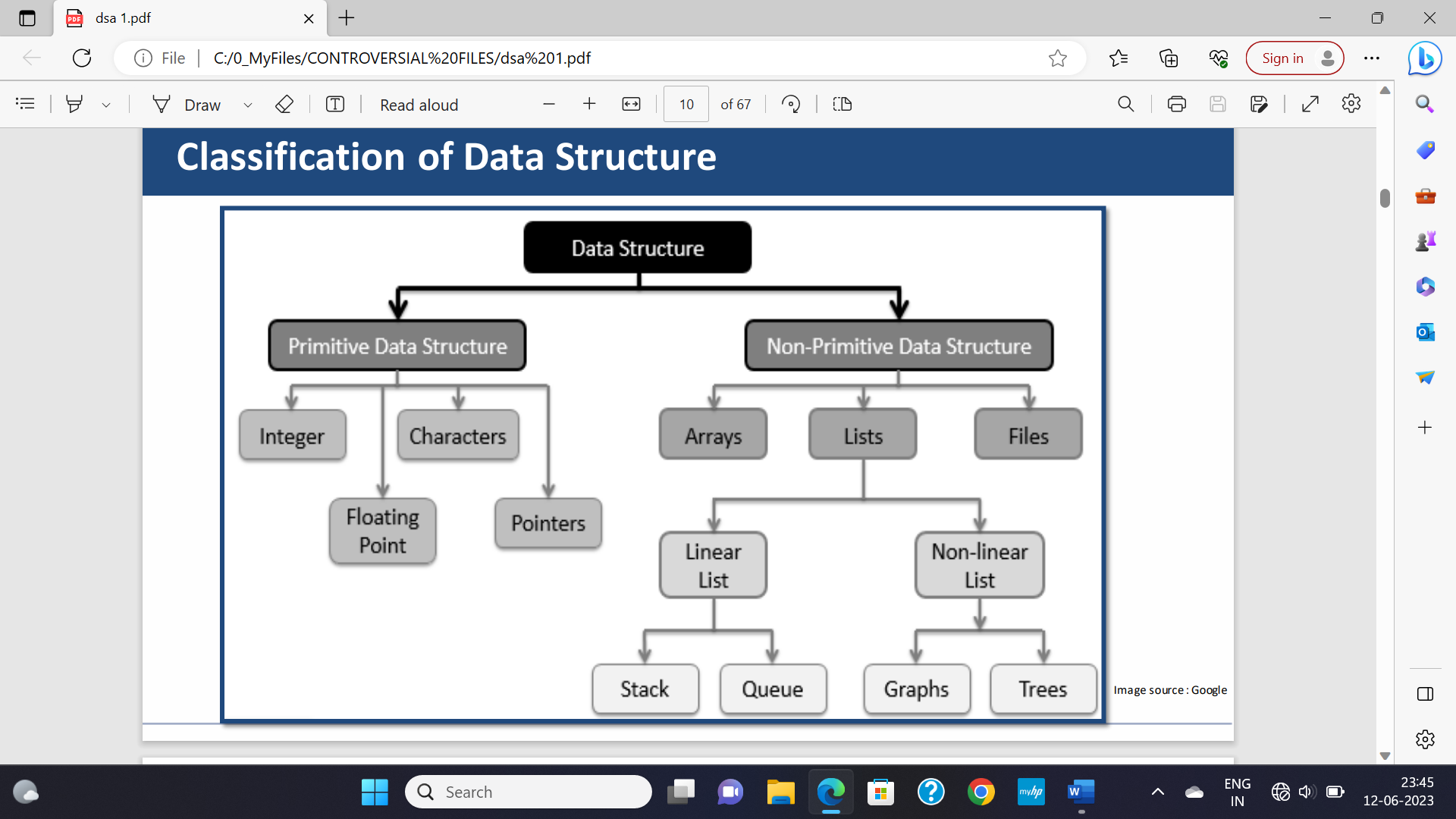
**COMPLEXITIES**

**Introduction**

* **Data Structure:** Systematic way of organizing and accessing of data.

**Classification Of Data Structures**



**Non-Primitive Data Type**

* Non-primitive data type is focused on structuring homogeneous & heterogeneous data.

