EAST WEST UNIVERSITY

P R O J E C T R E P O R T

Word Dictionary

Department of CSE
5th Semester, Summer – 2023

Course Code : CSE207
Course Title : Data Structures

Submitted By:

Group Number : 05
Junnun Mohamed Karim (ID: 2022-1-60-108)
Md Murad Khan Limon (ID: 2022-1-60-044)
Md. Yousuf Hozaifa (ID: 2022-1-60-162)

Submitted To:

Dr. Mohammad Manzurul Islam
Assistant Professor
Department of Computer Science & Engineering

Submission Date:
3rd September, 2023

Table of Contents

1 INTRODUCTION

1.1 Background

In the realm of modern communication and information retrieval, dictionaries serve as fundamental tools for understanding language nuances, word meanings, and contextual usage. The evolution of dictionaries from traditional printed forms to digital platforms has brought about increased accessibility and dynamic functionalities. This project endeavors to develop a sophisticated digital word dictionary leveraging the power of a Binary Search Tree (BST) as its foundational data structure. The dictionary's core functionalities encompass word addition, retrieval, removal, and the innovative feature of suggesting akin words to aid users in their quest for accurate word definitions.

1.2 Problem Statement

The project aims to develop a digital word dictionary using a Binary Search Tree (BST) as the underlying data structure. The main functionalities of the dictionary include adding new words, searching for words, deleting words, and finding words with similar meanings. The project also introduces the feature of providing word suggestions when a word is not found during a search, to enhance user experience.

The key challenges that need to be addressed in this project are as follows:

1. **BST Implementation:** Designing and implementing a Binary Search Tree to store and manage the list of words efficiently while maintaining the order for quick access during search operations.

2. **Addition and Deletion:** Enabling the addition and deletion of words while ensuring that the BST properties are maintained at all times. This includes proper insertion and removal of nodes in the BST.

3. **Search Functionality:** Developing a search function that can quickly locate a word in the BST. If the word is not found, providing suggestions for similar words to assist the user in finding the correct term.

4. **Word Suggestion:** Implementing a mechanism to suggest word.

2 Objective

Our primary goal is to craft a user-friendly digital word dictionary that brings the richness of language to users' fingertips. We intend to achieve this by harnessing the capabilities of a Binary Search Tree, ensuring swift and accurate word retrieval. With a focus on enhancing user experience, we aim to seamlessly integrate functions for adding and removing words, making the dictionary a dynamic repository. Our objective extends beyond mere word definitions; we aspire to provide intuitive search functionality that not only locates words but also offers suggestions for closely related terms, thereby aiding users in their linguistic explorations. By incorporating these elements and enabling the import of data from text files, our project seeks to empower users with a comprehensive language companion that adapts to their language-learning journey.

3 PROBLEM ANALYSIS

3.1 Understanding the Problem

The core challenge of this project lies in creating a functional word dictionary using a Binary Search Tree (BST) structure. This entails ensuring the proper insertion, deletion, and retrieval of words while maintaining the inherent ordering of the BST. Additionally, implementing a search functionality that suggests similar words when a query term is not found adds another layer of complexity. The project demands a deep comprehension of BSTs, linguistics, and algorithms to create a seamless user experience.

3.2 Input Requirements

To construct the dictionary, we require a dataset of words paired with their respective definitions. These entries could be sourced from a text file, each containing a word and its meaning. Additionally, user inputs for adding, searching, and deleting words will be necessary to dynamically manipulate the dictionary.

3.3 Output Requirements

The dictionary must offer precise word definitions and meanings to users upon request. In the case of a search for a non-existent word, the system should intelligently suggest closely related words to guide users. For deletions and additions, appropriate confirmations or notifications must be generated. The system's output should be presented in a user-friendly and readable format.

3.4 Processing Requirements

The project involves intricate processing tasks. Constructing and maintaining the Binary Search Tree requires efficient algorithms for insertion and deletion while ensuring the tree's ordered property. The search mechanism should efficiently navigate the tree and offer word suggestions based on partial queries. The determination of similar meaning words requires leveraging linguistic databases or synonym matching algorithms, adding a layer of semantic analysis to the processing requirements.

3.5 Technical Feasibility

From a technical perspective, creating a word dictionary with BSTs is feasible. BSTs offer logarithmic time complexity for search operations. Implementing word suggestion mechanisms can rely on techniques like Levenshtein distance or utilizing pre-existing synonym databases. Reading and writing data from text files is a common task, and building a user interface for interaction is achievable using programming languages and frameworks. However, ensuring efficient performance and user-friendliness may require careful optimization and design considerations.

4 ALGORITHM DEVELOPMENT

4.1 Binary Search Tree (BST)

The main data structure that we've used in this project is Binary Search Tree (BST). The implementation of the Binary Search Tree (BST) data structure is within the context of a C++ header and template file. The BST serves as a foundational component for creating a dynamic and efficient word dictionary, in line with the project's objectives.

The **bst.h** file defines the BST class within the namespace **ds** (presumably short for "data structure"). The template class **bst** is parameterized by the data type **T**, which could be any type that supports comparison operations. The class structure includes a nested **Node** struct, each representing a node within the BST. Nodes contain the actual data, pointers to left and right child nodes, and are equipped with constructors for initialization.

The **bst** class itself comprises private and public sections, adhering to encapsulation principles:

Private Section:

Private Member Variables: The class maintains a count of nodes (`size_t count`) and a pointer to the root node (`std::unique_ptr<Node> root`).

Private Helper Functions: Several private helper functions implement critical operations for the BST:

`insert_helper()`	Recursively adds a new node to the BST while maintaining the proper ordering.
`search_helper()`	Recursively searches for a node with a specific key and returns a pointer to it.
`remove_helper()`	Recursively removes a node with a specific value while preserving the BST properties.
`print_helper()`	Recursively prints the BST structure, indicating the relationship between nodes.

Public Section:

Constructor and Destructor: The public interface provides a constructor and destructor for creating and cleaning up the BST.

Public Member Functions: The key functionalities are exposed through the public interface:

`insert()`	Inserts a new element into the BST, ensuring the proper ordering.
`remove()`	Removes an element from the BST while maintaining its structure.
`search()`	Searches for an element in the BST and returns a pointer to its data.
`search_node()`	Searches for a node with a specific key and returns a pointer to it.
`print()`	Prints the structure of the BST for visualization.

Utility Functions: Additional functions provide information about the BST's state:

`is_empty()`	Checks if the BST is empty.
`is_full()`	Determines whether the BST is full (each node has either zero or two children).
`get_count()`	Returns the count of nodes in the BST.
`get_root()`	Retrieves a pointer to the root node.

```
ALGORITHM 4-1-1 Recursive Insert Helper

Algorithm insert_helper(node, value)

This algorithm recursively inserts a new node with the given value into the
binary search tree.

  Pre    node is a pointer to the current node
         value is the value to be inserted
  Post   New node with the value is inserted into the binary search tree under
         the appropriate node
  Return true if insertion is successful; false otherwise

1 try
  1 If node is null:
    1 Create a new Node with data set to value
    2 Set node to point to the new Node
    3 Return true
2 else if value < node.data:
  1 Return the result of insert_helper(node.left, value)
3 else if value > node.data:
  1 Return the result of insert_helper(node.right, value)
4 else:
  1 Return false
5 catch std::bad_alloc e
  1 Print "Memory allocation failed: " concatenated with e.what()
  2 Return false
end insert_helper
```

```
ALGORITHM 4-1-2 Insert Node into Binary Search Tree

Algorithm insert(value)

This algorithm inserts a new node with the given value into the binary search
tree.

  Pre    value is the value to be inserted
  Post   New node with the value is inserted into the binary search tree
  Return true if insertion is successful; false otherwise

1 try
  1 If insert_helper(root, value) returns true:
```

```
    1 Increment count by 1
    2 Return true
  2 Else:
    1 Return false
2 catch std::bad_alloc e
  1 Print "Memory allocation failed: " concatenated with e.what()
  2 Return false
end insert
```

ALGORITHM 4-1-3 Recursive Remove Helper

```
Algorithm remove_helper(node, value)

This algorithm recursively removes a node with the given value from the binary
search tree.

  Pre    node is a pointer to the current node
         value is the value to be removed
  Post   Node with the value is removed from the binary search tree
  Return true if removal is successful; false otherwise

1 try
  1 If node is null:
    1 Return false
2 else if value < node.data:
  1 Return the result of remove_helper(node.left, value)
3 else if value > node.data:
  1 Return the result of remove_helper(node.right, value)
4 else:
  1 If node.left is null:
    1 Set node to move(node.right)
  2 Else if node.right is null:
    1 Set node to move(node.left)
  3 Else:
    1 Set successor to node.right
    2 While successor.left is not null:
      1 Set successor to successor.left
    3 Set node.data to successor.data
    4 Call remove_helper(node.right, successor.data)
  2 Return true
5 catch std::exception e
  1 Print "Exception occurred: " concatenated with e.what()
  2 Return false
end remove_helper
```

ALGORITHM 4-1-4 Remove Node from Binary Search Tree

```
Algorithm remove(value)

This algorithm removes a node with the given value from the binary search
tree.
```

```
  Pre    value is the value to be removed
  Post   Node with the value is removed from the binary search tree
  Return true if removal is successful; false otherwise

1 try
  1 If remove_helper(root, value) returns true:
    1 Decrement count by 1
    2 Return true
  2 Else:
    1 Return false
2 catch std::exception e
  1 Print "Exception occurred: " concatenated with e.what()
  2 Return false
end remove
```

ALGORITHM 4-1-5 Recursive Search Helper

```
Algorithm search_helper(node, key)

This algorithm recursively searches for a node with the given key in the
binary search tree.

  Pre    node is a pointer to the current node
         key is the key to search for
  Post   Node with the key is found and returned; null if not found
  Return Pointer to the node with the key; null if not found

1 try
  1 If node is null:
    1 Return null
2 else if key < node.data:
  1 Return the result of search_helper(node.left, key)
3 else if key > node.data:
  1 Return the result of search_helper(node.right, key)
4 else:
  1 Return node
5 catch std::exception e
  1 Print "Exception occurred: " concatenated with e.what()
  2 Return null
end search_helper
```

ALGORITHM 4-1-6 Search Node Data

```
Algorithm search(key)

This algorithm searches for a node with the given key in the binary search
tree and returns a pointer to its data.

  Pre    key is the key to search for
  Post   Node with the key is found and its data is returned; nullptr if not
         found
  Return Pointer to the data of the node with the key; nullptr if not found
```

```
1 try
  1 Return a pointer to the data of the node returned by search_helper(root,
key)
2 catch std::exception e
  1 Print "Exception occurred: " concatenated with e.what()
  2 Return nullptr
end search
```

ALGORITHM 4-1-7 Search Node

```
Algorithm search_node(key)

This algorithm searches for a node with the given key in the binary search
tree and returns a pointer to the node itself.

  Pre    key is the key to search for
  Post   Node with the key is found and returned; nullptr if not found
  Return Pointer to the node with the key; nullptr if not found

1 try
  1 Return the result of search_helper(root, key)
2 catch std::exception e
  1 Print "Exception occurred: " concatenated with e.what()
  2 Return nullptr
end search_node
```

ALGORITHM 4-1-8 Recursive Print Helper

```
Algorithm print_helper(root, level, prefix)

This algorithm recursively prints the nodes of the binary search tree in a
structured format.

  Pre  root is a pointer to the current node being printed
       level specifies the level of the node in the tree
       prefix is the string indicating the node type (root, left, right)
  Post Nodes of the binary search tree are printed in a structured format

1 If root is not null:
  1 If level is 0:
    1 Print prefix concatenated with root.data
  2 Else:
    1 Set indent to a string of level * 4 spaces
    2 Print indent concatenated with "└── " and prefix concatenated with
root.data
2 If root.left is not null or root.right is not null:
  1 Call print_helper(root.left, level + 1, "Left: ")
  2 Call print_helper(root.right, level + 1, "Right: ")
end print_helper
```

ALGORITHM 4-1-9 Print Binary Search Tree

```
Algorithm print()

This algorithm prints the nodes of the binary search tree in a structured
format.

  Pre  None
  Post Nodes of the binary search tree are printed in a structured format

1 If root is not null:
  1 Call print_helper(root.get(), 0, "Root: ")
2 Else:
  1 Print "Binary-Search Tree is empty!"
3 Clear the input stream
4 Ignore characters in the input stream up to the newline character
end print
```

```
ALGORITHM 4-1-10 Check if Binary Search Tree is Empty

Algorithm is_empty()

This algorithm checks whether the binary search tree is empty or not.

  Pre  None
  Post True is returned if the binary search tree is empty; false otherwise

1 Return (count = 0)
end is_empty
```

```
ALGORITHM 4-1-11 Recursive Check for Fullness Helper

Algorithm is_full_helper(node)

This algorithm recursively checks whether the binary search tree is full (each
node has either 0 or 2 children).

  Pre  node is a pointer to the current node being checked
  Post True is returned if the binary search tree rooted at the given node is
       full; false otherwise

1 try
  1 If node is null:
    1 Return true
  2 If node has one child but not both:
    1 Return false
  3 Return the logical AND of is_full_helper(node.left) and
is_full_helper(node.right)
2 catch std::exception e
  1 Print "Exception occurred: " concatenated with e.what()
  2 Return false
end is_full_helper
```

```
ALGORITHM 4-1-12 Check if Binary Search Tree is Full
```

```
Algorithm is_full()

This algorithm checks whether the binary search tree is full (each node has
either 0 or 2 children).

  Pre  None
  Post True is returned if the binary search tree is full; false otherwise

1 try
  1 Return the result of is_full_helper(root)
2 catch std::exception e
  1 Print "Exception occurred: " concatenated with e.what()
  2 Return false
end is_full
```

```
ALGORITHM 4-1-13 Get Node Count

Algorithm get_count()

This algorithm retrieves the count of nodes in the binary search tree.

  Pre  None
  Post Count of nodes in the binary search tree is returned

1 Return count
end get_count
```

```
ALGORITHM 4-1-14 Get Root Node

Algorithm get_root()

This algorithm retrieves a pointer to the root node of the binary search tree.

  Pre  None
  Post Pointer to the root node of the binary search tree is returned

1 Return root.get()
end get_root
```

4.2 List Data Structure

"list.h" defines a C++ header file that contains the template implementation of a singly linked list data structure. This linked list is designed to store elements of a specified type (templated type T). The list class contains private member variables and public member functions that enable the manipulation and management of the linked list. Here's an overview of the key components in "list.h":

Node Structure: The node structure defines the building block of the linked list. Each node contains two fields: the data of type T and a pointer to the next node.

Private Member Variables:

- head: A pointer to the first node in the linked list.

- size: An integer representing the current number of nodes in the linked list.

Public Member Functions:

- Constructor and Destructor: Initialize the linked list and release memory when the list is destroyed.
- Insertion Functions: Insert elements at the front or back of the list.
- Deletion Functions: Remove elements from the front, back, or based on a specific key.
- Search Function: Check if a given element is present in the list.
- Print Function: Display the elements of the list.
- Traversal Functions: Access the beginning and end nodes of the list, and retrieve the size of the list.
- Indexing Operator: Access elements by index.

Templated Class: The list class is defined as a template class, allowing it to work with various data types.

"list.tpp" is the implementation file that complements the declarations in "list.h." It contains the definitions of the member functions of the list class. The provided definitions outline how each member function operates and interacts with the linked list. Here's a brief overview of the contents of "list.tpp":

Constructor and Destructor:

- Definition of the constructor that initializes the linked list.
- Definition of the destructor that frees memory used by the nodes.

Insertion and Deletion Functions:

- Definitions of functions that insert elements at the front or back of the list.
- Definitions of functions that delete elements from the front, back, or based on a specific key.

Search and Print Functions:

- Definition of the function that searches for an element in the list.
- Definition of the function that prints the contents of the list.

Traversal and Size Functions:

- Definitions of functions that provide access to the beginning and end nodes of the list, and retrieve the size.

Indexing Operator:

- Definition of the indexing operator that allows access to elements by index.

