

# Practical Work Report - File Structures and Data Structures

## Hierarchical Index with T1/T2 Trees

By:

- Azzouz Zaki Group 12 Section C
- Amara Khaled Walid Group 12 Section C

### 1. Introduction

This practical work implements two file indexing structures: a two-level binary tree (T1/T2) and a B-Tree of order 5. The goal is to manage indexed file access with efficient search, insertion, and persistence mechanisms.

### 2. Key Implementation Approach

#### 2.1 Index Loading and Saving

##### Loading Module

Sequentially reads data file blocks and inserts each record into the tree:

```
nBlocks ← getHeader(f, "Nblocks")
while i < nBlocks do
    Read block i+1 into Buf
    for j = 0 to Buf.Nrec - 1 do
        Insert into tree (Root, Buf.Tab[j].Key,
                          Buf.Tab[j].blkAddr,
                          Buf.Tab[j].recAddr)
    end for
end while
```

- Linear file scan → Tree construction
- Maintains block boundaries during load

##### Saving Module

Uses in-order traversal to write sorted records back to file:

##### Main control:

```
ProcessT1(f, Root, Nblocks)
setHeader(f, "Nblocks", Nblocks)
```

##### T1 traversal (in-order, recursively):

```

ProcessT1(f, Node.LC, Nblocks)      // Left subtree
ProcessT2(f, Node.R, Nblocks)        // T2 subtree
ProcessT1(f, Node.RC, Nblocks)       // Right subtree

```

### T2 processing with block buffering:

```

// In-order traversal using stack
while current != NULL do
    Push current onto S
    current ← current.LC
end while

// Process node
Pop S to current
Add record to Buf

// Write when buffer full
if j > MAXTAB then
    Write Buf to block Nblocks
    Nblocks ← Nblocks + 1
    j ← 0
end if

```

### Key techniques:

- **In-order traversal** → sorted output
- **Stack-based iteration** → handles deep trees
- **Block buffering** → efficient disk writes
- **Header update** → stores block count for reloading

## 2.2 Search Algorithm

The search follows the T1 routing logic, then performs BST search in T2:

```

Pointer to t_T1 current ← Root
while current != NULL AND NOT Found do
    if Key < current.V1 then
        current ← current.LC
    else if Key > current.V2 then
        current ← current.RC
    else
        // Search within T2 subtree
        Pointer to t_T2 currentT2 ← current.R
        while currentT2 != NULL AND NOT Found do
            // Binary search in T2

```

## 2.3 Insertion Mechanism

Insertion always occurs in T2 trees, with T1 nodes updated only when:

- A new key extends the current [V1, V2] range
- A new T1 leaf needs to be created

## 2.4 B-Tree Leaf Splitting

When a leaf node overflows (5 keys), it splits:

```
procedure SplitLeafNode(...)  
    Create sorted array with new key  
    middleValue ← tmpArr[2] // Median promoted to parent  
    newLeftNode ← first 2 keys  
    newRightNode ← last 2 keys
```

## 3. Conclusion

The implementation demonstrates:

1. **Efficient range-based indexing** through T1/T2 separation
2. **Persistent storage** with block-aware serialization
3. **Modular design** allowing B-Tree comparison
4. **Practical trade-offs**: T1/T2 simplifies certain range queries while B-Tree optimizes for disk I/O and balance

The pseudo-code provides a clear blueprint for implementing these structures in any programming language, emphasizing algorithmic logic over syntactic details.