CS 2413 – Data Structures – Fall 2025 – Project Two
Due 11:59 PM, November 3, 2025

Description:

In this project, you will implement a Nested Binary Search Tree (NestedBST) designed for value-driven search that returns keys. Unlike a regular BST that maps a single key to a value, the NestedBST lets you search by a K-dimensional value (one scalar per dimension) and progressively narrow the candidate **keys** until a single key remains.
Each node in the NestedBST represents ordering by **one value dimension** and stores two generic fields:

- Value (DT2) — a single attribute used for comparison at this level.
- Keys (DT1[]) — an array of candidate keys consistent with all value constraints from the root down to this node.
- Dimension — a integer value that indicates the dimension of the tree.

Traversal uses standard BST logic on the node's Value at the current dimension: smaller goes left, larger goes right. When values are equal at this dimension, search continues into the node's nextTree, which orders the same records by the next dimension. As you descend, the Keys array shrinks (it keeps only the keys that remain compatible with the value prefix you have matched so far). By the time you reach the last dimension (K-1), the Keys array is guaranteed to have size = 1, yielding the unique key whose value tuple exactly matches the query.

This nested, dimension-by-dimension refinement provides an intuitive, pointer-based alternative to multidimensional indexes: it performs binary search at every dimension, carries forward only the viable keys, and resolves ties by drilling into deeper trees until a single key remains.

- The goal of this project is to:
- Reinforce recursion, templates, and pointer-based design in C++.
- Practice building hierarchically linked BST**s** that operate on different dimensions.
- Gain experience with value-driven queries that return keys (inverse lookup).
- Develop a command-driven program using redirected input/output (no files, no STL).

The NestedBST thus offers a clean way to support queries of the form "find the key whose value tuple equals (v_0, …, v_{k-1})," while still using the familiar mechanics of binary search trees—extended across multiple dimensions and coupled with a shrinking candidate-key array that resolves to a single key at the final level.

Operations and Commands

Notation

- K = number of value dimensions (given at runtime).
- Pattern = a sequence of K tokens, each either a concrete value (type DT2) or a * wildcard.
- Keys are type DT1.
- Values are K scalars of type DT2 (one per dimension).

1) Insert (Upsert)

Command: I <key> v0 v1 ... v{K-1}
Meaning: Insert or update the mapping from the **value tuple** (v0,…,v{K-1}) to **key** <key>.
Rules:

- If (v0,…,v{K-1}) is **new**, create the nested path and record <key> at the last dimension; update candidate key lists along the equality chain.
- If (v0,…,v{K-1}) **already exists**:
    - If the stored key is the **same** as <key>, do nothing (idempotent).

- If it's a **different** key, **replace** the key (acts like update).

Output:

- Inserted key=<key> for (v0,...,v{K-1}) (first insert), or
- Updated key=<key> for (v0,...,v{K-1}) (replacement), or
- Unchanged key=<key> for (v0,...,v{K-1}) (idempotent).

Example (K=3):
I A123 10 23 5

Output Structure:

```
I 123 10 23 5
Inserted key=123 for (10,23,5)

I 123 10 23 5
Unchanged key=123 for (10,23,5)

I 556 10 23 5
Updated key=556 for (10,23,5)
```

2) Find (Wildcard Pattern)

Command: F p0 p1 ... p{K-1} where each pi is a DT2 value or *

Meaning: Return all keys whose value tuple matches the pattern (exact match on specified positions; any value on * positions).

Rules:

- At dimension d:
  - If pd is a value, do BST compare by that value (left/right); on equality, descend nextTree.
  - If pd is *, traverse both left and right (and the equality chain via nextTree) because any value at this dimension is allowed.
- At the final dimension (K-1), collect the leaf's single key.

Output:

- If matches found: one line per key, in lexicographic order of their full value tuples:
  key=<key> for (v0,...,v{K-1})
- If none: EMPTY

Examples (K=3):

- F * 23 * → all keys whose second value is 23.
- F 10 23 5 → the single key for exactly (10,23,5) if present.
- F * * * → all keys in the tree.

Output Structure

```
F * 23 *
key=123 for (10,23,5)
key=991 for (9,23,7)

F 10 23 5
key=123 for (10,23,5)

F * * *
key=123 for (10,23,5)
key=991 for (9,23,7)
```

```
key=210 for (12,11,4)

F 50 * *
EMPTY
```

2) Display (Dimension-Aware Dump)

Command: D
Meaning: Human-readable proof the tree is built. Perform inorder at each dimension, printing each node's value for that dimension and a small summary of the candidate set; if a nextTree exists, print it indented with its next dimension.
Output Structure (example, K=3):

```
[dim 0] val=10  (candidates=5)
  -> dim 1
    [dim 1] val=23  (candidates=2)
      -> dim 2
        [dim 2] val=5  key=123
        [dim 2] val=7  key=991
    [dim 1] val=30  (candidates=1)
      -> dim 2
        [dim 2] val=1  key=210
[dim 0] val=12  (candidates=3)
  ...
```

- At [dim K-1], print the unique leaf as key=<key>.
- At higher dims, candidates = number of keys reachable through that node's equality chain.