ALGORITHM 4-2-1 Insert Front

```
Algorithm insert_front(data)

This algorithm inserts a new node with the given data at the front of the
linked list.

  Pre    data is the value to be inserted
  Post   A new node is inserted at the front of the linked list
  Return true if insertion is successful; false if memory allocation fails

1 try
  1 Create a new node with the given data and assign it to new_node
  2 Set the next pointer of new_node to point to the current head node
  3 Update the head pointer to point to the new_node, making it the new head
  4 Increment the size of the linked list
  5 Return true to indicate successful insertion
2 catch std::bad_alloc e
  1   Output "Unable to allocate more memory"
  2   Return false to indicate unsuccessful insertion due to memory allocation
end insert_front
```

ALGORITHM 4-2-2 Insert Back

```
Algorithm insert_back(data)

This algorithm inserts a new node with the given data at the end of the linked
list.

    Pre    data is the value to be inserted
    Post   A new node is inserted at the end of the linked list
    Return true if insertion is successful; false if memory allocation fails

1  if size is 0
   1  Call insert_front(data) and return its result
2  else
   1  try
      1  Create a new node with the given data and assign it to new_node
      2  Set temp to the head node
      3  While temp's next pointer is not nullptr, set temp to temp's next
node
      4  Set the next pointer of temp to point to new_node, making it the last
node
      5  Set the next pointer of new_node to nullptr
      6  Increment the size of the linked list
      7  Return true to indicate successful insertion
   2  catch std::bad_alloc e
      1  Output "Unable to allocate more memory"
      2  Return false to indicate unsuccessful insertion due to memory
allocation
end insert_back
```

ALGORITHM 4-2-3 Delete Front

```
Algorithm delete_front()

This algorithm deletes the node at the front of the linked list.

    Pre    None
    Post   The node at the front of the linked list is deleted
    Return true if deletion is successful; false if the linked list is empty

1  if head is nullptr
   1  Return false to indicate unsuccessful deletion due to an empty linked
list
2  else
   1  Set temp to the head node
   2  Update the head pointer to point to temp's next node, removing the front
node
   3  Delete temp to free its memory
   4  Decrement the size of the linked list
   5  Return true to indicate successful deletion
end delete_front
```

```
ALGORITHM 4-2-4 Delete Back

Algorithm delete_back()

This algorithm deletes the node at the back of the linked list.

    Pre    None
    Post   The node at the back of the linked list is deleted
    Return true if deletion is successful; false if the linked list is empty

1  if head is nullptr
   1  Return false to indicate unsuccessful deletion due to an empty linked
list
2  else
   1  Set temp to the head node
   2  While temp's next node's next pointer is not nullptr, set temp to temp's
next node
   3  Delete temp's next node to free its memory, which is the last node
   4  Set temp's next pointer to nullptr, making it the new last node
   5  Decrement the size of the linked list
   6  Return true to indicate successful deletion
end delete_back
```

```
ALGORITHM 4-2-5 Delete Key

Algorithm delete_key(key)

This algorithm deletes the node with the given key from the linked list.

    Pre    key is the value to be deleted
    Post   The node with the key is deleted from the linked list
```

```
    Return true if deletion is successful; false if the key is not found

1  if size is 0
   1  Return false to indicate unsuccessful deletion due to an empty linked
list
2  else if size is 1
   1  if head's data is equal to key
      1  Delete head to free its memory
      2  Set head to nullptr
      3  Decrement the size of the linked list
      4  Return true to indicate successful deletion
   2  end if
   3  end if
3  else if size is 2
   1  if head's data is equal to key
      1  Set temp to head
      2  Set head to head's next node
      3  Delete temp to free its memory
      4  Decrement the size of the linked list
      5  Return true to indicate successful deletion
   2  else if head's next node's data is equal to key
      1  Delete head's next node to free its memory
      2  Set head's next pointer to nullptr
      3  Decrement the size of the linked list
      4  Return true to indicate successful deletion
   3  end if
   4  end if
4  else
   1  for temp from head to temp's next node is not nullptr, incrementing temp
      1  if temp's next node is not nullptr and temp's next node's data is
equal to key
         1  Set target to temp's next node
         2  Set temp's next pointer to target's next pointer, skipping the
target node
         3  Delete target to free its memory
         4  Decrement the size of the linked list
         5  Return true to indicate successful deletion
      2  end if
   2  end for
5  Return false to indicate unsuccessful deletion (key not found)
end delete_key
```

```
ALGORITHM 4-2-6 Search

Algorithm search(key)

This algorithm searches for a node with the given key in the linked list.

    Pre    key is the key to search for
    Post   Node with the key is found and returned; false if not found
    Return true if a node with the key is found; false if not found

1  for temp from head to temp is not nullptr, incrementing temp
```

```
   1  if temp's data is equal to key
      1  Return true to indicate successful search
   2  end if
2  end for
3  Return false to indicate unsuccessful search (key not found)
end search
```

ALGORITHM 4-2-7 Print

```
Algorithm print()

This algorithm prints the elements of the linked list.

1 Output "["
2 For i from head to i is not nullptr, incrementing i
  1 Output " " concatenated with i's data
  2 If i's next is not nullptr
    1 Output ","
3 Output "]" concatenated with a new line

end print
```

ALGORITHM 4-2-8 Begin

```
Algorithm begin()

This algorithm returns a pointer to the first node of the linked list.

  Return Pointer to the head node of the linked list

1 Return head

end begin
```

ALGORITHM 4-2-9 End

```
Algorithm end()

This algorithm returns a pointer to the last node of the linked list.

  Return Pointer to the last node of the linked list

1 Set temp as the head node
2 While temp's next pointer is not nullptr:
  1 Update temp to the next node
3 Return temp

end end
```

ALGORITHM 4-2-10 Get Size

```
Algorithm get_size()

This algorithm returns the current size of the linked list.

  Return The current size of the linked list

1 Return size

end get_size
```

```
ALGORITHM 4-2-11 Index Operator

Algorithm operator[](index)

This algorithm returns a reference to the element at the specified index in
the linked list.

  Pre    index is the index of the element to be accessed
  Post   The element at the specified index is returned by reference
  Return Reference to the element at the specified index

1 if index is less than 0 or index is greater than or equal to size
  1 Throw an out_of_range exception with the message "Index out of bounds"
2 Create a pointer current and assign it the value of head
3 Loop from 0 to index - 1
  1 Update current to point to the next node
4 Return the data of the node pointed to by current by reference

end operator[]
```

4.3 Word Class

The word class is an integral component of a larger project aimed at creating a comprehensive word dictionary application. This class encapsulates the concept of a dictionary word, storing its term and corresponding definition. The class offers a variety of functionalities for managing and manipulating word objects, including setting and retrieving terms and definitions, comparing words for equality and order, and transforming words to lowercase.

Class Structure

The word class is composed of member variables to hold the term and definition of a word. The key functionalities and features of the class are as follows:

Constructors and Destructor: The class provides a default constructor that initializes the term and definition to empty strings. Another constructor allows initializing the term and definition during object creation. The destructor is responsible for proper memory management.

Accessors and Mutators: The class offers methods to access and modify both the term and definition. These methods ensure proper encapsulation of data, allowing controlled interaction with the class's internal attributes.

Display Method: A `display()` method presents the term and definition of a word in a human-readable format. It formats and prints the term followed by its definition.

Transform to Lowercase: The `operator()` method transforms the term of a word to lowercase. This functionality is essential for uniform comparison of words regardless of their case.

Comparison Operators: The class provides comparison operators (`==, !=, <, >`) to facilitate comparisons between word objects based on their terms. These operators are crucial for sorting and searching operations.

Type Conversion Operator: The operator `std::string()` allows implicit conversion of a word object to its term as a string. This feature simplifies the process of using word objects in string contexts.

Friend Function: A friend function `operator<<` is defined to enable easy printing of word objects. It overloads the stream insertion operator to allow straightforward output of word terms.

Design Considerations

Encapsulation: The class encapsulates the term and definition attributes, providing controlled access through accessor and mutator methods. This encapsulation ensures data integrity and allows future changes to the internal representation without affecting client code.

String Transformation: The use of `std::transform` in the constructor and `operator()` highlights the class's flexibility. It enables consistent and case-insensitive comparisons between words.

Comparison Operators: By implementing comparison operators, the class supports sorting and searching operations in data structures like binary search trees. These operators are vital for efficient word management.

4.4 Utility

The `util` namespace is an integral part of the word dictionary application, offering a collection of utility functions and tools to streamline user interactions, data management, and text processing. This namespace encapsulates various functionalities required for input handling, data storage, formatting, and more.

The util namespace is composed of a set of utility functions, each serving a specific purpose within the application:

String Splitting: The `str_split()` function splits a string based on a delimiter and returns a list of resulting tokens. This functionality is essential for parsing and processing data from files.

Loading Database: The `load_database()` function loads word entries from a file into the binary search tree (BST). It utilizes the `str_split()` function to extract term and definition pairs from each line of the file.

Storing Database: The `store_database()` function stores the word entries from the BST back into a file. It employs a helper function, `store_database_helper()`, to perform a pre-order traversal of the BST and write each word's information to the file.

Saving Changes: The `save_changes()` function prompts the user to confirm whether they want to save changes made to the BST. If confirmed, it invokes the `store_database()` function to persist the modifications.

Input Handling: The namespace provides functions to handle user input, such as `input_word()` for entering valid alphabetic words and `input_sentence()` for capturing sentences.

String Transformation: The `to_lowercase()` function converts a string to lowercase, ensuring consistent and case-insensitive processing of input.

Clearing Screen: The `clear_screen()` function clears the terminal screen to improve user interface readability.

Confirmation Check: The `confirmation_check()` function prompts the user for a yes/no confirmation and returns a boolean based on the response.

```
ALGORITHM 4-4-1 String Split

Algorithm str_split(line, delimiter)

This algorithm splits a given line into tokens using the specified delimiter
and returns a list of tokens.

  Pre    line is a pointer to the input line
         delimiter is the character used for splitting
  Post   The line is split into tokens and a list of tokens is returned
  Return List of tokens

1 Create a new istringstream named line_stream and initialize it with the value
of line
2 Initialize an empty string named token
3 Create a new empty list of strings named token_list
4 Repeat until end of line_stream is reached:
  1 Read a token from line_stream until the delimiter is encountered, and store
it in the token variable
  2 Insert the token into the token_list using the insert_back operation
5 Return the token_list

end str_split
```

```
ALGORITHM 4-4-2 Load Database

Algorithm load_database(WORD_TREE)

This algorithm loads word data from a file and populates a binary search tree
with the data.

  Pre    WORD_TREE is a pointer to the binary search tree
  Post   Binary search tree is populated with word data from the file
  Return None

1 Set filename to "../data/database"
```

```
2 Open the file with input_file using filename
3 If input_file fails to open:
  1 Print "Error loading database from file!"
  2 Call wait_for_input function
4 Else:
  1 Create an empty string named line
  2 Set delimeter to '|'
  3 Repeat until the end of input_file is reached:
    1 Read a line from input_file into the line variable
    2 Create a new list of strings named token_list using the str_split function
with line and delimeter
    3 Create a new word object named new_word
    4 If the size of token_list is 1:
      1 Call new_word's operator() with token_list[0] and an empty string
    5 Else if the size of token_list is greater than or equal to 2:
      1 Call new_word's operator() with token_list[0] and token_list[1]
    6 Else:
      1 Break the loop
    7 Insert new_word into the binary search tree pointed by WORD_TREE using
its insert method
  4 Close input_file
5 End If

end load_database
```

ALGORITHM 4-4-3 Store Database Helper

```
Algorithm store_database_helper(node, output_file_ptr)

This algorithm performs a pre-order traversal of a binary search tree and writes
the term and definition of each word to an output file.

  Pre    node is a pointer to the current node in the binary search tree
         output_file_ptr is a pointer to the output file stream
  Post   Binary search tree nodes are written to the output file in pre-order
         traversal
  Return None

1 If node is nullptr, return
2 Write node's term followed by "|" and then node's definition to the output
file using output_file_ptr
3 Call store_database_helper with node's left child and output_file_ptr
4 Call store_database_helper with node's right child and output_file_ptr

end store_database_helper
```

ALGORITHM 4-4-4 Store Database

```
Algorithm store_database(WORD_TREE)

This algorithm stores the contents of a binary search tree containing words into
an output file.

  Pre    WORD_TREE is a pointer to the binary search tree
```

```
  Post   Binary search tree's contents are written to the output file
  Return None

1 Set filename to "../data/database"
2 Create an output_file with filename
3 If output_file is not successfully opened:
4   Print "Error loading database from file!"
5   Call wait_for_input() to pause and wait for user input
6 Else:
7    Call store_database_helper with the root of WORD_TREE and the address of
output_file
8   Close output_file

end store_database
```

ALGORITHM 4-4-5 Save Changes

```
Algorithm save_changes(WORD_TREE)

This algorithm prompts the user to save changes made to a binary search tree
and writes the changes to a file if confirmed.

  Pre    WORD_TREE is a pointer to the binary search tree
  Post   Changes made to the binary search tree are saved to a file based on
         user's input
  Return None

1 Set message to "Do you want to save the changes to the Binary Search Tree
(Yes/No): "
2 Call confirmation_check with the address of message and store the result in
confirmation
3 If confirmation is true:
4   Call util::store_database with WORD_TREE to save the changes to the file
5   Print "Changes were saved."
6 Else:
7   Print "Changes were not saved."

end save_changes
```

ALGORITHM 4-4-6 Clear Input Buffer

```
Algorithm clear_input_buffer()

This algorithm clears the input buffer to ensure that any remaining characters
from previous inputs are removed.

  Pre    None
  Post   Input buffer is cleared
  Return None

1 Clear the error flags of the input stream using std::cin.clear()
2 Ignore all characters in the input buffer until a newline character using
std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n')
```

```
end clear_input_buffer
```

ALGORITHM 4-4-7 Wait for Input

```
Algorithm wait_for_input()

This algorithm displays a message and waits for the user to press the Enter key.

  Pre    None
  Post   Waits for user to press the Enter key
  Return None

1 Create an empty string variable named "line"
2 Display an empty line to add some space
3 Display a line of dashes as a separator
4 Display an empty line to add some space
5 Read a line of text from the input stream using std::getline and store it in
the "line" variable
6 Wait for the user to press the Enter key

end wait_for_input
```

ALGORITHM 4-4-8 To Lowercase

```
Algorithm to_lowercase(str)

This algorithm converts the characters in the input string to lowercase.

  Pre    str is a pointer to the input string
  Post   The characters in the input string are converted to lowercase
  Return A new string containing the lowercase version of the input string

1 Create a new string variable named "lowercase" and initialize it with the
dereferenced value of "str"
2 For each character "c" in the "lowercase" string:
  1 Apply a lambda function that converts the character to lowercase and assign
the result back to "c"
3 Return the "lowercase" string

end to_lowercase
```

ALGORITHM 4-4-9 Clear Screen

```
Algorithm clear_screen()

This algorithm clears the console screen. The method used depends on the
operating system.

1 Check if the operating system is Windows (detected using _WIN32 macro)
  1 If true, execute the command "cls" using the system function to clear the
screen
  2 If false, execute the command "clear" using the system function to clear
the screen on Linux
```

```
2 End if

end clear_screen
```

ALGORITHM 4-4-10 Word Is Alpha

```
Algorithm word_is_alpha(word)

This algorithm checks if all characters in a given word are alphabetic.

  Pre    word is the input word to be checked
  Post   True is returned if all characters are alphabetic, false otherwise
  Return True if all characters are alphabetic, false otherwise

1 For each character c in the input word
  1 If c is not an alphabetic character
    1 Return false
  2 End if
2 End for
3 Return true

end word_is_alpha
```

ALGORITHM 4-4-11 Input Word

```
Algorithm input_word(message)

This algorithm takes user input for a word and ensures that the input consists
of only alphabetic characters.

  Pre    message is an optional message to be displayed as a prompt
  Post   User input word is obtained and returned after ensuring it consists of
         alphabetic characters
  Return User input word consisting of alphabetic characters

1 Create an empty string variable str
2 Repeat the following steps until a valid word is entered:
  1 If message is not null
    1 Display message as a prompt
  2 Else
    1 Display "Enter a word: " as a prompt
  3 Take user input and store it in str
  4 Clear the input buffer using clear_input_buffer()
  5 If word_is_alpha(&str) returns false
    1 Display an error message indicating that the word should only contain
alphabetic characters
    2 Display "Please try again!"
    3 Wait for user input using wait_for_input()
  6 Else
    1 Exit the loop
3 Convert str to lowercase using the to_lowercase function
4 Return str

end input_word
```

```
ALGORITHM 4-4-12 Input Sentence

Algorithm input_sentence()

This algorithm takes user input for a sentence and returns the entered sentence.

  Post   User input sentence is obtained and returned
  Return User input sentence

1 Create an empty string variable sentence
2 Read a line of input from the user and store it in the sentence variable using
std::getline(std::cin, sentence)
3 Return sentence

end input_sentence
```

```
ALGORITHM 4-4-13 Confirmation Check

Algorithm confirmation_check(message)

This algorithm displays a message to the user and expects a confirmation input
('yes' or 'no').
It returns true if the user confirms with 'yes' and false if the user answers
'no'.

  Pre    message is an optional message to display
  Post   User confirmation is obtained and returned as true or false
  Return true if user confirms with 'yes', false if user answers with 'no'

1 Create an empty string variable confirmation
2 Repeat the following steps until a valid confirmation is received:
  1 Call util::input_word(message) and store the result in the confirmation
variable
  2 If confirmation is equal to "y" or "yes":
    1 Return true
  3 Else if confirmation is equal to "n" or "no":
    1 Return false
  4 Else:
    1 Display "Please answer with either 'yes' or 'no'!"

end confirmation_check
```

4.5 The ‘main()’

The `main.cpp` file is the heart of the Word Dictionary application, responsible for managing the user interface and interactions. This file includes various functions to facilitate word operations like adding, searching, updating, deleting, and displaying words within a Binary Search Tree (BST). The main loop of the program continually presents a user menu, captures user choices, and performs corresponding actions. Here is an overview of the key components and functionalities:

Initialization and Data Loading:

- The program starts by initializing a BST named `WORD_TREE` to store word objects.
- It loads existing word data from a file using the `util::load_database()` function.

User Interaction and Menu:

- The main loop continually presents a user menu using the `print_menu()` function.
- The choice function captures and validates the user's menu choice.

Adding a Word:

- The `add_word()` function allows users to input a new word and its definition.
- It uses the `add_word_helper()` function to handle the input process and word insertion into the BST.

Searching for a Word:

- The `search_word()` function enables users to search for a word within the BST.
- It uses the `print_suggestions()` function to provide suggestions if the word is not found.

Deleting a Word:

- The `delete_word()` function lets users delete a word from the BST.
- It checks for the existence of the word and confirms deletion using `util::confirmation_check()`.

Updating a Word:

- The `update_word()` function allows users to update an existing word's definition.
- It first removes the word and then adds the updated version using `add_word_helper()`.

Displaying the Word Tree:

The `print_tree()` function displays the entire contents of the BST, giving users an overview of stored words.

Exiting and Saving Changes:

Users can choose to exit the program and save any changes to the word database.
The `util::save_changes()` function ensures that modifications are written back to the file.

