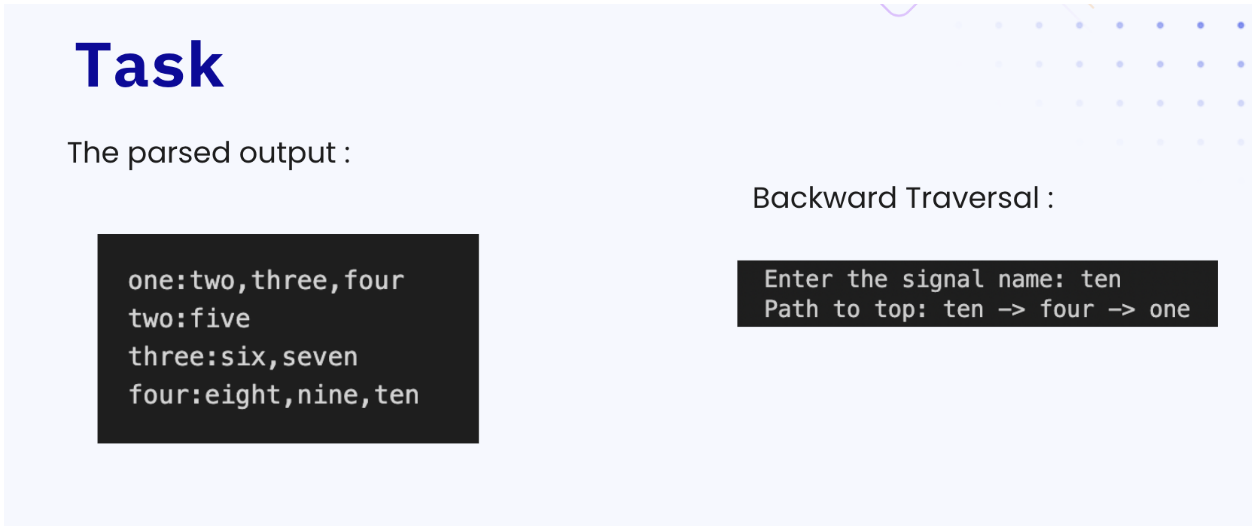
REPORT



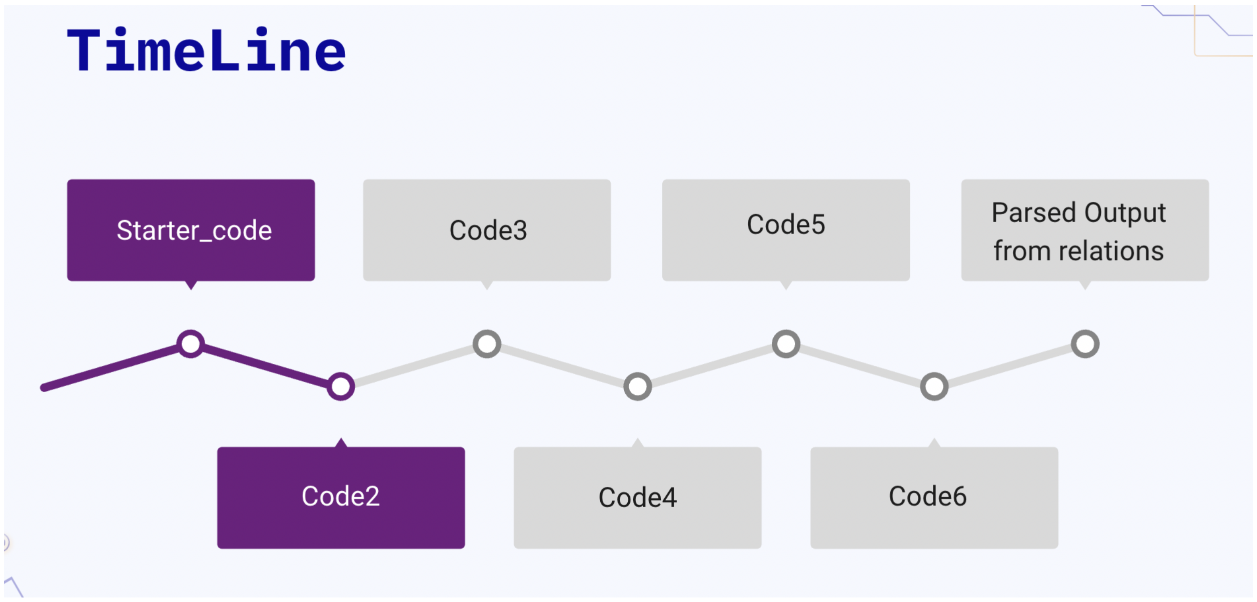
Task

The parsed output :

```
one:two,three,four
two:five
three:six,seven
four:eight,nine,ten
```

Backward Traversal :

```
Enter the signal name: ten
Path to top: ten -> four -> one
```



