f (n) ∈ o(g(n))) if for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that f (n) < c ∗ g(n) for every integer n ≥ n0.

5. Little-Omega Notation (ω-notation): Little omega (ω) notation is used to describe a loose lower bound of f (n).
   - Let f (n) and g(n) be functions that map positive integers to positive real numbers. We say that f(n) is ω(g(n)) (or f (n) ∈ ω(g(n))) if for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that f (n) > c · g(n) for every integer n ≥ n0.

Searching

Searching: The process of finding a Particular record in a given set of records is called searching.

Types [2]:

I. Linear
II. Binary

1. **Linear:** A **linear search** or **sequential search** is a method for finding an element within a list. It sequentially checks each element of the list until a match is found or the whole list has been searched.

- **Time complexity = O(n)**
- **Space complexity = O(1)**

```c
#include <stdio.h>

void main( )
{  int num;

    int i,  keynum, found = 0;

    printf("Enter the number of elements ");
    scanf("%d", &num);
    int array[num];
    printf("Enter the elements one by one \n");
    for (i = 0; i < num; i++)
    {
        scanf("%d", &array[i]);
    }

    printf("Enter the element to be searched ");
    scanf("%d", &keynum);
    /*  Linear search begins */
    for (i = 0; i < num ; i++)
    {
        if (keynum == array[i] )
        {
            found = 1;
            break;
        }
    }
    if (found == 1)
        printf("Element is present in the array at position %d",i+1);
    else
        printf("Element is not present in the array\n");
}
```

2. **Binary:** A binary search, also known as half-interval search, logarithmic search, or binary chop, is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

- **Time complexity = O(log** n**)**
- **Space complexity = O(1)**

```c
#include <stdio.h>
int main()
{
    int i, low, high, mid, n, key, array[100];
    printf("Enter number of elementsn");
    scanf("%d",&n);

    printf("Enter %d integersn", n);
    for(i = 0; i < n; i++)
    scanf("%d",&array[i]);

    printf("Enter value to findn");
    scanf("%d", &key);

    low = 0;
    high = n - 1;
    mid = (low+high)/2;
    while (low <= high)
     {
        if(array[mid] < key)
        {
            low = mid + 1;
        }
        else if (array[mid] == key)
        {
            printf("%d found at location %d.n", key, mid+1);
            break;
        }
        else
        {
            high = mid - 1;
            mid = (low + high)/2;
        }
    }

    if(low > high)
    {
        printf("Not found! %d isn't present in the list.n", key);
    }

    return 0;
}
```