The program is structured to provide a user-friendly interface for managing a collection of words. It uses the utility functions defined in `utility.h` and `utility.cpp` to handle tasks like input validation, clearing the screen, and managing the file database. The word operations are performed on the `WORD_TREE` BST, which is dynamically updated based on user interactions. The program allows users to maintain a personalized word dictionary efficiently.

```
ALGORITHM 4-5-1 Find Closest Node

Algorithm find_closest_node(node, letter, index)
```

```
This algorithm finds the closest node in a binary search tree based on a given
letter and its index.

Pre:
   node is a pointer to the current node being evaluated
   letter is the character to match in the word
   index is the index of the character in the word

Post:
   The closest node in the binary search tree is returned, or nullptr if no
   matching node is found

Return:
   A pointer to the closest node in the binary search tree or nullptr if no
   matching node is found

1 Create a try-catch block to handle exceptions:
   1 Try the following block of code
   2 Catch any exceptions of type std::exception and store the exception message
     in a variable e
     1 Display "Exception occurred : " concatenated with e's message
     2 Return nullptr

2 If node is nullptr:
   1 Return nullptr

3 Create a string variable current_word and set it to node's data

4 If (index + 1) > current_word's size or letter > current_word at index:
   1 Call find_closest_node(node's right child, letter, index) and return the
     result

5 Else if letter < current_word at index:
   1 Call find_closest_node(node's left child, letter, index) and return the
     result

6 Else:
   1 Return node

end algorithm find_closest_node
```

ALGORITHM 4-5-2 Print Suggestions Helper

```
Algorithm print_suggestions_helper(subtree, suggestion_list, matching_substr)

This algorithm recursively traverses a subtree of a binary search tree and
populates a suggestion list based on matching substrings.

Pre:
   subtree is a pointer to the current subtree being traversed
   suggestion_list is a pointer to a list to store suggestions
   matching_substr is the substring to match against the word's beginning
```

```
Post:
   The suggestion_list is populated with matching suggestions

1 Create a try-catch block to handle exceptions:
   1 Try the following block of code
   2 Catch any exceptions of type std::exception and store the exception message
     in a variable e
     1 Display "Exception occurred : " concatenated with e's message

2 If subtree is nullptr:
   1 Return

3 Call print_suggestions_helper(subtree's left child, suggestion_list,
  matching_substr)

4 Call print_suggestions_helper(subtree's right child, suggestion_list,
  matching_substr)

5 If the first matching_substr characters of subtree's data's term match
  matching_substr:
   1 Call suggestion_list's insert_front method with subtree's data's term

end algorithm print_suggestions_helper
```

ALGORITHM 4-5-3 Print Suggestions

```
Algorithm print_suggestions(search)

This algorithm displays suggestions for a given search term based on the closest
matches found in a binary search tree.

Pre:
   search is a pointer to the search term

Post:
   Suggestions are displayed to the user based on the closest matches found in
   the binary search tree.

Return:
   None

1 Call util::clear_screen()
2 Create an empty string variable target_str and set it to the value of search
3 Create an empty string variable matching_substr
4 Create a pointer current_node and set it to the root of WORD_TREE
5 Create a pointer temp_node and set it to nullptr
6 Display "                                        Suggestions"
7 Display "————————————————————————————————————————————————"
8 Display "The word '" concatenated with the value of search, followed by "'
  was not found!" and an empty line
9 For each character (letter) at index i in target_str:
   1 Set letter to target_str[i]
   2 Set temp_node to the result of find_closest_node(current_node, letter, i)
```

```
   3 If temp_node is nullptr and i is equal to 0:
      1 Return
   4 Else if temp_node is nullptr:
      1 Break the loop
   5 Else:
      1 Set current_node to temp_node
      2 Append letter to matching_substr
10 Set suggestion_count to 5
11 Create an empty list suggestion_list
12 Call print_suggestions_helper(current_node, &suggestion_list,
   matching_substr)
13 If the size of suggestion_list is less than suggestion_count:
   1 Set suggestion_count to the size of suggestion_list
14 If the size of suggestion_list is not equal to 0:
   1 Display "Did you mean:"
   2 Display "[ "
   3 For each index from 0 to suggestion_count - 1:
      1 Display suggestion_list[index]
      2 If index is not equal to suggestion_count - 1:
         1 Display ", "
   4 Display " ]"

end algorithm print_suggestions
```

ALGORITHM 4-5-4 Add Word Helper

```
Algorithm add_word_helper(message)

This algorithm assists in adding a new word and its definition to the binary
search tree.

Pre:
   message is an optional message to be displayed as a prompt

Post:
   A new word and its definition are added to the binary search tree if not
   already present.

Return:
   None

1 Set terminology to the result of util::input_word(message)
2 Display "Enter the definition of the word: "
3 Set definition to the result of util::input_sentence()
4 Create a word object named new_word using terminology and definition
5 If the result of WORD_TREE.search(new_word) is equal to nullptr:
   1 If the result of WORD_TREE.insert(new_word) is true:
      1 Display an empty line
      2 Display "Added the word '" concatenated with terminology, followed by
        "' to the Binary Search Tree."
      3 Call util::wait_for_input()
      4 Return
   2 Else:
      1 Display "Error! Could not add the word '" concatenated with terminology,
```

```
         followed by "'."
6 Else:
   1 Display "Could not add '" concatenated with terminology, followed by "'!
     It is already in the BST!"
7 Call util::wait_for_input()

end algorithm add_word_helper
```

ALGORITHM 4-5-5 Add Word

```
Algorithm add_word()

This algorithm provides a user interface for adding a new word and its definition
to the binary search tree.

Pre:
    None

Post:
    A new word and its definition are added to the binary search tree if not
    already present.

Return:
    None

1 Call util::clear_screen()
2 Display "                                          Add Word"
3 Display "──────────────────────────────────────────────────"
4 Set message to "Enter the word you want to add: "
5 Call add_word_helper with message as the argument

end algorithm add_word
```

ALGORITHM 4-5-6 Search Word

```
Algorithm search_word()

This algorithm provides a user interface for searching for a word in the binary
search tree and displaying the search result.

Pre:
    None

Post:
    Search result for the entered word is displayed to the user.

Return:
    None

1 Call util::clear_screen()
2 Create an empty string variable target_str
3 Get the root node of the binary search tree and store it in root
4 Display "                                            Search"
5 Display "──────────────────────────────────────────────────"
```

```
6 Set message to "Enter the word you want to search: "
7 Take user input for the target word using util::input_word() with message as
  an argument and store it in target_str

8 Create a word object named 'target' with target_str as the term and an empty
  definition
9 Search for the node containing 'target' in the binary search tree and store
  the result in found_node
10 If root is not null:
  1 If found_node is null:
    1 Call print_suggestions with 'target' as the argument
    2 Call util::wait_for_input()
  2 Else:
    1 Display an empty line
    2 Display "Search result:"
    3 Display the term and definition stored in (*found_node).data
    4 Call util::wait_for_input()

end algorithm search_word
```

```
ALGORITHM 4-5-7 Delete Word

Algorithm delete_word()

This algorithm provides a user interface for deleting a word from the binary
search tree.

Pre:
    None

Post:
    The selected word is deleted from the binary search tree.

Return:
    None

1 Call util::clear_screen()
2 Create an empty string variable terminology
3 Display "                                        Delete Word"
4 Display "──────────────────────────────────────────────────"
5 Display "Word Tree:"
6 Call WORD_TREE.print()
7 Display an empty line
8 Set message to "Enter the word you want to delete: "
9 Take user input for the word to be deleted using util::input_word() with
  message as an argument and store it in terminology
10 Create a word object named 'new_word' with terminology as the term and an
   empty definition
11 If new_word is not found in WORD_TREE:
  1 Display "The word '" + terminology + "' was not found!"
  2 Call util::wait_for_input()
12 Else:
  1 Set confirmation_message to "Are you sure you want to delete the word '" +
```

```
    terminology + "' (Yes/No): "
  2 Call util::confirmation_check with confirmation_message as the argument and
    store the result in confirmation
  3 If confirmation is true:
    1 If WORD_TREE.remove(new_word) returns true:
      1 Display an empty line
      2 Display "The word '" + terminology + "' was deleted successfully!"
      3 Call util::wait_for_input()
      4 Return
    2 Else:
      1 Display "Error! The word '" + terminology + "' could not be deleted!"
      2 Call util::wait_for_input()

end algorithm delete_word
```

ALGORITHM 4-5-8 Update Word

```
Algorithm update_word()

This algorithm provides a user interface for updating a word in the binary
search tree.

Pre:
    None

Post:
    The selected word is updated in the binary search tree.

Return:
    None

1 Call util::clear_screen()
2 Create empty string variables target_terminology and terminology
3 Display "                                        Update Word"
4 Display "————————————————————————————————————————————————"
5 Display "Word Tree:"
6 Call WORD_TREE.print()
7 Display an empty line
8 Set message to "Enter the word you want to update: "
9 Take user input for the word to be updated using util::input_word() with
  message as an argument and store it in target_terminology

10 Create a word object named 'target_word' with target_terminology as the term
   and an empty definition
11 If target_word is not found in WORD_TREE:
  1 Display "Error! The word '" + target_terminology + "' was not found!"
  2 Call util::wait_for_input()
12 Else:
  1 Set word_to_delete to the data of the node found by searching for target_word
    in WORD_TREE
  2 Display an empty line
  3 Display "Updating the following word:"
  4 Call word_to_delete.display()
```

```
  5 Call WORD_TREE.remove(word_to_delete)
  6 Display an empty line
  7 Set message to "Enter the updated word: "
  8 Call add_word_helper() with message as an argument

end algorithm update_word
```

```
ALGORITHM 4-5-9 Display Tree

Algorithm print_tree()

This algorithm displays the contents of the binary search tree.

Pre:
    None

Post:
    The contents of the binary search tree are displayed.

Return:
    None

1 Call util::clear_screen()
2 Display "                                          Display"
3 Display "──────────────────────────────────────────────────"
4 Call WORD_TREE.print()
5 Call util::wait_for_input()

end algorithm print_tree
```

