University Ferhat Abess-Sétif 1
Faculty of sciences
Computer sciences Departement
Module: ASD3
Level: 2nd-year Computer Science Engineering Students **Dr. Lakehal Abderrahim**

TD/TP 5

Objectives: The goal of this exercise series is to develop a deep, practical understanding of hierarchical data structures, from the simplest tree forms to advanced self-balancing binary search trees.

A. AVL Tree Insertion and Deletion

You are tasked with building an AVL tree to store integer keys. You must perform **insertions** and **deletions** while keeping the tree balanced.

1. Start with an empty AVL tree.
2. Perform the following **insertions** in order: [1,3,4,8,9,2,0,5,10]

   After each insertion:

   - Draw the tree.
   - Calculate the **height** of each node.
   - Calculate the **balance factor** (height(left) - height(right)) for each node.
   - Apply any **rotations** necessary to maintain AVL balance.
3. Then perform the following **deletions** in order: [0, 2]

   After each deletion (check the path from the deleted node to root for imbalances):

   - Draw the tree.
   - Update heights and balance factors.
   - Apply rotations if needed.

Notes :

- Height of a leaf node = 0, height of null = -1.
- Balance factor = height(left) - height(right).

B. Red-Black Tree Insertion and Deletion

Problem Statement:

You are tasked with building a Red-Black Tree to manage **product IDs** in an inventory system. Each product ID is an integer. Your tree must maintain the following Red-Black properties:

1. Every node is either **red** or **black**.
2. The root is always **black**.
3. All leaves (NILs) are black.
4. Red nodes cannot have red children (no two reds in a row).
5. Every path from a node to its descendant NIL nodes contains the **same number of black nodes**.

You are given the sequence of product IDs to insert into the tree: [10, 20, 30, 15, 25, 5, 1]

Task:

1. Insert each key **in the given order**.

   After each insertion:

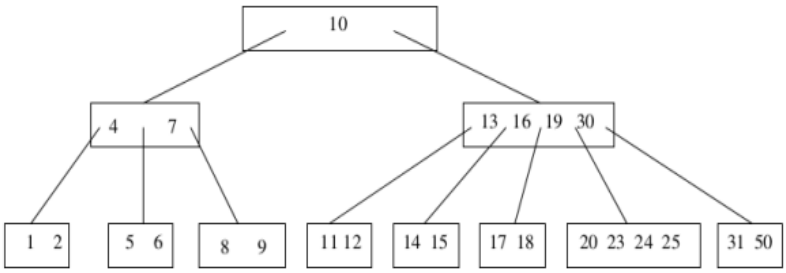
   - Draw the tree.
   - Mark the color of each node (Red or Black).
   - Identify any **violations** of Red-Black properties.

- Apply necessary **rotations** (left or right) and recoloring.

2. After insertion, delete the value 10, then 5.

C. B-Tree Insertion, Search and Deletion

You are given the B-Tree of **order 5** shown below (a B-Tree of order 5 can contain up to 4 keys per node and up to 5 children).



Task:

1. Insertion

   Insert the following keys into the B-Tree **in the given order**: **[21, 32, 62, 49]**

   After each insertion, maintain all B-Tree properties (node capacity, splitting, key ordering).

2. Search Path

   Using **in-order traversal**, list the sequence of **visited nodes** when searching for the key: **62 (**Clearly indicate the keys in each visited node during the search)

3. Deletion

   Delete the key 14 from the B-Tree: After the deletion restore and rearrange B-Tree properties.

D. File System Simulation Using an N-Ary Tree

A file system (like Windows, Linux, macOS) is naturally structured as an **N-ary tree**, where:

- Each **folder** may contain **0 or more files or subfolders**.
- Each **file** is a leaf node.
- Each node stores metadata (attributes).

Node Data Structure : Each node represents either a **file or a folder**.

```
struct Node {
    string name;            // file/folder name
    string type;            // "file" or "folder"
    int size;               // size in KB (files have size, folders size = 0)
    vector<Node*> children; // subfiles/subfolders (only for folders)
};
```

Example:

```
Node* root = new Node{
    "/",          // or "root"
    "folder",
    0,            // size is irrelevant for folders, computed dynamically
    {}            // empty children at the beginning
};
```

You will implement a simplified version of a file system using **C++ and tree structures**.

Implement an N-ary tree to simulate a file system and write functions that support typical operations:

I. Part 1 — Core Functionalities

1. Add a file/folder under a given folder

   Create a function:

   ```
   void add(Node* parent, string name, string type, int size = 0);
   ```

   Requirements

   - You can only add children under nodes of type **folder**.
   - Adding under a file must produce an error.
   - File size must be positive.
   - Folder size = 0 (folders don't directly store size).

   Example

   ```
   Root/
       Documents/
           resume.pdf (120 KB)
           photo.jpg (230 KB)
       Music/
           song.mp3 (4000 KB)
   ```

2. Compute the total size of a folder (DFS)

   Total size = sum of sizes of all descendants.

   Function Prototype

   ```
   int computeFolderSize(Node* folder);
   ```

   Algorithm (DFS)

   1. Start at folder node
   2. For each child:
      - If file → add its size
      - If folder → recursive DFS
   3. Return accumulated size

3. List all files using BFS (breadth-first search)

   ```
   void listFilesBFS(Node* root);
   ```

   Use BFS algorithm to print each folder with all its files, level by level.

   Example

   ```
   Folder: /
       Files: ...
   Folder: Documents
       Files: ...
   Folder: Images
       Files: ...
   ...
   ```

4. Find the deepest file path

   A path is something like:

   ```
   Root/Documents/Projects/Work/AI/report.txt
   ```

   Function Prototype

   ```
   void dfsDeepestFile(Node* root, vector<string>& currentPath, int currentDepth,
                       int& maxDepth, vector<string>& bestPath);
   ```

   Use DFS algorithm:

- Traverse the entire tree
- Maintain:
    - `currentDepth`
    - `maxDepth`
    - `currentPath`
    - `bestPath`
- Update `bestPath` when a leaf file at deeper level is found

II. Part 2 — Interaction Requirements

The program must:

- Start with a root folder: "Root" or "/".
- Provide a menu-like interface to test functions:
    1. Add file/folder
    2. Compute folder size
    3. List files (BFS)
    4. Show deepest file path
    5. Exit

III. Part 3 — Edge Cases to Handle

- Adding a file under a non-existing folder
- Adding a folder under a file (must be rejected)
- Two subitems with the same name under one folder
- Folder with no children (size = 0)
- Multiple deepest paths (choose any or take lexicographically smallest)

IV. Part 4 — Optional extensions (For stronger students)

Feature 1 — Remove a file/folder recursively

Deleting a folder must delete the entire subtree.

Feature 2 — Move an item from one folder to another

Simulates drag and drop.

Feature 3 — Search by name

Return full paths of all matching files.

Feature 4 — Pretty print the file system

Like the Linux tree command.

```
root
 ├── Documents
 │     ├── CV.pdf
 │     ├── Research.docx
 │     └── Archives
 │          └── 2023
 │               └── FinalReport.pdf
 └── Images
       ├── photo1.jpg
       └── photo2.png
```