

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:**

31th August, 2023

Table of Contents

//modification in print()

**1 INTRODUCTION**

**1.1 Background and Problem Statement**

* + 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.

**1.1.2 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.
5. **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 |