5 PROGRAM DEMONSTRATION

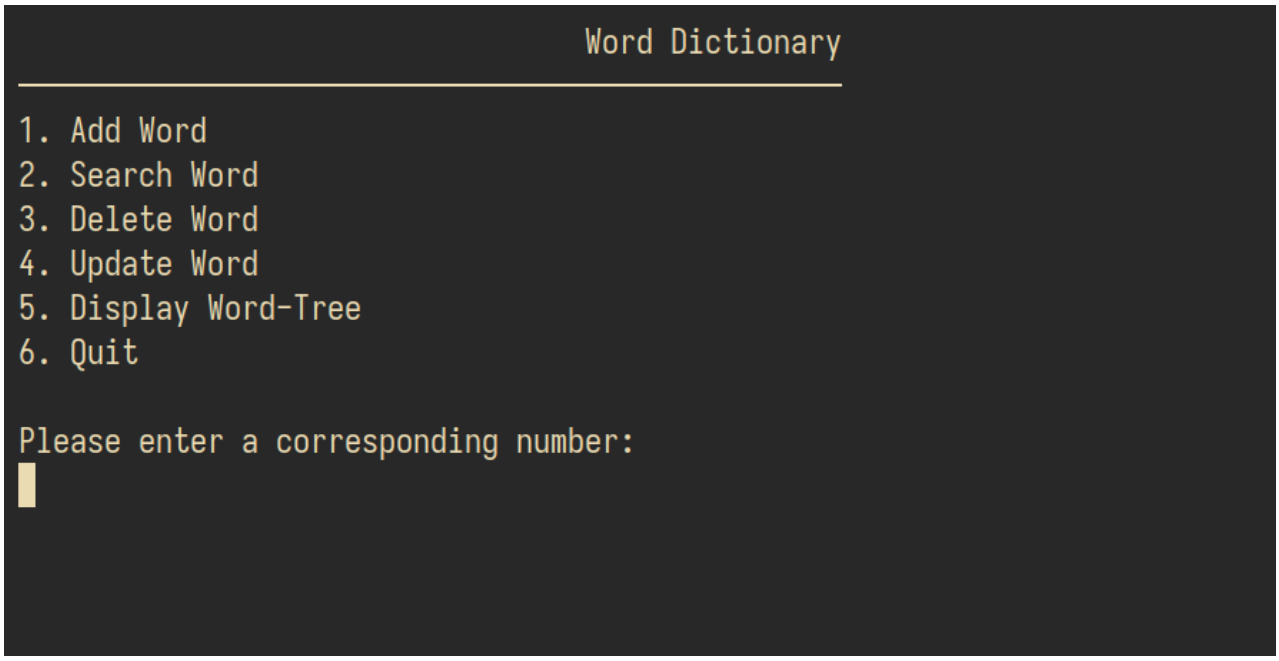

Figure 5-1: Program Menu

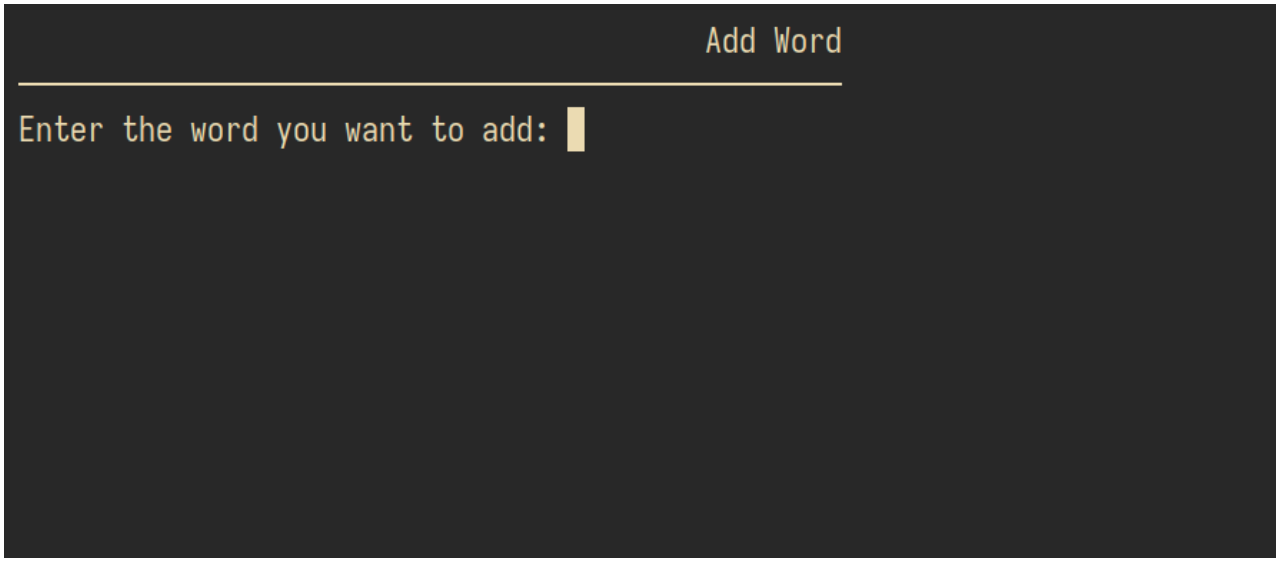

Figure 5-2: Add Word Menu

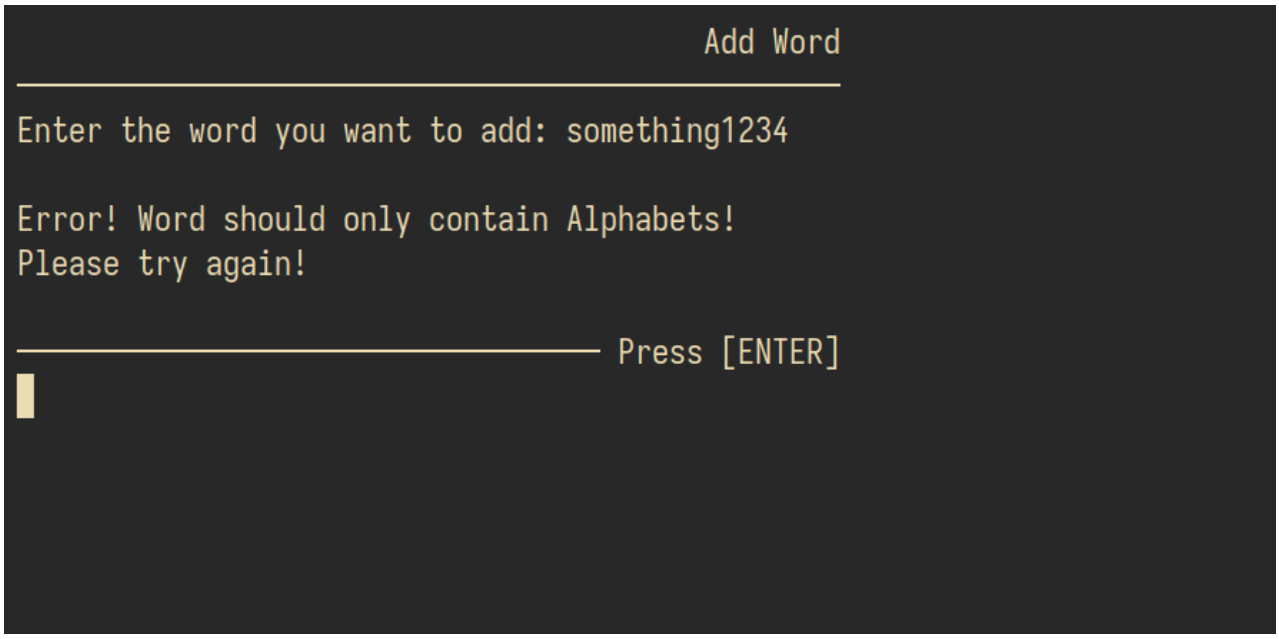

Figure 5-3: Invalid Terminology Handling

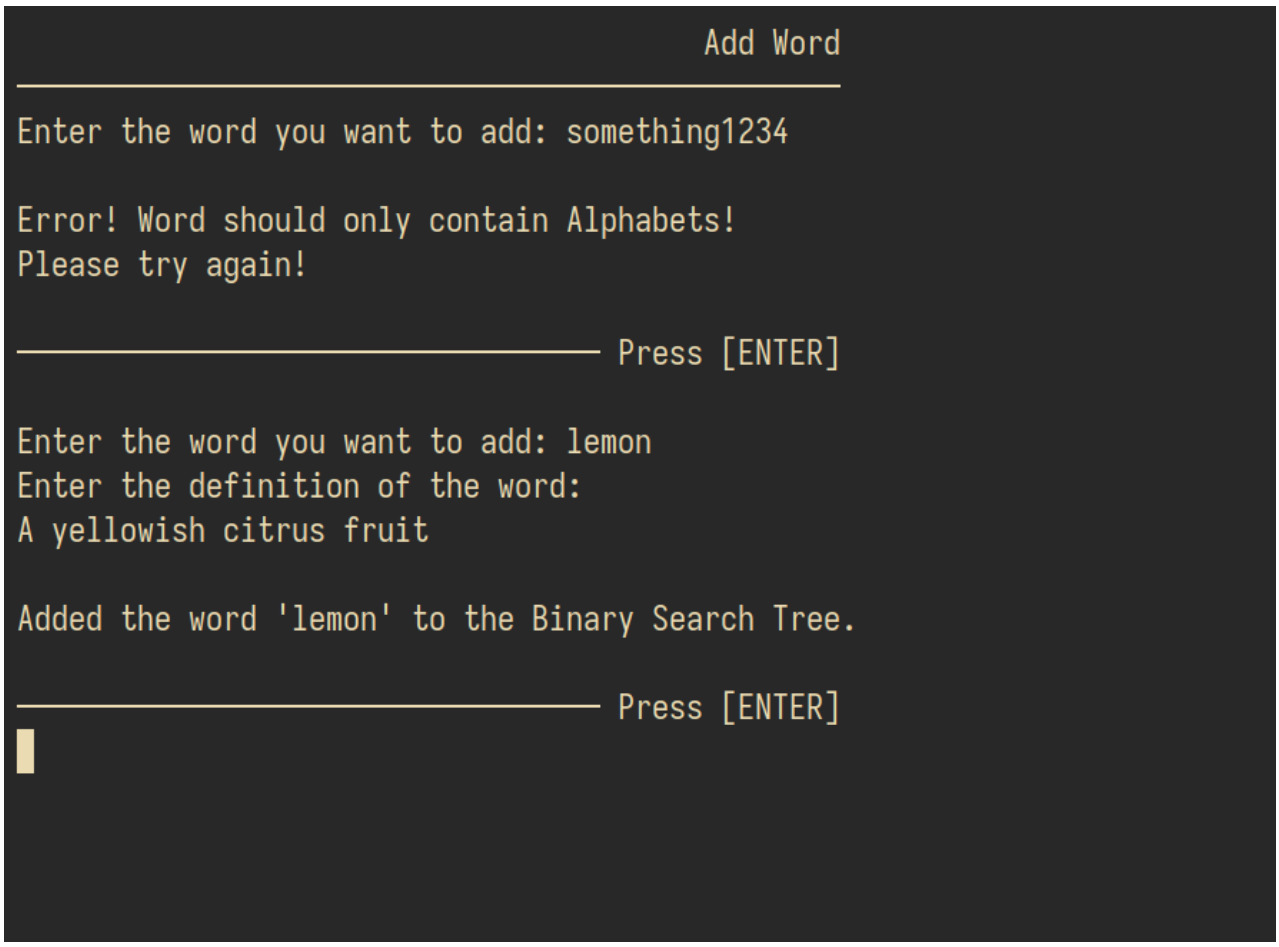

Figure 5-4: Successful Addition of Word

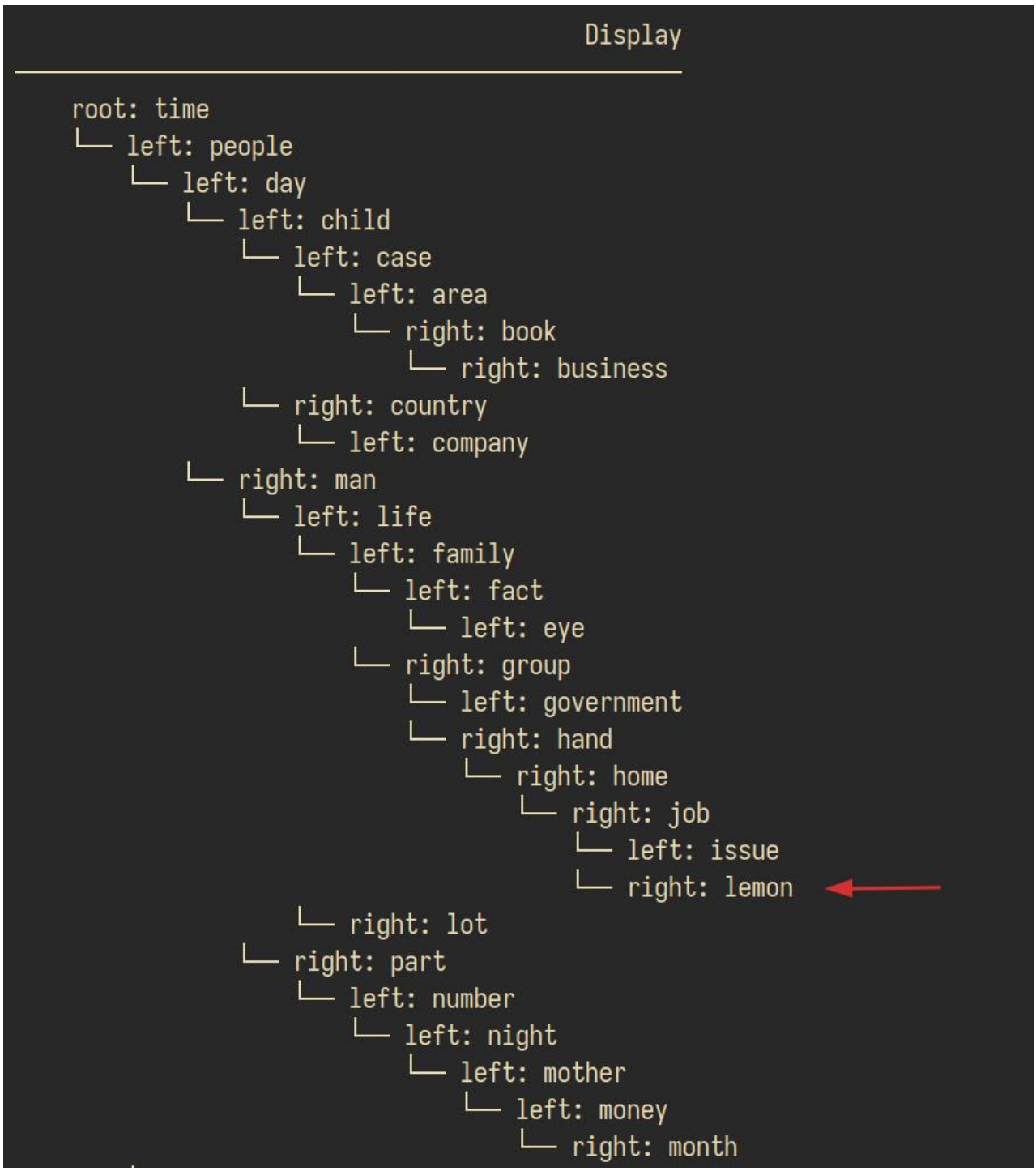

image continued…

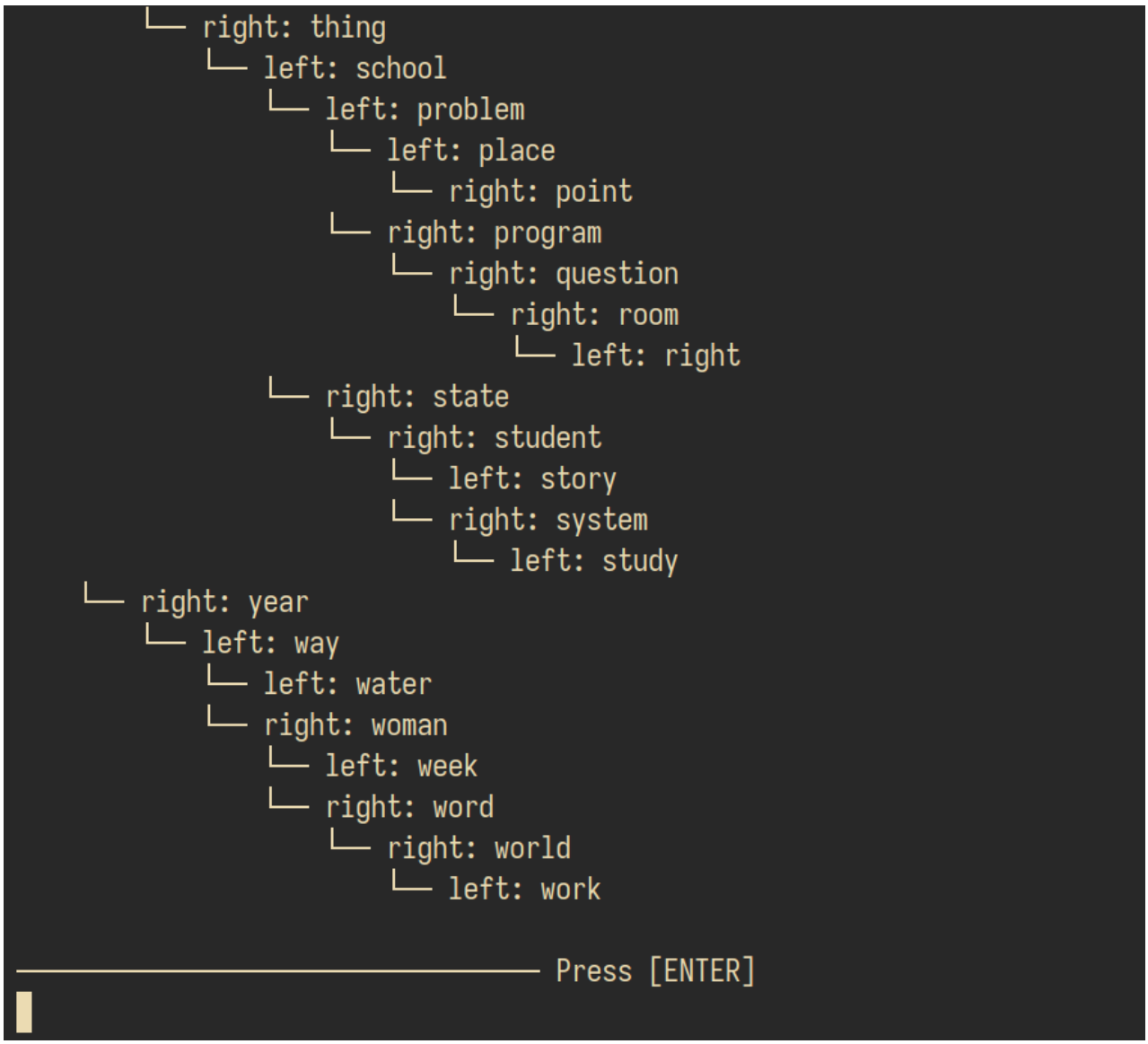

Figure 5-5: Tree Displayed with Newly Added Word

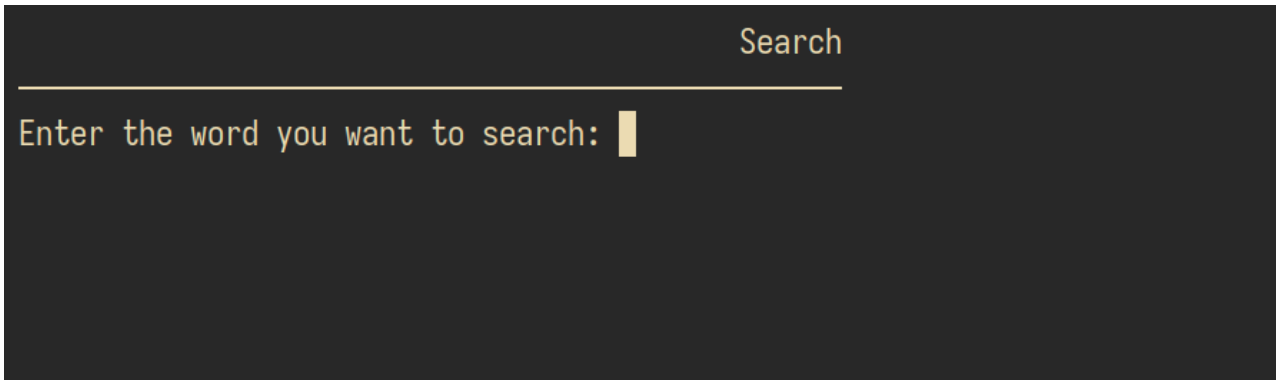

Figure 5-6 : Search Menu

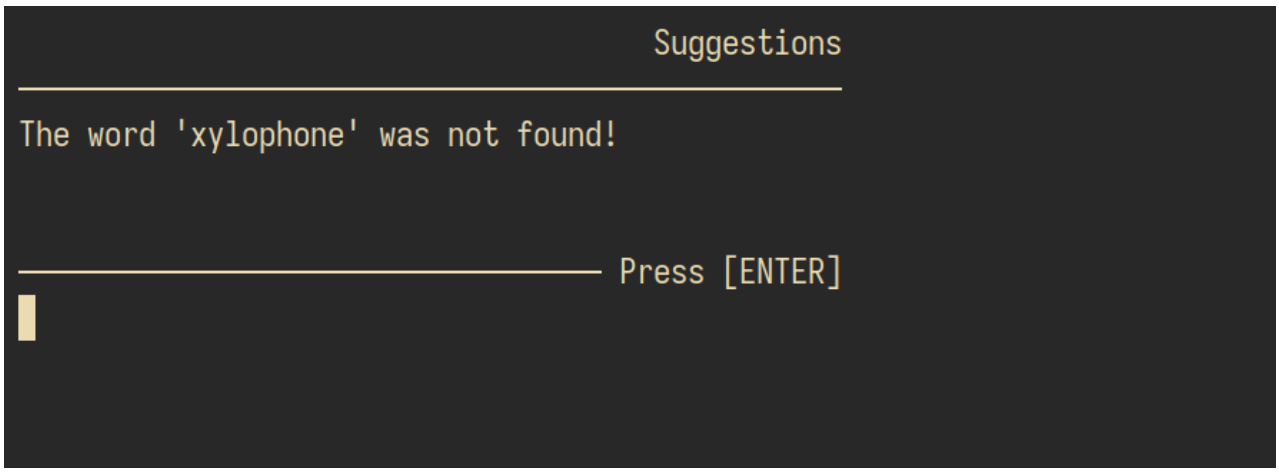

Figure 5-7: Search Result for Non-existing Word

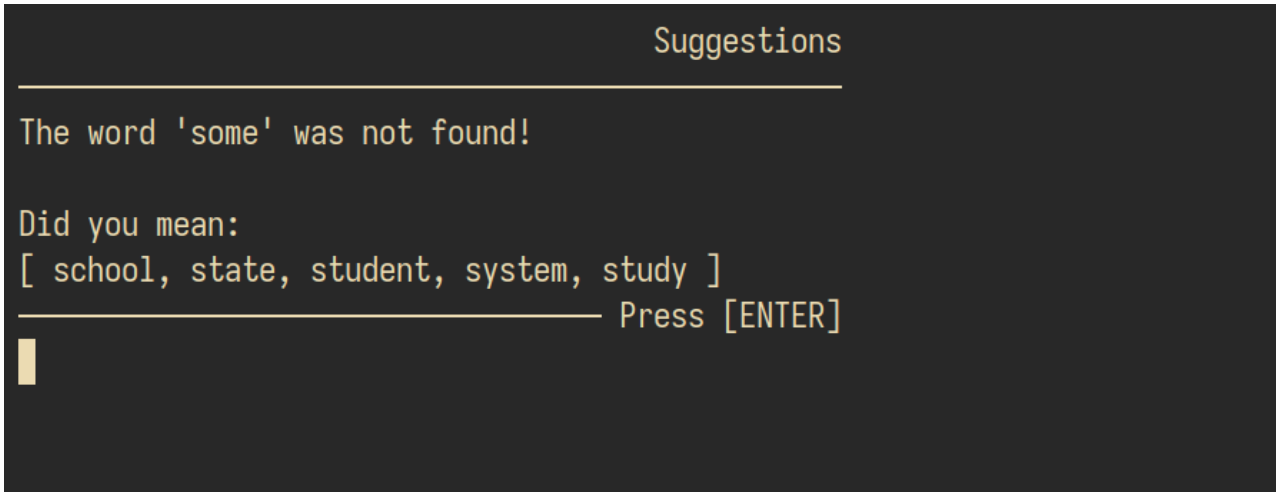

Figure 5-8: Closest Suggestion Result for A Search Keyword

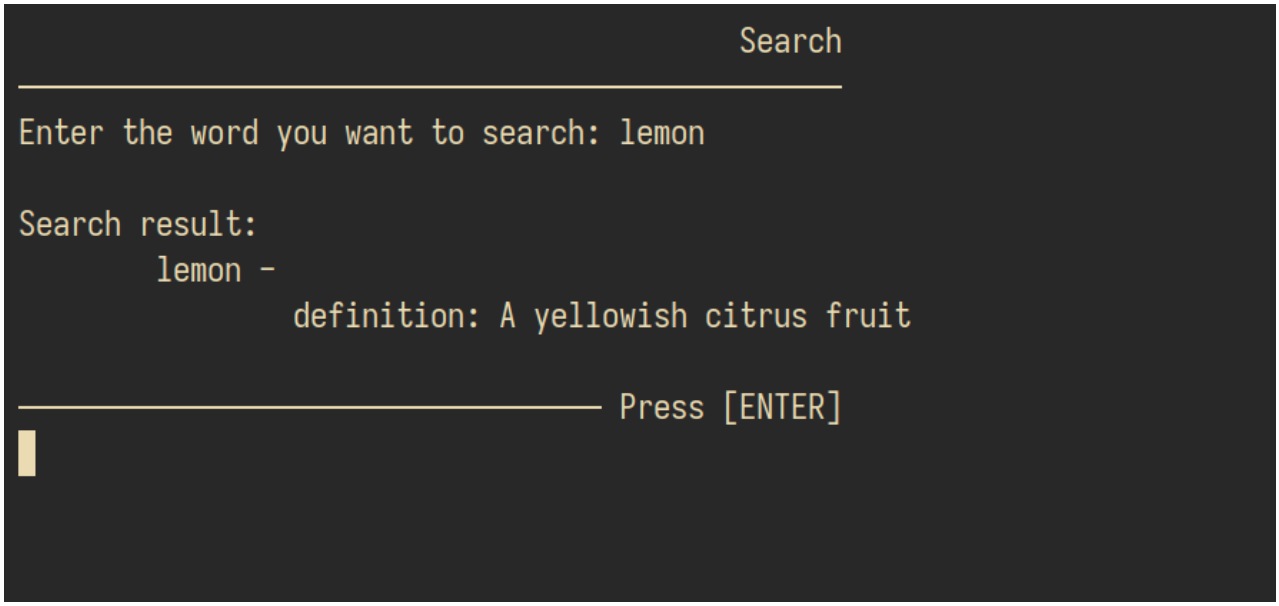

Figure 5-9: Search Result for an Existing Word

image continued…

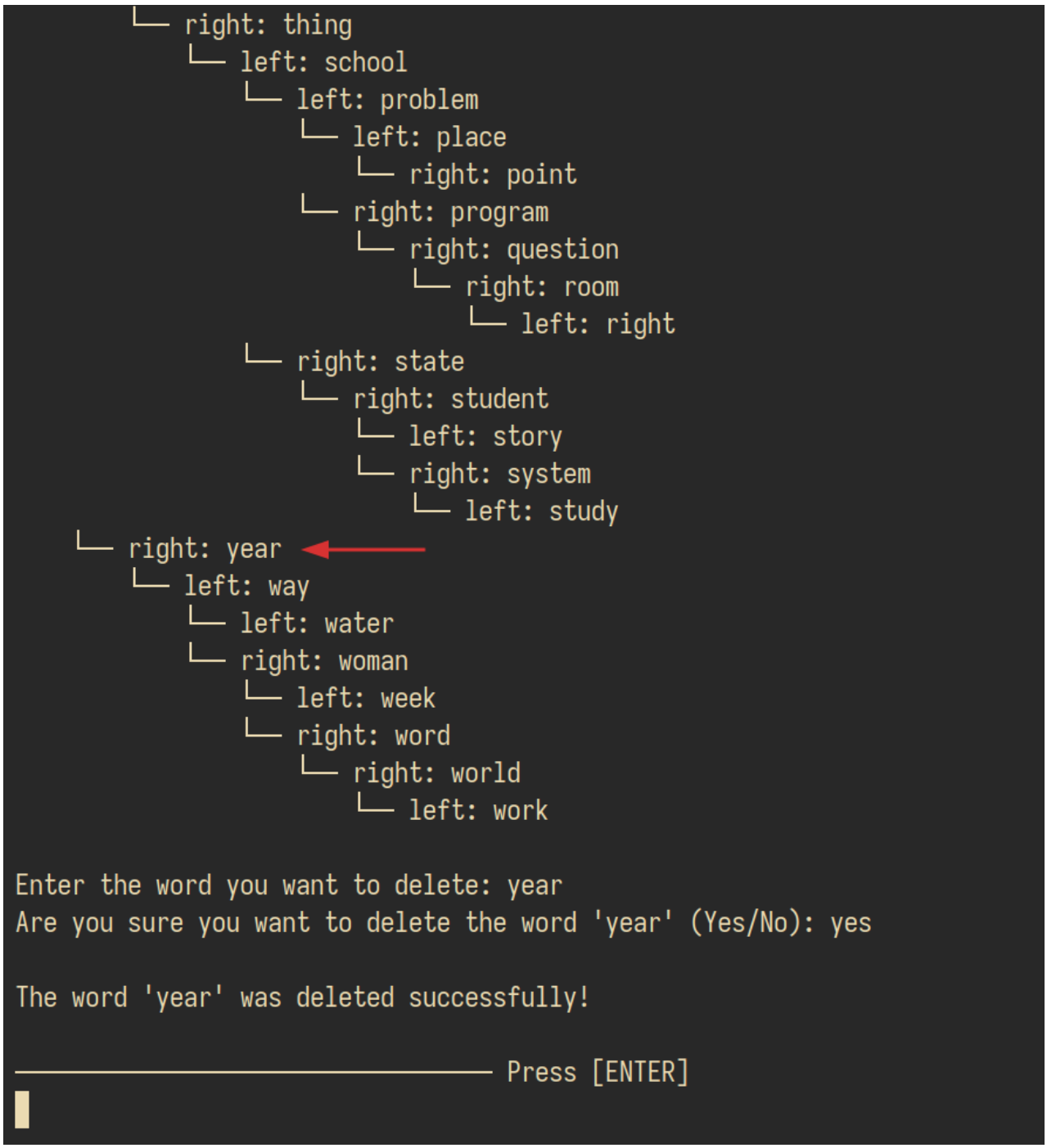

Figure 5-10: Delete Operation on Word BST

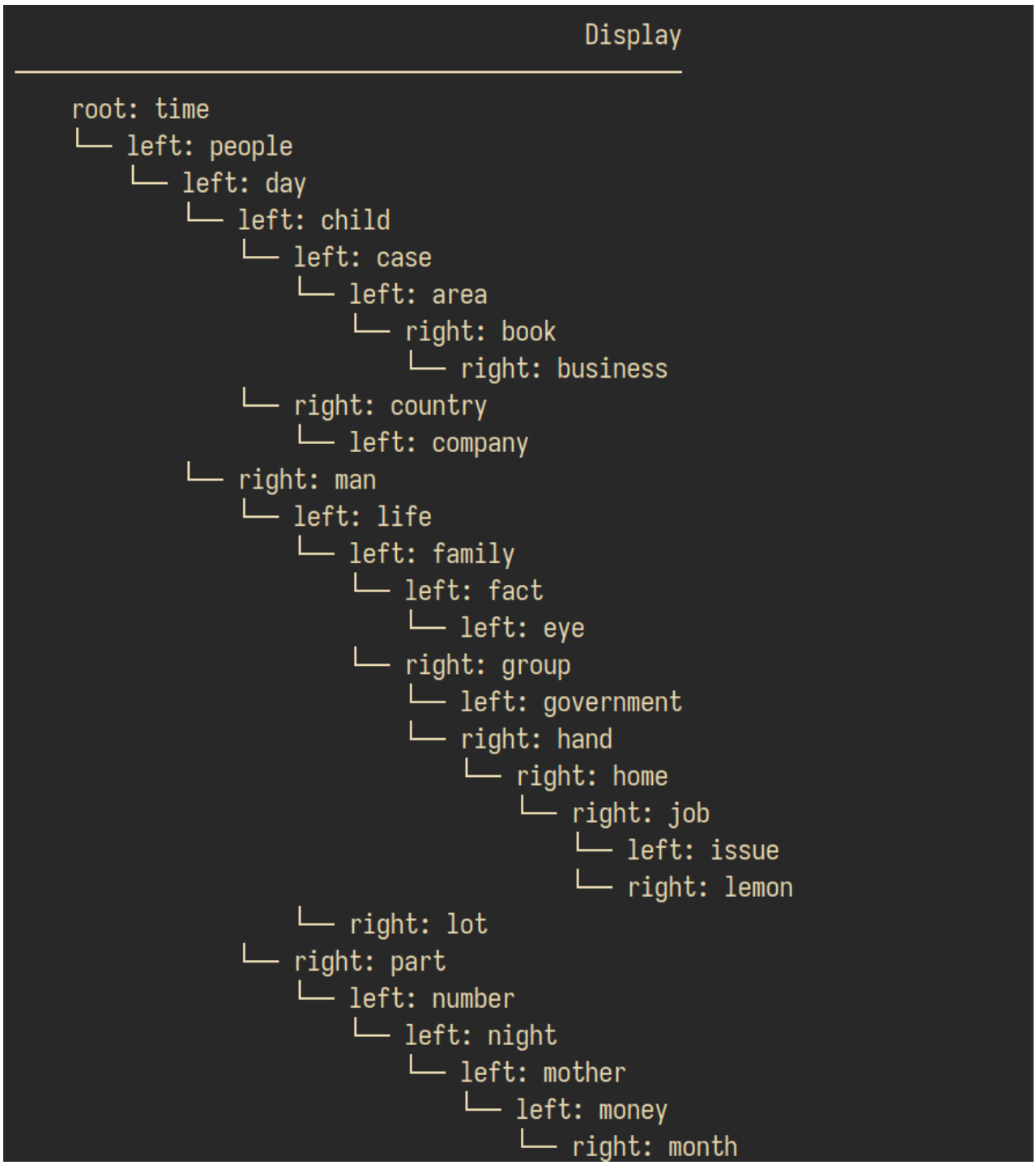

image continued…

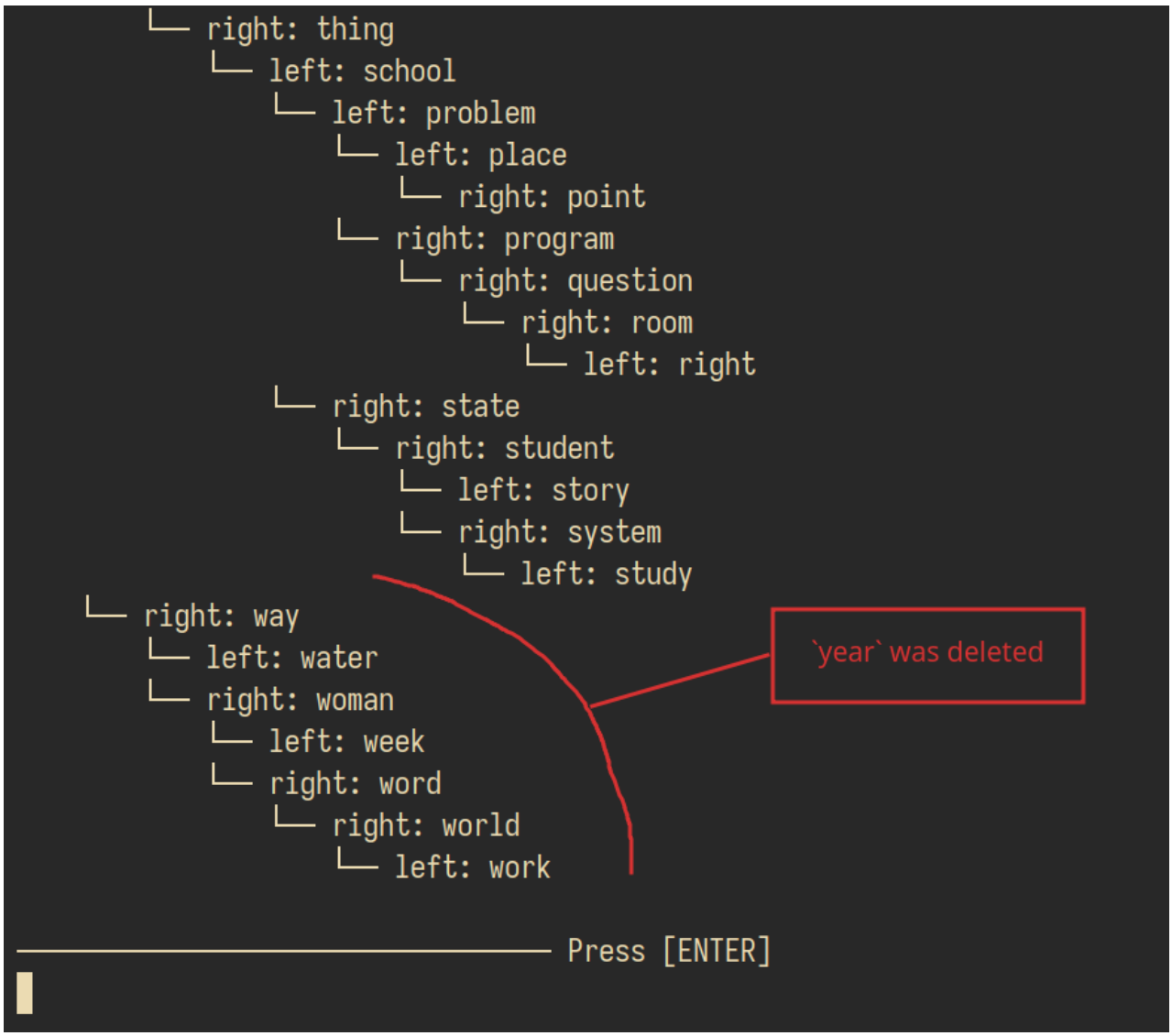

Figure 5-11: State of Word BST After Deletion

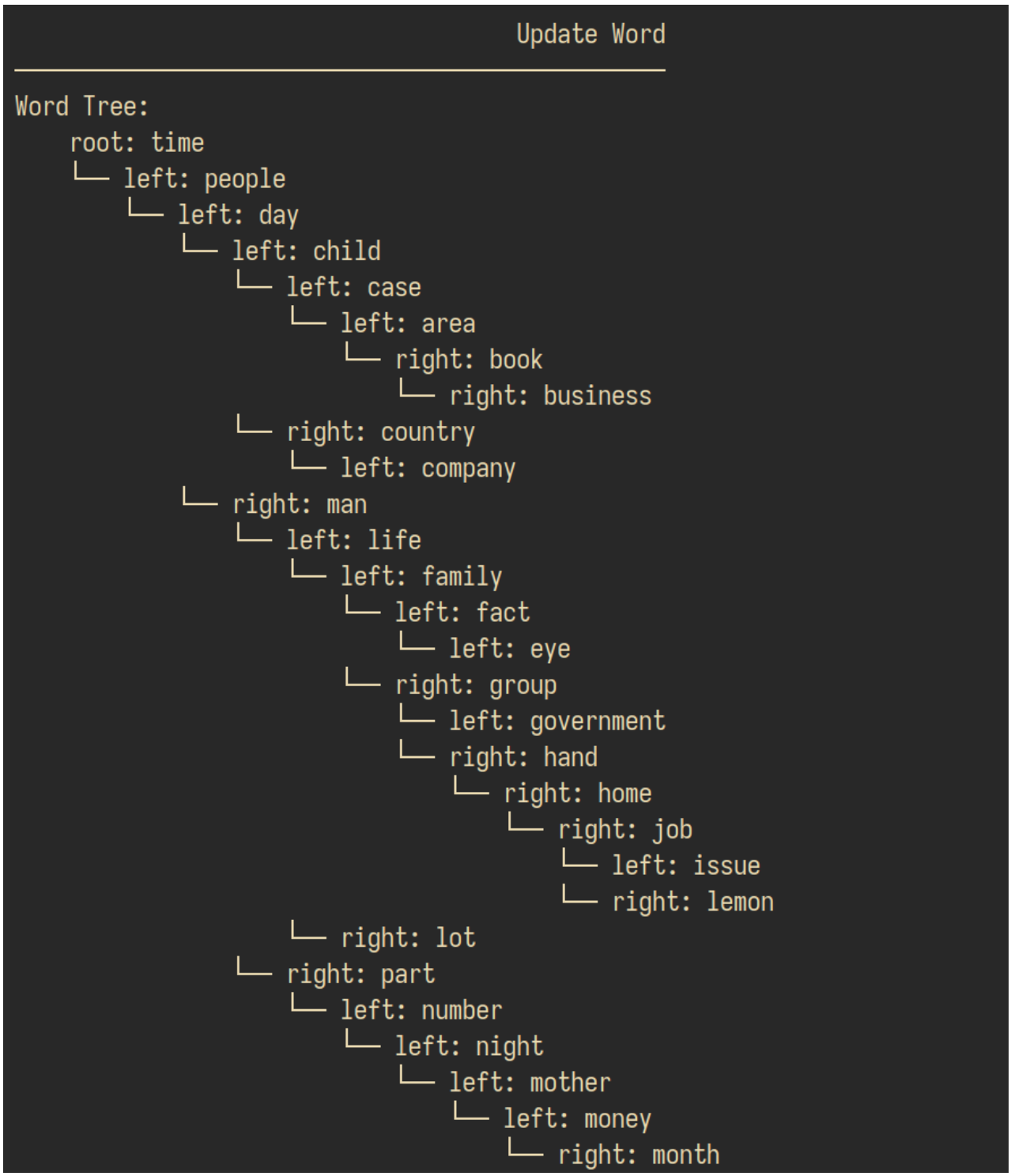

image continued…

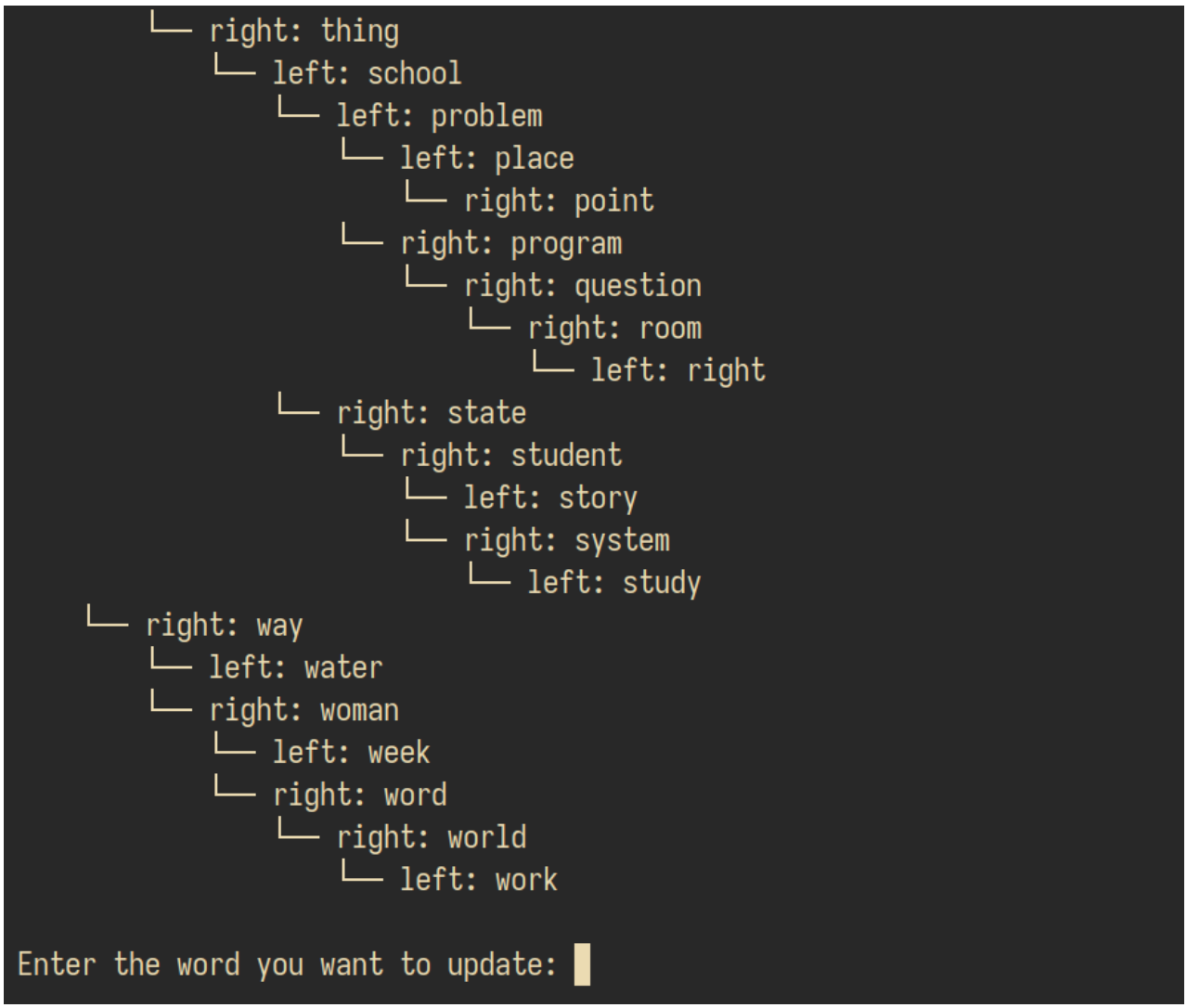

Figure 5-12: Update Menu

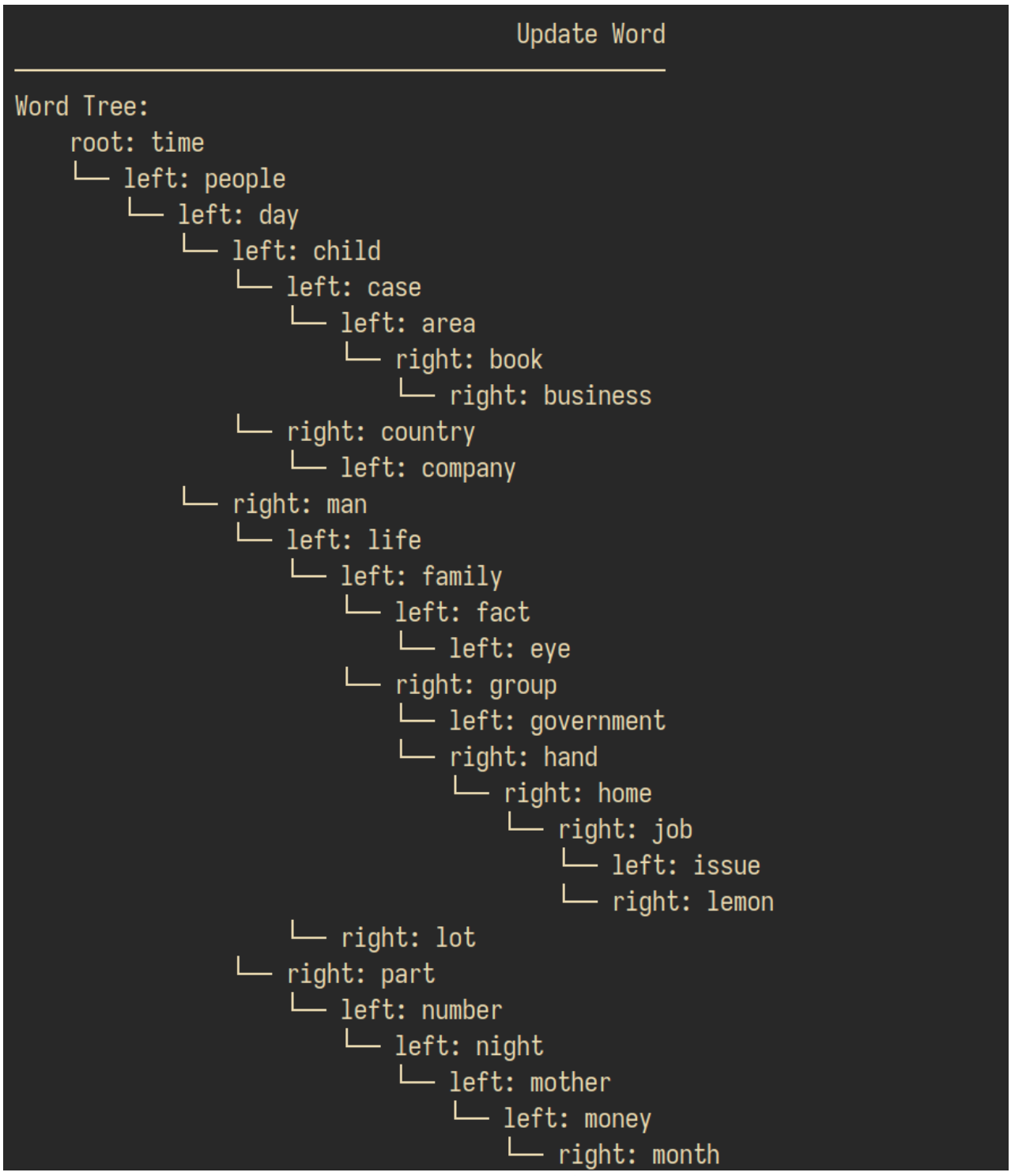

image continued…

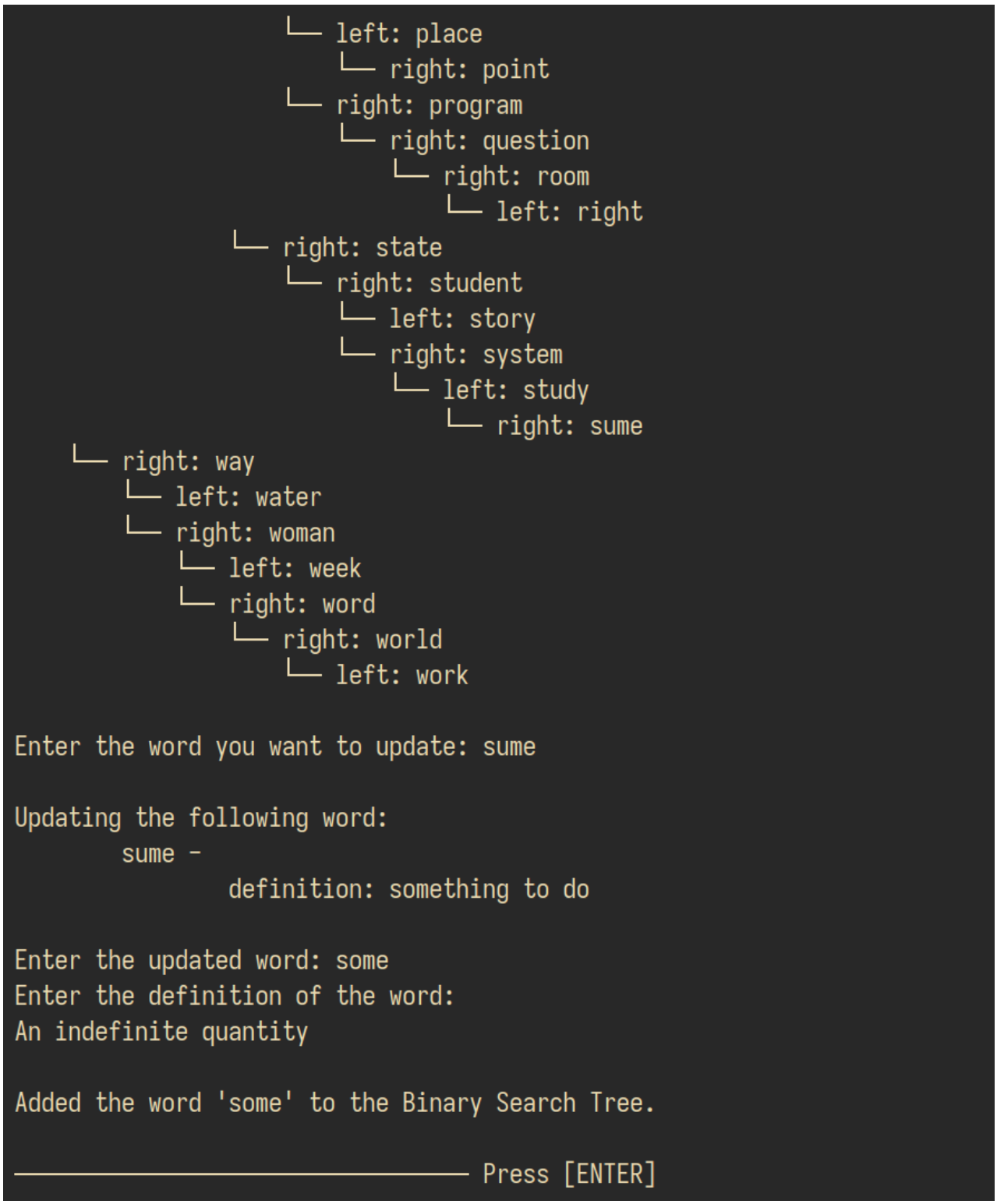

Figure 5-13: Update an Existing Word

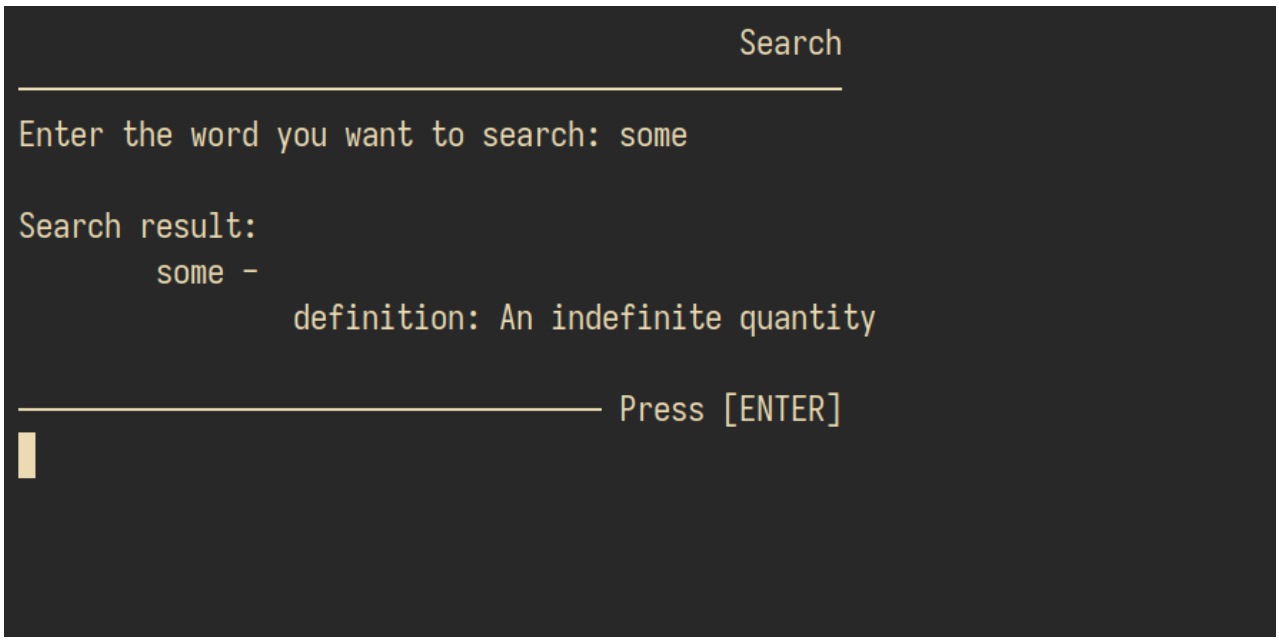

Figure 5-14: Status after Update

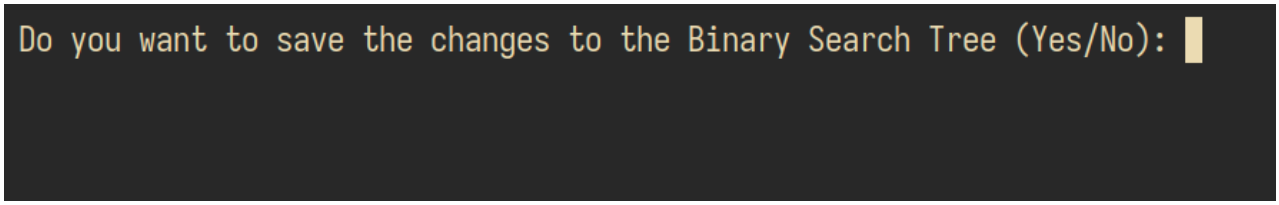

Figure 5-15: Exit with or without saving

6 LIMITATIONS

Limited Word and Definition Storage

The word dictionary's storage is solely based on a binary search tree (BST). While BSTs are efficient for searching, insertion, and deletion, they have limitations on the maximum number of words they can store. As the number of words increases, the BST can become unbalanced, affecting the overall efficiency of operations.

Case-Sensitive Search

The current implementation of the word dictionary performs case-sensitive searches. This means that searching for words with different cases (uppercase and lowercase) will yield different results. Users might expect case-insensitive searching for a more user-friendly experience.

Limited Suggestion Mechanism

The suggestion mechanism for similar words is based on finding words with the same prefix. While this can be helpful, it might not provide accurate suggestions for words that are not found due to small typos or variations.

