CS 2413 – Data Structures – Fall 2024 – Project Four
Due 11:59 PM, November 15, 2024

Objective: This project is not just an academic exercise, but a significant step in your understanding of data structures. It aims to implement a multiway search tree (M-Tree), a data structure designed to store and efficiently manage sorted data while maintaining balanced tree properties. The M-Tree provides efficient insertion, deletion, and search operations by splitting a sorted array recursively and using the split values for tree navigation. The M-Tree uses recursive splitting of arrays and places the split values in internal (non-leaf) nodes, while the leaf nodes store actual values in sorted order. Additionally, you will implement a **rebuild operation** to maintain balance in the tree after deletions. This project is a key part of your learning journey in data structures and will equip you with practical skills for your future career.

Search trees, particularly the M-Tree, are not just theoretical constructs. They enable efficient data retrieval and modification in many real-world applications, such as database indexing, file systems, and memory management. The M-Tree enhances traditional binary search trees by allowing each node to hold multiple values and up to M child pointers, resulting in more compact trees with fewer levels. This project will give you a hands-on understanding of how these structures are used in practical scenarios.

Through this project, students will gain practical experience in:

- Implementing and working with multiway search trees.
- Understanding how recursive node splitting is used to balance a tree dynamically.
- Managing a sorted array within tree nodes and determining split points for navigation.
- Exploring dynamic tree balancing techniques through a rebuild operation that reconstructs the tree when necessary.
- Gaining proficiency in memory management and pointer-based tree structures in C++.

The M-tree is structured around splitting values into M parts. The values used for splitting are not stored in the nodes themselves but are instead used to navigate to the correct child nodes. The left parts of the array are stored in the corresponding child nodes, and the tree expands dynamically as new values are inserted.

This project will give students hands-on experience with:

- Templated data structures in C++.
- Recursive algorithms for insertion and deletion in tree structures.
- Implementing a rebuild operation to maintain tree balance and efficiency.
- Optimizing search operations within the context of multiway trees.

By designing and implementing the M-Tree, students will explore how tree-based data structures can efficiently manage and search large datasets and how maintaining balance in a search tree directly impacts performance. This project reinforces core data structure concepts and helps understand how these structures are applied in systems like databases and memory hierarchies.

M-Tree Structure:

(a) Node Properties:

- Internal (Non-Leaf) Nodes:
    - Each non-leaf node stores M-1 values in a sorted array. These values are the split points used to guide navigation between subtrees.
    - Each non-leaf node has M child pointers. These pointers refer to subtrees, and the values between the pointers are used to determine the correct subtree for traversal.
    - The values in non-leaf nodes are used for navigation only. They are not part of the final data stored in the tree.
- Leaf Nodes:
    - Leaf nodes store the actual data values in sorted order.
    - During the tree's recursive construction, when a subarray contains fewer than M values, the recursive splitting stops, and the values are placed into a leaf node.
    - Leaf nodes contain no child pointers, and the values within are used for final retrieval and operations (e.g., searching).

(b) Insertion:

- During insertion, the new value is added to the appropriate leaf node.
- If the leaf node exceeds M values, the node performs splitting.

(c) Search:

- The search operation traverses the internal nodes using the split values to determine the correct child pointer.
- Once a leaf node is reached, the search operation checks for the value within the sorted values of the leaf node.

(d) Deletion:

- Deleting a value from a leaf node involves removing the value from the sorted array in the leaf.

