# TP SFSD: Report

**Team Members:** Rachid Mustapha Amine - Boucif Soheib
**Section:** C **Group:** 12

## Part I:

In this part, we focus exclusively on the **index structure** and do not directly manipulate the **data file**.
It is assumed that the **data file already exists** and that all records are already stored in it. Therefore, the work is performed **only on the index**, which is generated **randomly** to simulate an existing indexed file.

First, an **index file organized as a TOF structure** is generated using **random values**. The file is divided into several **blocks**, and for each block a random number of **entries** is created. Each entry contains a **key**, its corresponding **block number**, and its **displacement** inside the data file. Before writing each block into the TOF index file, all entries are **sorted in ascending order by key**, ensuring that every block remains **locally ordered**.

After generating the index file, it is **loaded into memory** to construct the index structure. A **primary binary tree (T1)** is built, where each node represents a block of the index file. For every T1 node, the **smallest key (v1)** and the **largest key (v2)** of the block are stored, defining the **key interval** managed by that node. For each T1 node, a **secondary binary search tree (T2)** is constructed from the block entries using a **binary search method**, which guarantees an **ordered** and relatively **balanced structure**.

During a **search operation**, traversal begins at the **T1 tree**. The searched key is compared with the **interval [v1, v2]** of each T1 node. If the key belongs to this interval, the search continues in the corresponding **T2 tree** to locate the exact key and retrieve its **position information**. Otherwise, traversal proceeds to the **left or right subtree** of T1 depending on the key value. This **two-level search strategy** improves search efficiency.

For the **insertion operation**, it is assumed that the **record corresponding to the key has already been inserted into the data file**. As a result, only the **index is updated**. If the key does not already exist in the index, a **new T2 node** is created and inserted in its correct position within the

**T2 tree**. If no suitable T1 node exists to contain this key, a **new T1 node** is created, its **key interval** is initialized, and it is linked into the **T1 tree**.

To ensure the correctness of the **loading**, **saving**, **searching**, and **inserting** operations, we provide the option to **save** and **reload** the file at will. All other options and features are available as well in a simple-to-use menu.

## Part II:

After the user enters random values, these values are first **sorted** before being inserted into the node **P**. Once the insertion is completed, the node **P** contains **four values**, which corresponds to the maximum number of keys allowed in a B-Tree node of order 5. At this stage, the node is considered **full**.

When the user attempts to insert an additional value into node **P**, a **split operation** becomes necessary. To perform this operation, a **helper table** is used. This table temporarily stores the **four existing values** of node **P** along with the **new value** to be inserted, resulting in a total of **five values**. These five values are then **sorted in ascending order**. After sorting:

- The **first two smallest values** are reassigned to the original node **P**.
- The **middle value** is selected as the **median key** and is promoted to the **parent node**.
- The **last two largest values** are assigned to a **new node Q**, which is dynamically created.

This process ensures that the B-Tree properties are preserved, particularly the constraints on the number of keys per node and the ordered structure of the tree.