Single Definition per Word
The current design only allows for a single definition for each word. In reality, words can have multiple meanings or usages. Adding support for multiple definitions or example sentences could enhance the usability of the dictionary.

Lack of Error Handling for Input

The project lacks comprehensive error handling for user input. Invalid inputs, such as entering non-alphabetic characters or providing incorrect responses to confirmation prompts, can lead to unexpected behavior or program crashes.

Performance Impact of String Manipulation

The implementation heavily relies on string manipulation, such as transforming words to lowercase and extracting substrings. String manipulation operations can impact performance, especially when dealing with a large number of words and extensive searches.

No Persistence of User Additions

While the project allows users to add words to the dictionary, these additions are not persisted beyond the runtime. If the program is closed, any user-added words will be lost, as there is no mechanism to save these changes to a file or database.

Command-Line Interface Only

The word dictionary's interaction is limited to a command-line interface (CLI). This might not provide the most user-friendly experience, especially for users who are not familiar with using CLI applications.

Lack of Automated Testing

The project lacks automated testing, making it challenging to verify the correctness of different functionalities. Without testing, identifying and fixing bugs can be time-consuming and error-prone.

Limited User Feedback and Documentation

The project lacks comprehensive user feedback messages and documentation to guide users through its functionalities. Clearer messages and instructions could help users understand how to interact with the dictionary effectively.

Lack of Advanced Data Structures

While a binary search tree is a basic data structure for storing and managing words, more advanced data structures, such as hash maps or tries, could improve search efficiency and provide better suggestions.

Limited Scalability

The project's design and implementation might not be suitable for handling large datasets of words and definitions. As the dataset grows, the program's performance and memory usage could become a concern.

7 CONCLUSION