TimeLine

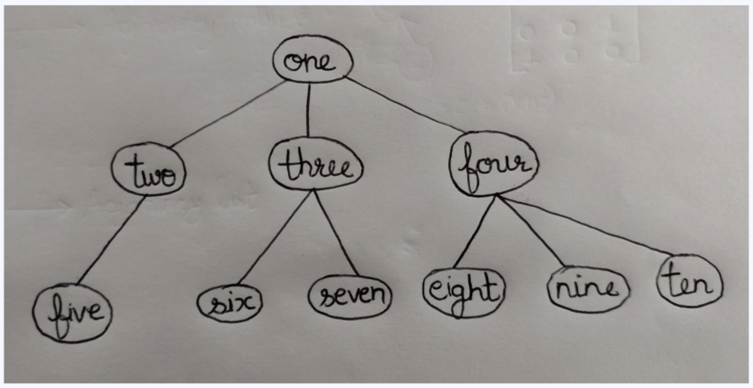
1) Starter_code.cpp :

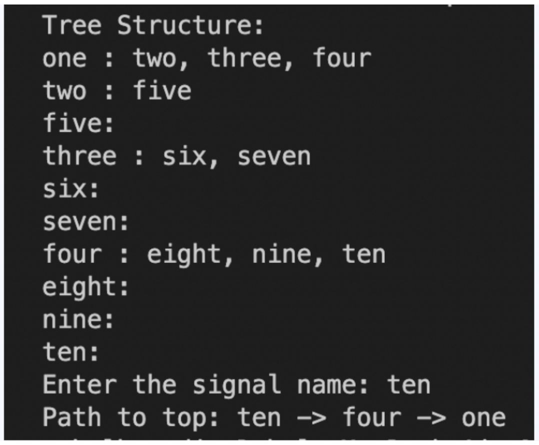
- The code implements a tree data structure using a class template called TreeNode<string>.
- The function takeInputLevelWise() allows the user to input the tree nodes level by level, creating the tree structure.
- The function printTree() prints the tree in a hierarchical format, showing the parent node followed by its children nodes.
- The function findNodePath() searches for a given target node in the tree and returns the path from the target node to the root.
- The main() function creates the tree, prints it, and prompts the user to enter a signal name to find its path.
- If the target node is found, the code prints the path from the target node to the root. Otherwise, it displays a "Signal not found!" message.
- The code utilizes recursion to traverse the tree and perform the necessary operations.

2) Code 2 : Inserting parsed output directly into the code in hardcoded form

- The code constructs a tree data structure based on a given set of input lines.
- The TreeNode structure represents a node in the tree and contains a string value and a vector of pointers to its children nodes.
- The constructTree() function parses each input line, extracts the parent and children information, and creates the corresponding nodes.
- The nodes are stored in a map called nodeMap, where the string value of the node is mapped to the node pointer.
- The printTree() function recursively prints the tree starting from a given node, displaying the value of each node and its children.
- In the main() function, a set of input lines is provided, and the constructTree() function is called to build the tree.
- The root node, identified by the value "one", is retrieved from the nodeMap and assigned to the root pointer.
- If the root node is found, the printTree() function is called with the root node to display the tree structure.
- Finally, memory cleanup is performed by deleting all the nodes in the nodeMap.
- Note that the code assumes that the input lines follow a specific format, where each line contains the parent node value followed by a colon and a comma-separated list of child node values.

Output for Code 2 to Code 6 :





```
Tree Structure:
one : two, three, four
two : five
five:
three : six, seven
six:
seven:
four : eight, nine, ten
eight:
nine:
ten:
Enter the signal name: ten
Path to top: ten -> four -> one
```

3) Code 3 : Extracting parsed output from a text file and then feeding it to the code

- 1. Code3 added the functionality to read input from a file named "input2.txt" using an ifstream object. It checks if the file is successfully opened and reads the input lines from the file into a vector of strings called inputLines.

- The constructTree( ) function in the Code3 includes additional logic to trim leading and trailing whitespaces from the parent and children strings before creating the nodes. This ensures that any unwanted whitespaces are removed.

- The Code3 introduced two new functions: printPathToTop() and isRootSignal(). printPathToTop() recursively finds the path from a given node to the top of the tree (root) and stores it in a vector called path. isRootSignal() checks if a given node is the root signal by comparing it with all other nodes in the nodeMap and their children.

- The main function in the Code3 reads the input from the file, constructs the tree, searches for the root signal, and prints the tree structure using printTree(). It prompts the user to enter a signal name and finds the path from the entered signal to the root if it exists.

- The Code3 uses dynamic memory allocation to create tree nodes, and it ensures memory cleanup by deleting all the nodes in the nodeMap at the end of the program.

Overall, the Code3 expands upon the functionality of the Code2 by reading input from a file, trimming whitespaces, and introducing new functions for path finding and root signal checking.

.

4) Code4.cpp : Optimizing Code3

We optimized Code3 by following ways:

- Using const references for function parameters: In some functions like printTree and printPathToTop, pass the tree nodes as const TreeNode* or const TreeNode& to avoid unnecessary copies of the nodes.

