(a) Maps and TreeMap

- **Use Case:**
  - When you need a sorted key-value store.
  - When you frequently need to iterate over keys in sorted order.
  - When performing range-based queries (finding keys in a specific range).
- **Time Complexity:**
  - **Insertion, Deletion, Search:** O(logn) (since it is implemented using Red-Black Tree)
  - **Ordered Traversal:** O(n) (since it maintains order)
- **Example Use Cases:**
  - Implementing an **LRU cache** where ordered keys help in maintaining recent accesses.
  - **Event scheduling** where keys represent timestamps.
  - **Range queries**, such as finding all elements between two dates.

```
#include <iostream>
#include <map>

int main() {
   map<int, string> student;


   student[101] = "Alice";
   student[103] = "Bob";
   student[102] = "Charlie";


   // Iterating over the map (sorted order)
   for (const auto &entry : student) {
      cout << entry.first << " -> " << entry.second << endl;
   }

   return 0;
}
```

Output**:**

```
101 -> Alice
102 -> Charlie
103 -> Bob
```

```
import java.util.Map;
import java.util.TreeMap;

public class TreeMapExample {
   public static void main(String[] args) {
      Map<Integer, String> student = new TreeMap<>();

      student.put(101, "Alice");
      student.put(103, "Bob");
      student.put(102, "Charlie");

      // Iterating in sorted order
      for (Map.Entry<Integer, String> entry : student.entrySet()) {
         System.out.println(entry.getKey() + " -> " + entry.getValue());
      }
   }
}
```

Output (Sorted Order by Key):

```
101 -> Alice
102 -> Charlie
103 -> Bob
```

(b) Unordered Map and HashMap

- **Use Case:**
    - When you need **fast lookups, insertions, and deletions** without worrying about order.
    - When keys do not need to be stored in any particular order.
- **Time Complexity:**
    - **Insertion, Deletion, Search:** O(1) on average (amortized), but **O(n)** in worst case (hash collisions).
- **Example Use Cases:**
    - **Counting word frequencies** in a document.
    - **Caching results** of function calls (memoization).
    - **Graph adjacency list representation** (when order of edges does not matter).

```
#include <iostream>
#include <unordered_map>

int main() {
    unordered_map<int, string> student;


    student[101] = "Alice";
    student[103] = "Bob";
    student[102] = "Charlie";


    // Iterating over the unordered_map (no guaranteed order)
    for (const auto &entry : student) {
        cout << entry.first << " -> " << entry.second << endl;
    }

    return 0;
}
```

Possible Output (Order May Vary):

```
103 -> Bob
101 -> Alice
102 -> Charlie
```

```
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        Map<Integer, String> student = new HashMap<>();

        student.put(101, "Alice");
        student.put(103, "Bob");
        student.put(102, "Charlie");

        // Iterating (unordered)
        for (Map.Entry<Integer, String> entry : student.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
    }
}
```

Output (Unordered Output):

```
103 -> Bob
101 -> Alice
102 -> Charlie
```

(c) Set and TreeSet

- **Use Case:**
  - When you need a **sorted** collection of unique elements.
  - When you frequently need **range queries**.
- **Time Complexity:**
  - **Insertion, Deletion, Search:** O(logn) (Red-Black Tree based)
  - **Ordered Traversal:** O(n)
- **Example Use Cases:**
  - **Keeping track of unique sorted elements**, like user IDs or timestamps.
  - **Finding the next greater or smaller element**.
  - **Storing ranked data** (e.g., leaderboard scores).

```
#include <iostream>
#include <set>

int main() {
   set<int> numbers;

   numbers.insert(40);
   numbers.insert(10);
   numbers.insert(30);
   numbers.insert(20);
   numbers.insert(10); // Duplicate, will be ignored

   // Iterating over the set (sorted order)
   for (int num : numbers) {
      cout << num << " ";
   }

   return 0;
}
```

Output:
10 20 30 40

```
import java.util.TreeSet;

public class TreeSetExample {
   public static void main(String[] args) {
      TreeSet<Integer> numbers = new TreeSet<>();
```

```
        numbers.add(40);
        numbers.add(10);
        numbers.add(30);
        numbers.add(20);
        numbers.add(10); // Duplicate, ignored

        // Iterating in sorted order
        for (int num : numbers) {
            System.out.print(num + " ");
        }
    }
}
```

(d) Unordered Set and HashSet

- **Use Case:**
  - When you only care about uniqueness, **not ordering**.
  - When **fast lookup, insert, and delete** operations are needed.
- **Time Complexity:**
  - **Insertion, Deletion, Search:** O(1) on average (amortized), but **O(n)** in worst case (hash collisions).
- **Example Use Cases:**
  - **Checking for duplicates** in an array.
  - **Storing visited nodes** in a graph traversal.
  - **Membership tests** (checking if an element exists).

```
#include <iostream>
#include <unordered_set>

int main() {
   unordered_set<int> numbers;

   numbers.insert(40);
   numbers.insert(10);
   numbers.insert(30);
   numbers.insert(20);
   numbers.insert(10); // Duplicate, will be ignored

   // Iterating over the unordered_set (no guaranteed order)
   for (int num : numbers) {
      cout << num << " ";
   }

   return 0;
}
```

Possible Output (Order May Vary):
30 40 10 20

```
import java.util.HashSet;

public class HashSetExample {
   public static void main(String[] args) {
```

```java
        HashSet<Integer> numbers = new HashSet<>();

        numbers.add(40);
        numbers.add(10);
        numbers.add(30);
        numbers.add(20);
        numbers.add(10); // Duplicate, ignored

        // Iterating (unordered)
        for (int num : numbers) {
            System.out.print(num + " ");
        }
    }
}
```

Summary Table

Data Structure	Use Case	Time Complexity
TreeMap (RB Tree Map)	Ordered key-value storage, range queries	O(logn) for insert, delete, search
Unordered Map (HashMap)	Fast key-value storage without order	O(1) avg, O(n) worst for insert, delete, search
TreeSet (RB Tree Set)	Unique elements in sorted order, range queries	O(logn) for insert, delete, search
Unordered Set (HashSet)	Unique elements without order, fast lookups	O(1) avg, O(n) worst for insert, delete, search

When to Choose Which?

- **Use a TreeMap / TreeSet** when **ordering** matters.
- **Use an Unordered Map / HashMap** when **speed** is the priority.
- **Use a TreeSet** if you need to **quickly find the next/previous element**.
- **Use an Unordered Set / HashSet** for **fast duplicate checking or membership tests**