In conclusion, the development of the word dictionary project has provided valuable insights into the world of data structures and their practical applications. This project aimed to create a functional dictionary that allows users to search for, add, delete, update, and display words and their definitions. Throughout the development process, various data structures and algorithms were employed to ensure efficient and accurate word operations. The project's primary data structure, the binary search tree (BST), served as the backbone for storing and managing words. This choice of data structure enabled fast search, insertion, and deletion operations, making the dictionary an effective tool for word lookups. The utilization of a self-balancing BST could have been explored further to mitigate potential performance issues caused by an unbalanced tree. The project also delved into string manipulation techniques for tasks such as converting words to lowercase and extracting substrings. While string manipulation was a vital component of the project, careful consideration of its performance implications is essential, especially as the word database grows. Despite its accomplishments, the project does have limitations, including a limited word storage capacity, case-sensitive search, and a relatively basic suggestion mechanism. These limitations offer opportunities for future improvements and expansions of the project. For instance, implementing case-insensitive search and enhancing the suggestion mechanism could enhance user experience. Additionally, the lack of automated testing and comprehensive user feedback could be addressed in future iterations. Robust testing procedures would help ensure the project's reliability and correctness, while improved user feedback and documentation could make the dictionary more user-friendly. Incorporating features such as support for multiple definitions per word and persistence of user-added words could further elevate the dictionary's functionality. Furthermore, exploring alternative data structures like hash maps or tries could enhance search efficiency and accommodate larger word datasets. Ultimately, this project has been a valuable learning experience, demonstrating the importance of selecting appropriate data structures, designing user-friendly interfaces, and considering potential limitations and improvements. As a student of data structures, this project has provided a platform to apply theoretical knowledge in a practical context, paving the way for further exploration and growth in the field of computer science.