Recursive steps for display method are provided below:

Steps: (in order traversal)

1. If root is NULL, return (nothing to display).
2. Recursively display the **left subtree**:
    - Call Display(root.left, dimension, indent).
3. Print the current node's information:
    - Indent output by indent spaces.
    - Print [dim dimension] value = <root.value>.
    - If this node's dimension is the last one (K−1), print its key.
    - If not the last dimension, also print how many candidate keys are below this node (optional summary).
4. If a nested tree exists (root.nextTree is not NULL):
    - Increase indentation by two spaces.
    - Print a marker line:
      indent spaces + "-> dim (dimension + 1)"
    - Recursively display the nested tree:
      Display(root.nextTree, dimension + 1, indent + 2).
5. Recursively display the **right subtree**:
    - Call Display(root.right, dimension, indent).

Implementation Details

- You are provided with **boilerplate code** containing:
    - **Class definitions – Add more methods if needed**
    - A **main function**
    - Comments to help you get started.
- You must ensure your **output format matches** the provided sample output to **pass the autograder**.

```
template <class DT>
class NestedBST {
    public:
        DT value;                   // value of the node
        vector<int> keys;           // vector of keys associated with this value
        int dimension;              // dimension of this node

        NestedBST* left;            // pointer to left child
        NestedBST* right;           // pointer to right child
        NestedBST* innerTree;       // pointer to nested BST (next dimension)

        // ----- Constructors -----
        NestedBST();                // default constructor
        NestedBST(DT val, int dim); // parameterized constructor

        // ----- Core Operations -----
        void insert(int key, const vector<DT>& values); // insert or update a key-value tuple
        void find(const vector<DT>& pattern); // find keys matching a pattern with wildcards
        void display(int indent = 0);    // print tree structure for verification
};
```

```
int main() {

    int numDimensions;   // number of value dimensions
    cin >> numDimensions;

    // Create the root tree (dimension 0)
    NestedBST<int>* root = new NestedBST<int>();
    root->dimension = 0;

    int numCommands;
    cin >> numCommands;

    char command;
    for (int i = 0; i < numCommands; i++) {
        cin >> command;
        switch (command) {
            case 'I': { // Insert
                int key;
                cin >> key;
                vector<int> values(numDimensions);
                for (int d = 0; d < numDimensions; d++) {
                    cin >> values[d];
                }
                root->insert(key, values);
                cout << "Inserted key=" << key << " for (";
                for (int d = 0; d < numDimensions; d++) {
                    cout << values[d];
                    if (d < numDimensions - 1) cout << ",";
```

```
                }
                cout << ")" << endl;
                break;
            }

            case 'F': { // Find
                vector<int> pattern(numDimensions);
                string token;
                for (int d = 0; d < numDimensions; d++) {
                    cin >> token;
                    if (token == "*") {
                        // Represent wildcard as sentinel (e.g., -999999)
                        pattern[d] = -999999;
                    } else {
                        pattern[d] = stoi(token);
                    }
                }
                cout << "Find pattern: (";
                for (int d = 0; d < numDimensions; d++) {
                    if (pattern[d] == -999999) cout << "*";
                    else cout << pattern[d];
                    if (d < numDimensions - 1) cout << ",";
                }
                cout << ")" << endl;
                root->find(pattern);
                break;
            }

            case 'D': { // Display
                cout << "NestedBST Structure:" << endl;
                root->display();
                break;
            }

            default:
                cout << "Unknown command: " << command << endl;
                break;
        }
    }
    // Clean up
    delete root;
    return 0;
}
```

Input Format

```
3
12
I 101 10 23 5
I 202 10 23 7
I 303 12 11 4
I 404 10 30 1
```

```
F * 23 *
F 10 23 5
F 12 * *
D
I 101 10 23 5
I 999 10 23 5
F * * *
F 99 * *
D
```

Output Format

```
Inserted key=101 for (10,23,5)
Inserted key=202 for (10,23,7)
Inserted key=303 for (12,11,4)
Inserted key=404 for (10,30,1)
key=101 for (10,23,5)
key=202 for (10,23,7)
key=101 for (10,23,5)
key=303 for (12,11,4)

NestedBST Structure:
[dim 0] value=10  (candidates=3)
  -> dim 1
    [dim 1] value=23  (candidates=2)
      -> dim 2
        [dim 2] value=5  key=101
        [dim 2] value=7  key=202
    [dim 1] value=30  (candidates=1)
      -> dim 2
        [dim 2] value=1  key=404
[dim 0] value=12  (candidates=1)
  -> dim 1
    [dim 1] value=11  (candidates=1)
      -> dim 2
        [dim 2] value=4  key=303

Unchanged key=101 for (10,23,5)
Updated key=999 for (10,23,5)
key=202 for (10,23,7)
key=303 for (12,11,4)
key=404 for (10,30,1)
key=Z999 for (10,23,5)
EMPTY

NestedBST Structure:
[dim 0] value=10  (candidates=3)
  -> dim 1
    [dim 1] value=23  (candidates=2)
      -> dim 2
        [dim 2] value=5  key=999
        [dim 2] value=7  key=202
    [dim 1] value=30  (candidates=1)
      -> dim 2
        [dim 2] value=1  key=404
[dim 0] value=12  (candidates=1)
```

```
  -> dim 1
    [dim 1] value=11  (candidates=1)
      -> dim 2
        [dim 2] value=4  key=303
```

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). A simple program that reads a matrix can be found below.