**Graph:** It is a collection of nodes, and connecting edges between them.

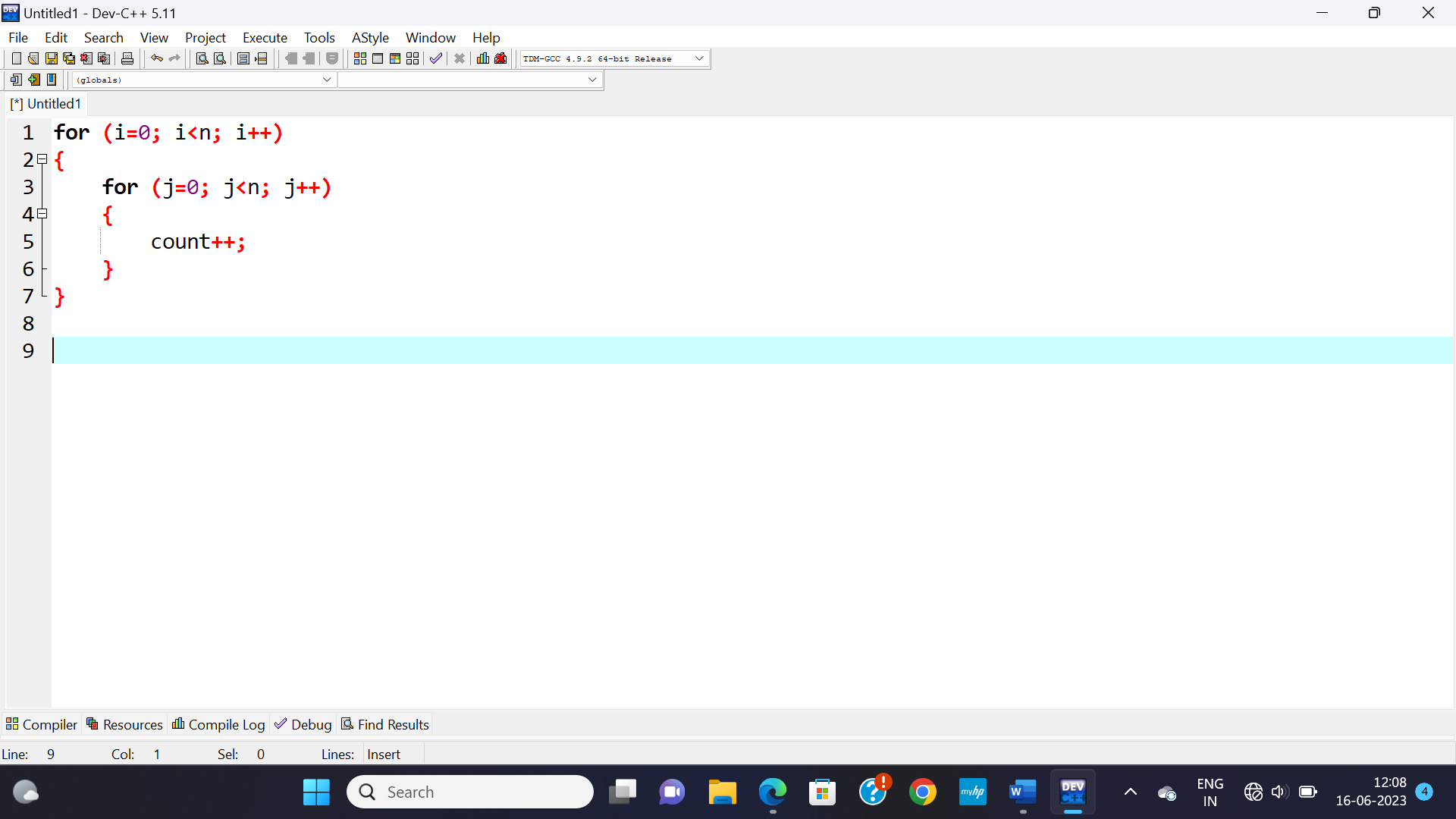
**Types Of Graphs**

\*Same as in discrete mathematics.\*

**Operations On Data Structures**

* There are total 12 types of operation in data structures.

**Example:-**



|  |  |
| --- | --- |
| **Element** | **Number Of Times Used** |
| i=0 | 1 |
| i<n | n+1 |
| i++ | n |
| j=0 | n |
| j<n | n(n+1) |
| j++ | n\*n |
| count++ | n\*n |

Summing up the number of times used gives the **time complexity**:

1+n+1+n+n+n(n+1)+n\*n+n\*n

= 3n2+4n+2

= O(n2)

**Order of Growth of Algorithm**

* It means **how** the computation time increases as the input size increases.
* **Large input** size make significant change to computation time.

**Example:-**

|  |  |  |
| --- | --- | --- |
| **Time Complexity** | **Name** | **Example** |
| 1 | Constant | Adding an element to the font of a linked list. |
| log n | Logarithmic | Finding an element in a sorted array. |
| n | Linear | Finding an element in an unsorted array. |
| n log n | Linear logarithmic | Sorting n items by merge sort. |
| n2 | Quadratic | Shortest path between two nodes in a graph. |
| n3 | Cubic | Matrix multiplication. |
| 2n | Exponential | The towers of Hanoi problem. |

**Searching Problems**

* There are two types of searches:
  + **Linear search**
  + **Binary search**.

**Linear Search Complexity Analysis**

1. **Case time:-**
2. **Worst case:** O(n)
3. **Average case:** O(n)
4. **Best case:** O(1)
5. **Space complexity:** O(1)

**Binary Search Complexity Analysis**

1. **Case time:-**
2. **Worst case:** O(logn)
3. **Average case:** O(log n)
4. **Best case:** O(1)
5. **Space complexity:** O(1)

**Binary Search: Calculating Time Complexity**

Let the length of array to be **“n”**.

* At iteration **1**, length of array = **n**
* At iteration **2**, length of array = **n/2**
* At iteration **3**, length of array = **n/22**
* At iteration **k**, length of array = **n/2k**

**“k”** is the last iteration, after which length of array becomes 1.

Applying **log2**:-

So, n/2k = 1

* n = 2k
* log2 n = log2 2k
* log2 n = k log2 2
* log2 n = k
* k = log2 n