APPENDIX

SOURCE CODE

bst.h

```
/*
  Header file for Binary Search Tree (bst).
  It contains structural definitions and
  prototypes for bst.
*/

#ifndef BST_H
#define BST_H

#include <memory>
#include <string>
```

```
#include <cstddef>

namespace ds {
  template <typename T>
  class bst {
    public:
      struct Node {
        T data;
        std::unique_ptr<Node> left;
        std::unique_ptr<Node> right;

        Node();
        Node(const T & value);
      };

    private:
      size_t count;
      std::unique_ptr<Node> root;

      bool insert_helper(std::unique_ptr<Node> & node, const T & value);
      Node * search_helper(const std::unique_ptr<Node> & node, const T & key) const;
      bool remove_helper(std::unique_ptr<Node> & node, const T & value);
      void print_helper(bst<T>::Node * root, size_t level, std::string prefix) const;
      bool is_full_helper(const std::unique_ptr<Node> & node) const;

    public:
      bst();
      ~bst();

      bool insert(const T& value);
      bool remove(const T& value);
      T *  search(const T& key);
      Node * search_node(const T & key);
      void print() const;

      bool is_empty() const;
      bool is_full() const;
      size_t get_count() const;
      Node * get_root() const;
  };
}

#include "bst.tpp"

#endif // bst_H
```

bst.tpp

```
/**
 * @file bst.tpp
 * @brief Implementation of a Binary Search Tree (BST) template class.
 */

#ifndef BST_TPP
#define BST_TPP
```

```cpp
#include "bst.h"

#include <iostream>
#include <memory>
#include <stdexcept>
#include <limits> // for clearing input buffer

namespace ds {

  /**
   * @brief Default constructor for Node.
   * Initializes a new Node with default values.
   */
  template <typename T>
  bst<T>::Node::Node() : data({}), left(nullptr), right(nullptr) {}

  /**
   * @brief Constructor for Node with initial value.
   * Initializes a new Node with the given value.
   * @param value The initial value to set.
   */
  template <typename T>
  bst<T>::Node::Node(const T & value) : data(value), left(nullptr), right(nullptr) {}

  /**
   * @brief Default constructor for Binary Search Tree.
   * Initializes a new BST with default values.
   */
  template <typename T>
  bst<T>::bst() : count(0), root(nullptr) {}

  /**
   * @brief Destructor for Binary Search Tree.
   * Cleans up memory used by the BST.
   */
  template <typename T>
  bst<T>::~bst() {}

  /**
   * @brief Helper function to insert a value into the Binary Search Tree.
   * Inserts the given value into the tree.
   * @param node Reference to the current node.
   * @param value The value to insert.
   * @return True if the insertion was successful, false otherwise.
   */
  template <typename T>
  bool bst<T>::insert_helper(std::unique_ptr<typename bst<T>::Node> & node, const T &
value) {
    try {
      if(!node) {
        node = std::make_unique<typename bst<T>::Node>(value);
        return true;
      }

      if(value < (node->data)) {
        return insert_helper(node->left, value);
      }
      else if (value > (node->data)) {
        return insert_helper(node->right, value);
      }
```

```
      else {
        return false;
      }
    }
    catch(const std::bad_alloc & e) {
      std::cerr << "Memory allocation failed: " << e.what() << std::endl;
      return false;
    }
  }

  /**
   * @brief Helper function to search for a value in the Binary Search Tree.
   * Recursively searches for the given value in the tree.
   * @param node Pointer to the current node.
   * @param key The value to search for.
   * @return Pointer to the node containing the value, or nullptr if not found.
   */
  template <typename T>
  typename bst<T>::Node * bst<T>::search_helper(const std::unique_ptr<typename
bst<T>::Node> & node, const T & key) const {
    try {
      if (!node) {
        return nullptr;
      }

      if (key < node->data) {
        return search_helper(node->left, key);
      }
      else if (key > node->data) {
        return search_helper(node->right, key);
      }
      else {
        return node.get();
      }
    }
    catch (const std::exception & e) {
      std::cerr << "Exception occured : " << e.what() << std::endl;
      return nullptr;
    }
  }

  /**
   * @brief Helper function to remove a value from the Binary Search Tree.
   * Recursively removes the given value from the tree.
   * @param node Reference to the current node.
   * @param value The value to remove.
   * @return True if the removal was successful, false otherwise.
   */
  template <typename T>
  bool bst<T>::remove_helper(std::unique_ptr<typename bst<T>::Node> & node, const T &
value) {
    try {
      if (!node) {
        return false;
      }

      if (value < node->data) {
        return remove_helper(node->left, value);
      }
      else if (value > node->data) {
```

```
        return remove_helper(node->right, value);
      }
      else {
        if (!node->left) {
          node = move(node->right);
        }
        else if (!node->right) {
          node = move(node->left);
        }
        else {
          Node * successor = node->right.get();
          while (successor->left) {
            successor = successor->left.get();
          }
          node->data = successor->data;
          remove_helper(node->right, successor->data);
        }
        return true;
      }
    }
    catch (const std::exception & e) {
      std::cerr << "Exception occured: " << e.what() << std::endl;
      return false;
    }
  }

  /**
   * @brief Helper function to print the Binary Search Tree.
   * Recursively prints the structure of the tree.
   * @param root Pointer to the root of the tree.
   * @param level Current level in the tree traversal.
   * @param prefix Prefix to display for each node.
   */
  template <typename T>
  void bst<T>::print_helper(bst<T>::Node * root, size_t level, std::string prefix)
const {
    if(root != nullptr) {
      if(level == 0) {
        std::cout << prefix << root -> data << std::endl;
      }
      else {
        std::string indent(level * 4, ' ');
        std::cout << indent << "└── " << prefix << root -> data << std::endl;
      }

      if(root -> left != nullptr || root -> right != nullptr) {
        print_helper(root -> left.get(), level + 1, "Left: ");
        print_helper(root -> right.get(), level + 1, "Right: ");
      }
    }
  }

  /**
   * @brief Insert a value into the Binary Search Tree.
   * Inserts the given value into the tree.
   * @param value The value to insert.
   * @return True if the insertion was successful, false otherwise.
   */
  template <typename T>
  bool bst<T>::insert(const T & value) {
```

```
  try {
    if (insert_helper(root, value)) {
      count++;
      return true;
    }
    return false;
  }
  catch (const std::bad_alloc & e) {
    std::cerr << "Memory allocation failed: " << e.what() << std::endl;
    return false;
  }
}

/**
 * @brief Print the Binary Search Tree.
 * Prints the structure of the tree.
 */
template <typename T>
void bst<T>::print() const {
  if(root != nullptr)
    print_helper(root.get(), 0, "Root: ");
  else
    std::cout << "Binary-Search Tree is empty!" << std::endl;

  std::cin.clear();
  std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}

template <typename T>
T * bst<T>::search(const T & key) {
  try {
    return & search_helper(root, key)->data;
  }
  catch (const std::exception & e) {
    std::cerr << "Exception occured : " << e.what() << std::endl;
    return nullptr;
  }
}

template <typename T>
typename bst<T>::Node * bst<T>::search_node(const T & key) {
  try {
    return search_helper(root, key);
  }
  catch(const std::exception & e) {
    std::cerr << "Exception occured : " << e.what() << std::endl;
    return nullptr;
  }
}

/**
 * @brief Remove a value from the Binary Search Tree.
 * Removes the given value from the tree.
 * @param value The value to remove.
 * @return True if the removal was successful, false otherwise.
 */
template <typename T>
bool bst<T>::remove(const T & value) {
  try {
    if (remove_helper(root, value)) {
```

```
      count--;
      return true;
    }
    return false;
  }
  catch (const std::exception & e) {
    std::cerr << "Exception occured: " << e.what() << std::endl;
    return false;
  }
}

/**
 * @brief Check if the Binary Search Tree is empty.
 * @return True if the tree is empty, false otherwise.
 */
template <typename T>
bool bst<T>::is_empty() const {
  return count == 0;
}

template <typename T>
bool bst<T>::is_full_helper(const std::unique_ptr<bst<T>::Node> & node) const {
  try {
    if(!node)
      return true;
    if((node->left && !node->right) || (!node->left && node->right))
      return false;
    return is_full_helper(node->left) && is_full_helper(node->right);
  }
  catch (const std::exception & e) {
    std::cerr << "Exception occured: " << e.what() << std::endl;
  }
}

/**
 * @brief Check if the Binary Search Tree is a full binary tree.
 * @return True if the tree is a full binary tree, false otherwise.
 */
template <typename T>
bool bst<T>::is_full() const {
  try {
    return is_full_helper(root);
  }
  catch (const std::exception & e) {
    std::cerr << "Exception occured: " << e.what() << std::endl;
  }
}

/**
 * @brief Get the count of nodes in the Binary Search Tree.
 * @return The number of nodes in the tree.
 */
template <typename T>
size_t bst<T>::get_count() const {
  return count;
}


/**
 * @brief Get a pointer to the root node of the Binary Search Tree.
```

```
   * @return Pointer to the root node, or nullptr if the tree is empty.
   */
  template <typename T>
  typename bst<T>::Node * bst<T>::get_root() const {
    return root.get();
  }
}

#endif
```

list.h

```
/*
  Header file for Linked-List.
  It contains structural definitions and
  prototypes for Linked-List.
*/

#ifndef LIST_H
#define LIST_H

namespace ds {
  template <typename T>
  class list {
    private:
      struct node {
        T data;
        node * next;

        node();
        node(const T & data);
      };

      node * head;
      int size;

    public:
      list();
      ~list();


      bool insert_front(const T & data);
      bool insert_back(const T & data);
      bool delete_front();
      bool delete_back();
      bool delete_key(const T & key);
      bool search(const T & key);
      void print();
      node * begin() const;
      node * end() const;
      int get_size() const;

      T & operator[](int index);
  };
}
```

```
#include "list.tpp"

#endif
```

list.tpp

```
/**
 * @file list.tpp
 * @brief Implementation of a singly linked list template class.
 */
#ifndef LIST_TPP
#define LIST_TPP

#include "list.h"

#include <iostream>
#include <stdexcept>

namespace ds {


  /**
   * @brief Default constructor for the Node class.
   * Initializes a new Node with default values.
   */
  template <typename T>
  list<T>::node::node() : data({}), next(nullptr) {}

  /**
   * @brief Constructor for the Node class with initial value.
   * Initializes a new Node with the given value.
   * @param data The initial value to set.
   */
  template <typename T>
  list<T>::node::node(const T & data) : data(data), next(nullptr) {}

  /**
   * @brief Default constructor for the singly linked list.
   * Initializes a new list with default values.
   */
  template <typename T>
  list<T>::list() : head(nullptr), size(0) {}


  /**
   * @brief Destructor for the singly linked list.
   * Cleans up memory used by the list.
   */
  template <typename T>
  list<T>::~list() {
    node * current = head;

    while(current != nullptr) {
      node * temp = current;

      current = current -> next;
```

```
    delete temp;
  }

  head = nullptr;
  size = 0;
}


template <typename T>
bool list<T>::insert_front(const T & data) {
  try {
    node * new_node = new node(data);

    new_node -> next = head; // makes the new node point to the previous head
    head = new_node; // makes the new node the new head

    size++;

    return true;
  }
  catch(const std::bad_alloc & e) {
    std::cout << "Unable to allocate more memory" << std::endl;
    return false;
  }
}

template <typename T>
bool list<T>::insert_back(const T & data) {
  if(size == 0)
    return insert_front(data);
  else {
    try {
      node * new_node = new node(data);
      node * temp = head;

      // find the last node
      while(temp -> next != nullptr)
        temp = temp -> next;

      temp -> next = new_node;
      new_node -> next = nullptr;

      size++;

      return true;
    }
    catch(const std::bad_alloc & e) {
      std::cout << "Unable to allocate more memory" << std::endl;
      return false;
    }
  }
}

template <typename T>
bool list<T>::delete_front() {
  if(head == nullptr)
    return false;
  else {
    node * temp = head;
```

```
    head = head -> next;
    delete temp;

    size--;

    return true;
  }
}

template <typename T>
bool list<T>::delete_back() {
  if(head == nullptr)
    return false;
  else {
    node * temp = head;

    // find the 2nd last node
    while(temp -> next -> next != nullptr)
      temp = temp -> next;

    delete temp -> next; // delete the last node
    temp -> next = nullptr; // make the 2nd last node the last node
    size--;

    return true;
  }
}

template <typename T>
bool list<T>::delete_key(const T & key) {
  //if(head == nullptr)
  if(size == 0)
    return false;
  else if(size == 1) {
    if(head -> data == key) {
      delete head;
      head = nullptr;
      size--;

      return true;
    }
  }
  else if(size == 2) {
    if(head -> data == key) {
      node * temp = head;
      head = head -> next;
      delete temp;
      size--;

      return true;
    }
    else if(head -> next -> data == key) {
      delete head -> next;
      head -> next = nullptr;
      size--;

      return true;
    }
  }
  else {
```

```
      for (node * temp = head; temp -> next != nullptr; temp = temp -> next) {
        if( (temp -> next != nullptr) && (temp -> next -> data == key) ) { // find the
previous node of the target node
          node * target = temp -> next;

          temp -> next = target -> next; // make the previous node of the target point
to the node after the target node
          //temp -> next = temp -> next -> next; // make the previous node of the
target point to the node after the target node
          delete target;
          size--;

          return true;
        }
      }
    }

    return false;
  }

  template <typename T>
  bool list<T>::search(const T & key) {
    for(node * temp = head; temp != nullptr; temp = temp -> next) {
      if(temp -> data == key)
        return true;
    }

    return false;
  }

  template <typename T>
  void list<T>::print() {
    std::cout << "[";

    for(node * i = head; i != nullptr; i = i -> next) {
      std::cout << " " << i -> data;

      if(i -> next == nullptr)
        std::cout << " ";
      else
        std::cout << ",";
    }

    std::cout << "]" << std::endl;
  }

  template <typename T>
  typename list<T>::node * list<T>::begin() const {
    return head;
  }

  template <typename T>
  typename list<T>::node * list<T>::end() const {
    node * temp = head;

    while(temp -> next != nullptr)
      temp = temp -> next;

    return temp;
  }
```

```
  template <typename T>
  int list<T>::get_size() const {
    return size;
  }


  template <typename T>
  T& ds::list<T>::operator[](int index) {
      if (index < 0 || index >= size) {
          throw std::out_of_range("Index out of bounds");
      }

      node* current = head;
      for (int i = 0; i < index; i++) {
          current = current->next;
      }

      return current->data;
  }
}

#endif
```

word.h

```
/*
  Header file for user defined data type word.
  It contains structural definitions and
  prototypes for bst.

  Written by : Md Murad Khan Limon
  Date       : 20 August, 2023
*/

#ifndef WORD_H
#define WORD_H

#include <string>

#include "bst.h"
#include "list.h"

class word {
  private:
    std::string term;
    std::string definition;

  public:
    word();
    word(const std::string & term, const std::string & definition);
    ~word();

    std::string get_term() const;
    std::string get_definition() const;
    void set_term(const std::string & term);
```

```
    void set_definition(const std::string & definition);
    void display() const;

    void operator()(const std::string & term, const std::string & definition);
    bool operator==(const word & other) const;
    bool operator!=(const word & other) const;
    bool operator<(const word & other) const;
    bool operator>(const word & other) const;
    friend std::ostream& operator<<(std::ostream& os, const word& w);

    operator std::string() const;
};

#endif //WORD_H
```

word.cpp

```
/**
 * @file word.cpp
 * @brief Contains the implementation of the word class.
 */
#include "../include/word.h"

#include <iostream>
#include <string>
#include <algorithm> // for std::transform
#include <cctype> // for std::tolower

/**
 * @brief Default constructor for the word class.
 */
word::word() : term({}), definition({}) {}

/**
 * @brief Parameterized constructor for the word class.
 * @param term The term associated with the word.
 * @param definition The definition of the word.
 */
word::word(const std::string & term, const std::string & definition) :
definition(definition) {
  this->term = term;

  std::transform(this->term.begin(), this->term.end(), this->term.begin(),
    [](unsigned char c) {return std::tolower(c);} );
}

/**
 * @brief Destructor for the word class.
 */
word::~word() {}


/**
 * @brief Get the term associated with the word.
 * @return The term as a string.
```

```
 */
std::string word::get_term() const {
  return term;
}

/**
 * @brief Get the definition of the word.
 * @return The definition as a string.
 */
std::string word::get_definition() const {
  return definition;
}

/**
 * @brief Set the term associated with the word.
 * @param term The term to set.
 */
void word::set_term(const std::string & term) {
  this->term = term;
}

/**
 * @brief Set the definition of the word.
 * @param definition The definition to set.
 */
void word::set_definition(const std::string & definition) {
  this->definition = definition;
}

/**
 * @brief Display the word and its definition.
 */
void word::display() const {
  std::cout << "\t" << term << " -" << std::endl;
  std::cout << "\t\t" << "definition: " << definition << std::endl;
}



/**
 * @brief Overloaded function call operator to set term and definition.
 * @param term The term to set.
 * @param definition The definition to set.
 */
void word::operator()(const std::string & term, const std::string & definition) {
  this->term = term;
  this->definition = definition;

  std::transform(this->term.begin(), this->term.end(), this->term.begin(),
    [](unsigned char c) {return std::tolower(c);} );
}

/**
 * @brief Overloaded equality operator.
 * @param other Another word object to compare with.
 * @return True if the words are equal, otherwise false.
 */
bool word::operator==(const word & other) const {
  return term == other.term;
}
```

```cpp
/**
 * @brief Inequality operator to compare two words for inequality.
 * @param other The other word to compare with.
 * @return True if the words are not equal, false otherwise.
 */
bool word::operator!=(const word & other) const {
  return !(*this == other);
}

/**
 * @brief Less than operator to compare two words.
 * @param other The other word to compare with.
 * @return True if this word is less than the other word, false otherwise.
 */
bool word::operator<(const word & other) const {
  return term < other.term;
}

/**
 * @brief Greater than operator to compare two words.
 * @param other The other word to compare with.
 * @return True if this word is greater than the other word, false otherwise.
 */
bool word::operator>(const word & other) const {
  return term > other.term;
}

/**
 * @brief Output stream operator to display a word.
 * @param os The output stream.
 * @param w The word to display.
 * @return The output stream.
 */
std::ostream & operator<<(std::ostream & os, const word & w) {
  os << w.term;

  return os;
}

/**
 * @brief Conversion operator to convert word to string.
 * @return The term of the word as a string.
 */
word::operator std::string() const {
  return term;
}
```