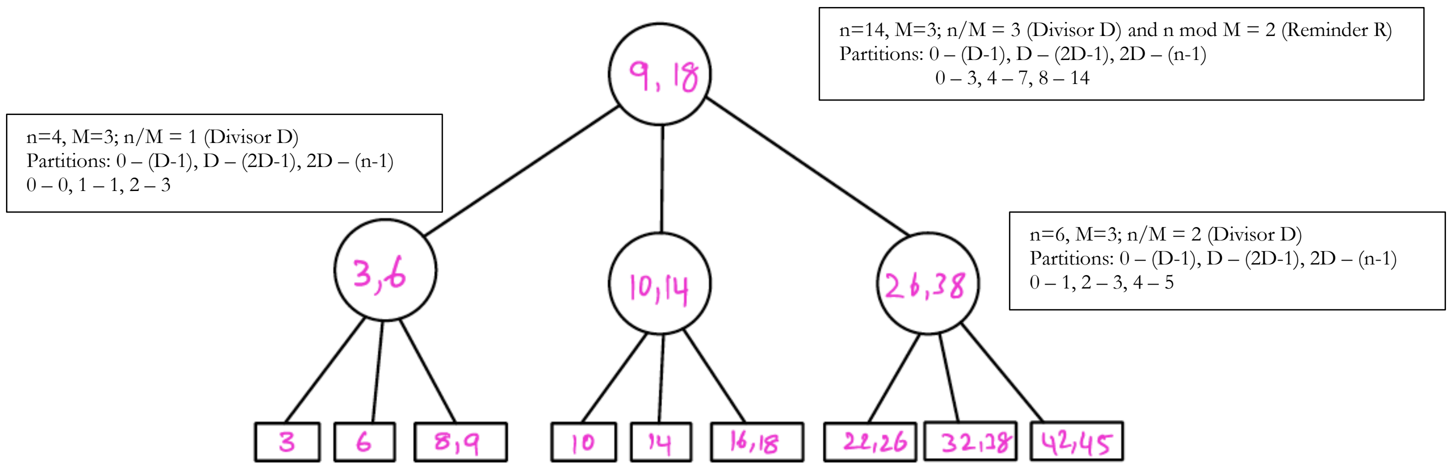
(e) Rebuild Tree Operation:

- The operation collects all the values in the leaf nodes, merges them into a single sorted array, and reconstructs the tree from scratch.

Assume that we have an array with numbers in the sorted order.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
3	6	8	9	10	14	16	18	22	26	32	38	42	45

Assume that the value of M=3, meaning there are three children and two values in each node (non-leaf). The leaf nodes will contain at least one value and, at most, M-1 values.


n=14, M=3; n/M = 3 (Divisor D) and n mod M = 2 (Reminder R)
Partitions: 0 – (D-1), D – (2D-1), 2D – (n-1)
0 – 3, 4 – 7, 8 – 14
n=4, M=3; n/M = 1 (Divisor D)
Partitions: 0 – (D-1), D – (2D-1), 2D – (n-1)
0 – 0, 1 – 1, 2 – 3
n=6, M=3; n/M = 2 (Divisor D)
Partitions: 0 – (D-1), D – (2D-1), 2D – (n-1)
0 – 1, 2 – 3, 4 – 5
9, 18
3,6
10,14
26,38
3
6
8,9
10
14
16,18
22,26
32,38
42,45

Data Structure to be Built

```
template <classname DT>
/***** Write your exception class here ******* SEE main function *****/
class MTree {
protected:
    int M;  // Maximum number of children per node (M+1 way split)
    vector<DT> values;  // Values stored in the node (M-1 values)
    vector<MTree*> children;  // Pointers to child MTrees (M+1 children)

public:
    MTree(int M);
    ~MTree();
    bool is_leaf() const;  // Check if the current node is a leaf
    void insert(const DT& value);  // Insert a value into the MTree
    void split_node();  // Split the node if it exceeds capacity (i.e >=M)
    MTree* find_child(const DT& value);  // Find the correct child to follow
    bool search(const DT& value);  // Search for a value in the MTree
    void remove(const DT& value);  // Delete a value from the MTree
    void buildTree(vector<DT>& input_values);  // Build the tree
    vector<DT>& collect_values();  // Collect values from all leaf nodes
    bool find (DT& value);
};
```

Command	Format	Description
I	`I value`	Inserts value into the tree
R	`R value`	Removes value from the tree, if not found, throw NotFoundException
B	`B`	Rebuild the tree
F	`F value`	Find value into the tree. bool is the output

Programming Objectives:

1. All code must be in C++. You will create a class given below with appropriate fields.
2. The class will have several methods whose prototypes are provided to you.
3. All input will be read via redirected input. That is, you will not open a file inside the program.
4. The class structure is shown below (you are responsible for fixing any syntax errors).
5. The structure for your main program is also provided. You will use that for your project.