- Improving data structures and algorithms: Using a hash-based data structure std::unordered_map instead of std::map can provide faster lookup times.

5) Code5.cpp : Using unordered_map instead of TreeNode class

Code 5 differs from Code 4 in the following ways:

1) Data Structure:

Code5 uses an unordered map nodeMap with string keys and a pair of string and vector<string> as the value type.
Code4 uses an unordered map nodeMap with string keys and TreeNode* as the value type. The tree nodes are represented using a custom TreeNode struct.

2) Trimming Whitespace:

Code5 trims leading and trailing whitespace from the parent and children strings using find_first_not_of() and find_last_not_of() functions.
Code4 also trims leading and trailing whitespace from the parent and children strings but uses substr() and find_first_not_of() functions.

3) Printing the Tree:

Code5 uses a recursive function printTree() that takes the node value as a string and uses nodeMap to access the children nodes.
Code4 uses a recursive function printTree() that takes a TreeNode* and directly accesses the children of the node.

4) Path Finding and Root Signal Checking:

Code5 uses the function printPathToTop() that takes strings as arguments and uses nodeMap to access the children nodes.
Code4 uses the function printPathToTop() that takes TreeNode* as arguments and directly accesses the children of the node.
Root signal checking function isRootSignal() also differs in how it checks for the root signal. Code5 uses the find() function from the algorithm library, while Code4 uses a loop to iterate over the children vector of each node.

6) Code6.cpp : Making a new class named Node

- The tree structure is represented using a custom struct called Node.
- The nodeMap is an unordered map with string keys and Node values. It stores the parent node as a string and the children nodes as a vector of strings.
- The constructTree function creates and populates the nodeMap by parsing the input string and storing the parent and children nodes in the Node structure.
- The printTree function recursively traverses the tree using the nodeMap and prints the node values along with their children.
- The printPathToTop function recursively finds the path from a given node to the top of the tree using the nodeMap and stores it in a vector of strings.
- The isRootSignal function checks if a given node is a root signal by iterating over the nodeMap and checking if the node is present as a child in any of the nodes.

Code 3 , Code 4 , Code 5 and Code 6 give the same output. They are just the optimized version of previous codes.

7) Convert.cpp : Outputs parsed output from relations

- The splitString function takes a string and a delimiter as input and splits the string into tokens based on the delimiter. It returns a vector of tokens.

- The trimString function takes a string and removes leading and trailing whitespaces. It returns the trimmed string.

- The main function:

  Opens the input file (input_pre.txt) for reading.
  Checks for a Byte Order Mark (BOM) at the beginning of the file and skips it if present.
  Creates a map called relationships to store parent-child relationships. The key is a parent string, and the value is a vector of children strings.
  Reads each line from the input file, trims the line, and splits it into two parts using the "<=" delimiter. The left part is considered the parent, and the right part is split further using the "&&" delimiter to get the children.
  Trims each child string and adds the parent-child relationship to the relationships map.
  Closes the input file.

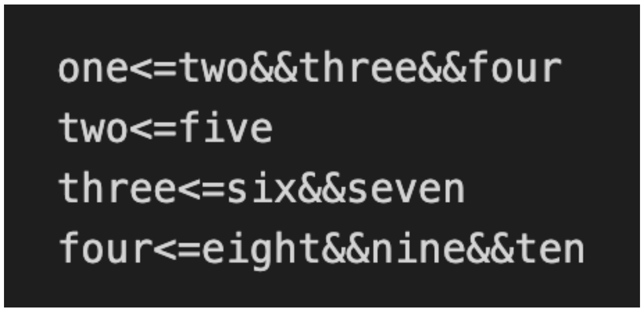
- Opens the output file (input_post.txt) for writing.

- Writes the relationships to the output file in the desired format:

  Iterates over the relationships map.
  Writes the parent string followed by a colon (":").
  Retrieves the children vector for the current parent and makes a copy.
  Sorts the copy of children vector in alphabetical order.
  Writes each child string to the output file, separated by commas.
  Closes the line with a newline character.
  Continues to the next parent-child relationship.

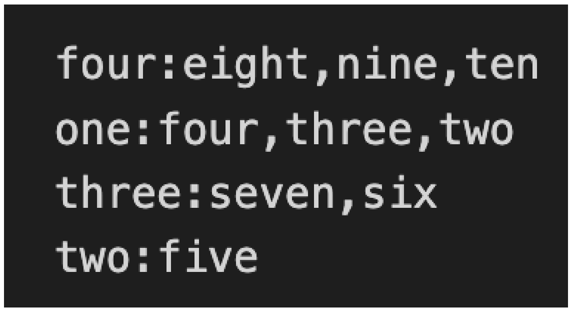
- Closes the output file.
- Prints a message indicating that the output has been written to the input_post.txt file.

Output :



```
one<=two&&three&&four
two<=five
three<=six&&seven
four<=eight&&nine&&ten
```

=>



```
four:eight,nine,ten
one:four,three,two
three:seven,six
two:five
```