utility.h

```cpp
#ifndef UTILITY_H
#define UTILITY_H

#include <string>

#include "bst.h"
#include "word.h"
```

```
namespace util {
  ds::list<std::string> str_split(std::string *line, char delimeter);
  void load_database(ds::bst<word> * WORD_TREE);
  void store_database_helper(ds::bst<word>::Node * node, std::ofstream *
output_file_ptr);
  void store_database(ds::bst<word> * WORD_TREE);
  void save_changes(ds::bst<word> * WORD_TREE);
  void clear_input_buffer();
  void wait_for_input();
  std::string to_lowercase(const std::string * str);
  void clear_screen();
  bool word_is_alpha(const std::string * word);
  std::string input_word(std::string * message = nullptr);
  std::string input_sentence();
  bool confirmation_check(std::string * message = nullptr);
}

#endif
```

utility.cpp

```
/**
 * @file utility.cpp
 * @brief Implementation of utility functions.
 */

#ifndef UTILITY_CPP
#define UTILITY_CPP

#include <iostream>
#include <limits> // for clearing input buffer
#include <algorithm> // for std::transform
#include <cctype> // for std::tolower
#include <string>
#include <fstream>
#include <sstream>

#include "../include/list.h"
#include "../include/bst.h"
#include "../include/utility.h"

namespace util {
  /**
   * @brief Splits a string into a list of tokens using the specified delimiter.
   * @param line The input string to be split.
   * @param delimeter The delimiter character.
   * @return A list of tokens extracted from the input string.
   */
  ds::list<std::string> str_split(std::string * line, char delimeter) {
    std::istringstream line_stream(*line);
    std::string token = {};
    ds::list<std::string> token_list;

    while(std::getline(line_stream, token, delimeter)) {
      token_list.insert_back(token);
```

```
    }

    return token_list;
  }

  /**
   * @brief Loads word data from a file into a binary search tree.
   * @param WORD_TREE Pointer to the binary search tree to be populated.
   */
  void load_database(ds::bst<word> * WORD_TREE) {
    std::string filename = "../data/database";

    std::ifstream input_file(filename);

    if(!input_file) {
      std::cerr << "Error loading database from file!" << std::endl;

      wait_for_input();
    }
    else {
      std::string line;
      char delimeter = '|';

      while(std::getline(input_file, line)) {
        ds::list<std::string> token_list = str_split(&line, delimeter);
        word new_word;

        if(token_list.get_size() == 1) {
          new_word(token_list[0], {});
        }
        else if(token_list.get_size() >= 2) {
          new_word(token_list[0], token_list[1]);
        }
        else {
          break;
        }

        (*WORD_TREE).insert(new_word);
      }
      input_file.close();
    }
  }

  /**
   * @brief Recursively stores word data from the binary search tree into a file.
   * @param node Pointer to the current node in the binary search tree.
   * @param output_file_ptr Pointer to the output file stream.
   */
  void store_database_helper(ds::bst<word>::Node * node, std::ofstream *
output_file_ptr) {
    if (node == nullptr)
      return;

    // pre-order traversal
    (*output_file_ptr) << node->data.get_term() << "|" << node->data.get_definition()
<< "\n";
    store_database_helper(node -> left.get(), output_file_ptr);
    store_database_helper(node -> right.get(), output_file_ptr);

  }
```

```
  /**
   * @brief Stores word data from the binary search tree into a file.
   * @param WORD_TREE Pointer to the binary search tree to be stored.
   */
  void store_database(ds::bst<word> * WORD_TREE) {
    std::string filename = "../data/database";

    std::ofstream output_file(filename);

    if(!output_file) {
      std::cerr << "Error loading database from file!" << std::endl;

      wait_for_input();
    }
    else {
      store_database_helper((*WORD_TREE).get_root(), &output_file);

      output_file.close();
    }
  }

  /**
   * @brief Saves changes made to the binary search tree back to the database file.
   * @param WORD_TREE Pointer to the binary search tree to be saved.
   */
  void save_changes(ds::bst<word> * WORD_TREE) {
    std::string message = "Do you want to save the changes to the Binary Search Tree
(Yes/No): ";

    if(util::confirmation_check(&message)) {
      util::store_database(WORD_TREE);
      std::cout << "Changes were saved." << std::endl;
    }
    else {
      std::cout << "Changes were not saved." << std::endl;
    }
  }

  /**
   * @brief Clears the input buffer.
   */
  void clear_input_buffer() {
    // Clear the input buffer
    std::cin.clear();
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
  }

  /**
   * @brief Pauses program execution and waits for user input.
   */
  void wait_for_input() {
    std::string line = {};

    std::cout << std::endl;
    std::cout << "———————————————————————————— Press [ENTER]";
    std::cout << std::endl;

    std::getline(std::cin, line); // wait for Enter key
  }
```

```
  /**
   * @brief Converts a string to lowercase.
   * @param str Pointer to the input string.
   * @return The input string converted to lowercase.
   */
  std::string to_lowercase(const std::string * str) {
    std::string lowercase = *str;

    // convert the input string to lowercase using lamda function
    std::transform(lowercase.begin(), lowercase.end(), lowercase.begin(),
      [](unsigned char c) {return std::tolower(c);} );

    return lowercase;
  }

  /**
   * @brief Clears the terminal screen.
   */
  void clear_screen() {
    #ifdef _WIN32
      system("cls"); // clear the screen on windows
    #else
      system("clear"); // clear the screen on linux
    #endif
  }

  /**
   * @brief Checks if a string consists of alphabetic characters only.
   * @param word Pointer to the input string.
   * @return True if the input string contains only alphabetic characters, false
otherwise.
   */
  bool word_is_alpha(const std::string * word) {
    for (char c : (*word)) {
      if (!std::isalpha(c)) {
          return false; // If a non-alphabetic character is found, return false
      }
    }
    return true; // All characters are alphabetic
  }

  /**
   * @brief Takes user input for a word, ensuring it only contains alphabetic
characters.
   * @param message Pointer to the input prompt message (optional).
   * @return A valid lowercase word entered by the user.
   */
  std::string input_word(std::string * message) {
    std::string str = {};

    // take input until string contains only alphabets
    do {
      if(message == nullptr)
        std::cout << "Enter a word: " << std::endl;
      else
        std::cout << (*message);

      std::cin >> str;
```

```
      clear_input_buffer();

      if(!word_is_alpha(&str)) {
        std::cout << std::endl;
        std::cout << "Error! Word should only contain Alphabets!" << std::endl;
        std::cout << "Please try again!" << std::endl;

        wait_for_input();
      }
      else {
        break;
      }
    } while(true);

    str = to_lowercase(&str);

    return str;
  }

  /**
   * @brief Takes user input for a sentence.
   * @return The sentence entered by the user.
   */
  std::string input_sentence() {
    std::string sentence = {};
    std::getline(std::cin, sentence);

    return sentence;
  }

  /**
   * @brief Prompts the user for confirmation with a given message.
   * @param message Pointer to the confirmation message.
   * @return True if the user confirms (yes), false if they decline (no).
   */
  bool confirmation_check(std::string * message) {
    std::string confirmation;

    do {
      confirmation = util::input_word(message);

      if(confirmation == "y" || confirmation == "yes")
        return true;
      else if(confirmation == "n" || confirmation == "no")
        return false;
      else {
        std::cout << "Please answer with either `yes` or `no`!" << std::endl;
      }
    } while(true);
  }
}

#endif
```

main.cpp

```cpp
/**********************************************
 * @file main.cpp
 * @brief Implementation of a Word Dictionary
 **********************************************/

//#include <cstdlib>
#include <iostream>
#include <string>

#include "../include/bst.h"
#include "../include/word.h"
#include "../include/utility.h"


// Function declarations
void print_menu();
int choice();
ds::bst<word>::Node * find_closest_node(ds::bst<word>::Node * node, const char
& letter, const int index);
void print_suggestions_helper(
  const ds::bst<word>::Node * subtree,
  ds::list<std::string> * suggestion_list,
  std::string matching_substr);
void print_suggestions(word * search);

void add_word_helper(std::string * message);

void add_word();
void search_word();
void delete_word();
void update_word();
void print_tree();

// Global BST to store words
static ds::bst<word> WORD_TREE;

/**
 * @brief The main function that initializes the Word Dictionary program.
 * @return The program exit status.
 */
int main() {
  util::load_database(&WORD_TREE);

  while(1) {
    print_menu();

    switch (choice()) {
      case 1:
        add_word();
        break;
      case 2:
        search_word();
        break;
      case 3:
```

```
        delete_word();
        break;
      case 4:
        update_word();
        break;
      case 5:
        print_tree();
        break;
      case 6:
        util::clear_screen();
        util::save_changes(&WORD_TREE);
        exit(0);
      default:
        std::cout << "Invalid input!" << std::endl;
        break;
    }
  }

  return 0;
}

/**
 * @brief Displays the main menu of the Word Dictionary program.
 */
void print_menu() {
  util::clear_screen();

  std::cout << "                                Word Dictionary" <<
std::endl;
  std::cout << "————————————————————————————————————————————————" <<
std::endl;
  std::cout << "1. Add Word" << std::endl;
  std::cout << "2. Search Word" << std::endl;
  std::cout << "3. Delete Word" << std::endl;
  std::cout << "4. Update Word" << std::endl;
  std::cout << "5. Display Word-Tree" << std::endl;
  std::cout << "6. Quit" << std::endl;
  std::cout << std::endl;
}

/**
 * @brief Takes user input for menu choice and validates it.
 * @return The validated user choice.
 */
int choice() {
  int choice = {};

  do {
    std::cout << "Please enter a corresponding number: " << std::endl;
    std::cin >> choice;

    if (std::cin.fail() || choice < 1 || choice > 6) {
      std::cout << "Invalid input. Please try again." << std::endl;
```

```
      util::clear_input_buffer();
    }
    else {
      break;
    }
  } while (true);

  return choice;
}

/**
 * @brief Recursively finds the closest node in the BST based on the given
letter and index.
 * @param node Pointer to the current node in the BST.
 * @param letter The target letter to search for.
 * @param index The index of the letter in the word.
 * @return Pointer to the closest node found.
 */
ds::bst<word>::Node * find_closest_node(ds::bst<word>::Node * node, const char
& letter, const int index) {
  try {
    if (!node) {
      return nullptr;
    }

    std::string current_word = node->data;

    if ( (index + 1) > current_word.size() || letter > current_word[index]) {
      return find_closest_node(node->right.get(), letter, index);
    }
    else if (letter < current_word[index]) {
      return find_closest_node(node->left.get(), letter, index);
    }
    else {
      return node;
    }
  }
  catch (const std::exception & e) {
    std::cerr << "Exception occured : " << e.what() << std::endl;
    return nullptr;
  }
}

/**
 * @brief Recursively searches and populates a list of suggestions for a given
matching substring.
 * @param subtree Pointer to the current subtree node.
 * @param suggestion_list Pointer to the list of suggestions.
 * @param matching_substr The substring to match.
 */
void print_suggestions_helper(
  const ds::bst<word>::Node * subtree,
  ds::list<std::string> * suggestion_list,
  std::string matching_substr) {
```

```
  try {
    if (!subtree) {
      return;
    }

    print_suggestions_helper(subtree->left.get(), suggestion_list,
matching_substr);
    print_suggestions_helper(subtree->right.get(), suggestion_list,
matching_substr);

    if(subtree->data.get_term().find(matching_substr) == 0) {
        (*suggestion_list).insert_front(subtree->data.get_term());
    }
  }
  catch (const std::exception & e) {
    std::cerr << "Exception occured : " << e.what() << std::endl;
  }
}

/**
 * @brief Displays suggestions for a word search that did not yield any
results.
 * @param search Pointer to the search term.
 */
void print_suggestions(word * search) {
  util::clear_screen();

  std::string target_str = *search;
  std::string matching_substr = "";
  ds::bst<word>::Node * current_node = WORD_TREE.get_root();
  ds::bst<word>::Node * temp_node = nullptr;

  std::cout << "                                    Suggestions" <<
std::endl;
  std::cout << "──────────────────────────────────────────────────" <<
std::endl;
  std::cout << "The word '" << *search << "' was not found!" << std::endl <<
std::endl;

  for(int i = 0; i < target_str.size(); i++) {
    char letter = target_str[i];

    temp_node = find_closest_node(current_node, letter, i);

    if(temp_node == nullptr && i == 0) {
      return;
    }
    else if(temp_node == nullptr) {
      break;
    }
    else {
      current_node = temp_node;
      matching_substr += letter;
```

```
    }
  }

  int suggestion_count = 5;
  ds::list<std::string> suggestion_list;

  print_suggestions_helper(current_node, &suggestion_list, matching_substr);

  if(suggestion_list.get_size() < suggestion_count)
    suggestion_count = suggestion_list.get_size();

  if(suggestion_list.get_size() != 0) {
    std::cout << "Did you mean:" << std::endl;

    std::cout << "[ ";
    for(int index = 0; index < suggestion_count; index++) {
      std::cout << suggestion_list[index];

      if(index != suggestion_count - 1)
        std::cout << ", ";
    }
    std::cout << " ]";
  }
}

/**
 * @brief Assists in adding a new word to the BST by taking user input for
terminology and definition.
 * @param message Pointer to the message to display.
 */
void add_word_helper(std::string * message) {
  std::string terminology = util::input_word(message);

  std::cout << "Enter the definition of the word: " << std::endl;
  std::string definition = util::input_sentence();

  // instantiate a word object named 'new_word'
  word new_word(terminology, definition);

  if(WORD_TREE.search(new_word) == nullptr) {
    if (WORD_TREE.insert(new_word)) {
      std::cout << std::endl;
      std::cout << "Added the word '" << terminology << "' to the Binary
Search Tree." << std::endl;

      util::wait_for_input();

      return;
    }
    else {
      std::cout << "Error! Could not add the word '" << terminology << "'." <<
std::endl;
    }
  }
```

```
  else {
    std::cout << "Could not add '" << terminology << "'! It is already in the
BST!" << std::endl;

    util::wait_for_input();
  }
}

/**
 * @brief Adds a new word to the BST with user-provided terminology and
definition.
 */
void add_word() {
  util::clear_screen();

  std::cout << "                                        Add Word" <<
std::endl;
  std::cout << "──────────────────────────────────────────────────" <<
std::endl;

  std::string message = "Enter the word you want to add: ";

  add_word_helper(&message);
}

/**
 * @brief Searches for a word in the BST and displays its definition if found;
otherwise, suggests similar words.
 */
void search_word() {
  util::clear_screen();

  std::string target_str = {};
  ds::bst<word>::Node * root = WORD_TREE.get_root();

  std::cout << "                                       Search" <<
std::endl;
  std::cout << "──────────────────────────────────────────────────" <<
std::endl;

  std::string message = "Enter the word you want to search: ";
  target_str = util::input_word(&message);

  // instantiate a word object named 'target'
  word target(target_str, {});
  ds::bst<word>::Node * found_node = WORD_TREE.search_node(target);

  if(root != nullptr) {
    if(found_node == nullptr) {
      //std::cout << "The word '" << target_str << "' was not found." <<
std::endl;
      print_suggestions(&target);

      util::wait_for_input();
```

```
    }
    else {
      std::cout << std::endl;
      std::cout << "Search result:" << std::endl;

      (*found_node).data.display();

      util::wait_for_input();
    }
  }
}

/**
 * @brief Deletes a word from the BST if it exists.
 */
void delete_word() {
  util::clear_screen();

  std::string terminology = {};

  std::cout << "                                    Delete Word" <<
std::endl;
  std::cout << "──────────────────────────────────────────────────" <<
std::endl;

  std::cout << "Word Tree:" << std::endl;
  WORD_TREE.print();
  std::cout << std::endl;

  std::string message = "Enter the word you want to delete: ";
  terminology = util::input_word(&message);

  word new_word(terminology, {});

  if(!WORD_TREE.search(new_word)) {
    std::cout << "The word '" << terminology << "' was not found!" <<
std::endl;

    util::wait_for_input();
  }
  else {
    std::string message = "Are you sure you want to delete the word '" +
terminology + "' (Yes/No): ";

    if(util::confirmation_check(&message)) {
      if (WORD_TREE.remove(new_word)) {
        std::cout << std::endl;
        std::cout << "The word '" << terminology << "' was deleted
successfully!" << std::endl;

        util::wait_for_input();

        return;
      }
```

```
      else {
        std::cout << "Error! The word '" << terminology << "' could not be
deleted!" << std::endl;

        util::wait_for_input();
      }
    }
  }
}

/**
 * @brief Updates a word in the BST by removing and re-adding it with new
values.
 */
void update_word() {
  util::clear_screen();

  std::string target_terminology = {};
  std::string terminology = {};

  std::cout << "                                    Update Word" <<
std::endl;
  std::cout << "──────────────────────────────────────────────────" <<
std::endl;

  std::cout << "Word Tree:" << std::endl;
  WORD_TREE.print();
  std::cout << std::endl;


  std::string message = "Enter the word you want to update: ";
  target_terminology = util::input_word(&message);

  word target_word(target_terminology, {});

  if(!WORD_TREE.search(target_word)) {
    std::cout << "Error! The word '" << target_terminology << "' was not
found!" << std::endl;

    util::wait_for_input();
  }
  else {
    word word_to_delete = WORD_TREE.search_node(target_word)->data;

    std::cout << std::endl;
    std::cout << "Updating the following word:" << std::endl;

    word_to_delete.display();

    WORD_TREE.remove(word_to_delete);

    std::cout << std::endl;
    message = "Enter the updated word: ";
    add_word_helper(&message);
```

```
  }
}

/**
 * @brief Displays the entire contents of the Word Dictionary BST.
 */
void print_tree() {
  util::clear_screen();

  std::cout << "                                        Display" <<
std::endl;
  std::cout << "────────────────────────────────────────────────" <<
std::endl;

  WORD_TREE.print();

  util::wait_for_input();
}
```