

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

**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[] |