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

int main () {

        int r,c,nsv;
        cin >> r >> c >> cv;
          cout << r << c << cv << endl;
        for (int i=0; i < nsv; i++) {
           cin >> rn >> cn >> val;
           cout << rn << cn << val << endl;
        }
        return 0;
}
```

Constraints

1. In this project, the headers you will use is #include <iostream> using namespace std, and #include<string>
2. None of the projects is a group project. 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: Nested Binary Search Tree (NestedBST)

1. Code Development (75%)

You must implement the provided NestedBST class structure to support a multidimensional binary search tree representation with hierarchical nesting across dimensions. Your implementation must be fully compatible with the provided main() skeleton. Specifically, your code must include methods to:

Insert (I)

- Add or update a record defined by a multi-dimensional value tuple (v_0, v_1, …, v_{k-1}) associated with a given key.
- Create a new node if the value tuple does not exist.
- If a node with the same value tuple already exists:
  - Replace the key if it differs (output: *Updated*).
  - Do nothing if both key and tuple are identical (output: *Unchanged*).
- Correctly navigate the tree:
  - Compare using the current dimension's value.
  - For equal values, traverse into the innerTree (next dimension).
  - For smaller/larger values, traverse into left or right respectively.

Find (F)

- Accept a value pattern consisting of actual values and wildcards (*) for any dimension.
- Traverse the tree recursively:
  - When a wildcard is encountered, explore both left and right subtrees (and the equality chain).
  - When a specific value is given, perform standard BST comparisons at that dimension.
- Print all matching records in lexicographic order:
  - Output format: key=<key> for (v_0,…,v_{k-1})
  - If no matches, print EMPTY.

Display (D)

- Perform a dimension-aware inorder traversal of the NestedBST.
- Show each node's value, its current dimension, and nested structure.
- For the deepest dimension (K−1), display the associated key.
- Indent nested trees to visually represent hierarchy.
- Used to verify that the NestedBST has been constructed correctly.

Correct Execution

- Your program must exactly match the provided input/output format for all commands.
- It must support redirected input and output (no STL containers for structural storage, no file I/O).
- The implementation must use templates and pointers as specified.
- Each node must contain:
  - DT value
  - vector<int> keys
  - int dimension
  - NestedBST *left, *right, *innerTree
- Failure to follow the provided class structure will result in zero points for this section.

2. LLM and GitHub Copilot Usage Documentation (15%)

If you use AI tools, you must submit a PDF report documenting your usage. This report should include:

- Prompts and Suggestions:
  Include actual AI prompts used during development.
  Example prompts:
  - "Generate a C++ recursive insert method for a nested binary search tree by dimension."
  - "Explain how to implement wildcard-based search in a recursive tree traversal."
- Rationale for AI Usage:
  Explain why you used particular prompts and how AI suggestions shaped your implementation.
  Describe how AI tools helped refine logic, resolve recursion issues, or debug insertion paths.

- Incremental Development:
  Show how AI-assisted suggestions evolved across development steps.
  If you modified or rejected parts of generated code, describe what you changed and why.

3. Debugging and Testing Plan (10%)

You must provide a detailed testing and debugging plan to verify correctness. Include the following:

a) Specific Test Cases

- Verify that Insert correctly builds nested trees across dimensions.
- Ensure that Find with wildcards (*) returns all matching keys.
- Confirm that Display accurately shows the tree hierarchy.
- Test deep nesting (multiple equal dimensions) and shallow cases (few dimensions).
- Verify that duplicate inserts are handled correctly (Updated/Unchanged cases).

b) Issues and Resolutions

Document major implementation challenges such as:

- Managing recursion across dimensions and trees.
- Preventing infinite loops when equal values occur repeatedly.
- Ensuring consistent indentation and dimension labeling in Display.
- Avoiding memory leaks when freeing subtrees.
- Handling wildcard expansion efficiently without duplication.

c) Verification

Describe how you verified that your implementation is correct.

Include edge cases, such as:

- Inserting into an empty tree.
- Searching with all wildcards (F * * *).
- Searching with no matches.
- Updating an existing value tuple with a new key.
- Displaying after multiple nested insertions.

Example Submission Structure

1. Code Implementation (.cpp)
   - Submit a single .cpp file containing:
     - Implementation of the NestedBST class with methods insert(), find(), and display().
     - A main() function that reads commands and produces output in the exact specified format.
2. LLM Usage Documentation (PDF)
   - Provide screenshots or text logs of AI interactions.
   - Include explanations of how AI tools contributed to your solution.
3. Testing Plan (PDF or .txt)
   - Include your debugging steps, identified issues, and resolutions.
   - Provide sample input/output verifying correctness of insert, find (with wildcards), and display.