The Main Program

Your program must execute the main program. We may change the main program but will ensure we only call the methods you have been asked to create.

```
#include <vector>
#include <iostream>
using namespace std;

int main() {
    int n; // number of numbers in the initial sorted array
    int MValue;
    int numCommands;
    char command;
    int value;

    vector<int> mySortedValues(n);

    //read n numbers from the input and add them to the vector mySortedValues

    //Get the M value
    cin >> M;
    MTree<int>* myTree = new MTree<int>(M);

    //Build the tree
     (*myTree).buildTree (mySortedValues);

    cin >> numCommands;  // Read the number of commands


/************** Read each command Process ***************//

    for (int i = 0; i < numCommands; i++) {
        cin >> command;  // Read the command type

        switch (command) {
            case 'I': {  // Insert
                cin >> value;
                try {
                  (*myTree).insert(value);
                catch (duplicateInsertion& e) {
                  cout ≪ "The value = " ≪ value ≪ " already in the tree."
                  cout ≪ endl;
                }
                break;
            }
            case 'R': {  // Remove
                cin >> value;
                try {
                        (*myTree).remove(value);
                }
                catch (NotFoundException& e) {
                   cout ≪ "The value = " ≪ value ≪ " not found."
                }
                            break;
            }
            case 'F': {  // Find
                cin >> value;
                if (*myTree).find(value)
                  cout ≪ "The element with value = " ≪ value ≪ " was found."
                else
                  cout ≪ "The element with value = " ≪ value ≪ " not found."
                cout ≪ endl;
                break;
            }

            case 'B': {  // rebuild tree
                vector<int> myValues = (*myTree).collect_values();
                (*myTree).buildTree (myValues);
                break;
            }

            default:
                cout << "Invalid command!" << endl;
        }
    }

    delete myTree;
    return 0;
}
```

Redirected Input: Redirected input provides you a way to send a file to the standard input of a program without typing it using the keyboard. To use redirected input in Visual Studio environment, follow these steps: After you have opened or created a new project, on the menu go to project, project properties, expand configuration properties until you see Debugging, on the right you will see a set of options, and in the command arguments type "< **input filename**". The < sign is for redirected input and the **input filename** is the name of the input file (including the path if not in the working directory).

Constraints

1. In this project, the only header you will use is #include <iostream> and #include <vector> and using namespace std.
2. None of the projects are group projects. Consulting with other members of this class our seeking coding solutions from other sources including the web on programming projects is strictly not allowed and plagiarism charges will be imposed on students who do not follow this.

Project Submission Requirements:

1. **Code Development (75%):** Implement the provided classes and the required methods Your implementation must be fully compatible with the main program provided. Your code must run successfully with the main program and the corresponding input, demonstrating the correct functionality of all classes and its methods.

   - **LLM/AI Tool Usage:** You can use Large Language Models (LLMs) or AI tools, such as GitHub Copilot, to assist in writing and refining your classes and their methods. If you did not use LLM or AI tools to write your project, you still have to show for 2. below how you would have used it to find a solution to the project. **You need to make sure that you use the class structure provided to you as the basis, and failure to do that will result in zero points for this project.**

2. **LLM and GitHub Copilot Usage Documentation (15%):** If you choose to use LLM tools or GitHub Copilot, you must document your usage. This documentation (in PDF Format) should include:

   - **Prompts and Suggestions:** Provide the specific prompts or suggestions you used, such as "Generate a method for this method" or "How can I implement this method for an MTree class?"
   - **Rationale:** Explain why you chose these prompts or suggestions and how they contributed to the development of your classes. For instance, you might describe how a suggestion helped structure the program.
   - **Incremental Development:** Detail how you used the tools to build and refine your classes and methods incrementally. For example, you might start by generating a basic structure for the classes and then refine individual methods, ensuring each integrates smoothly with the main program.

3. **Debugging and Testing Plan (10%):** Submit a comprehensive debugging and testing plan. This should include:

   - **Specific Tests:** Describe your tests on your class methods, such as checking that the compute method works correctly for various circuits.

- **Issues and Resolutions:** Document any issues you encountered, such as handling zero values or optimizing for performance, and how you resolved them.
- **Verification:** Explain how you verified that your classes work correctly with the provided main program. This could involve running a series of test cases the main program provides or creating additional test cases to ensure robustness.