Data Structures Project



Muhammad Shahmir Ahmed Siddiqui 21K-3563

Abdul Hayyan 21K-3626

Daniyal Ahmed 21K-3577

BCY-4B

Dictionary in C++ Using Binary Trees

**Abstract:**

Our project is an implementation of a dictionary using hashing technique. The program initializes an empty dictionary and then takes input from the user to either add a word with its meaning, search for a word, or exit the program. Each word is added to the dictionary by generating a hash value of the word and then using that value as an index to store the word and its meaning in a hash table. When a user searches for a word, the program generates the hash value of the word and then looks up the hash table to see if the word exists. If the word is found, the program displays its meaning, otherwise it informs the user that the word is not in the dictionary. The program continues to run until the user chooses to exit.

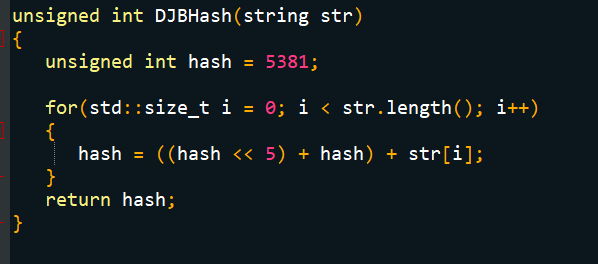
**Intended Uses:**

The above program is a simple implementation of a dictionary using hash tables. Such a program could be used in a variety of contexts where there is a need to store and retrieve data efficiently. Some examples include:

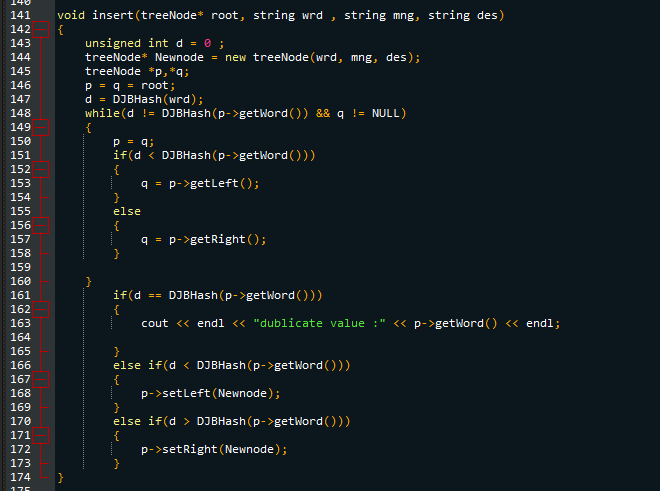
* Spell checkers: In spell checkers, a dictionary can be used to check if a given word is spelled correctly or not. By using hash tables to store the dictionary, the spell checker can quickly determine if a word is valid or not.
* Search engines: Search engines like Google or Bing maintain large indexes of web pages and their contents. A hash table-based implementation of a dictionary could be used to store and retrieve the indexed pages and their contents, allowing for fast and efficient searching.
* Language translators: Language translators need to maintain dictionaries of words and their translations in different languages. A hash table-based implementation of a dictionary can be used to quickly lookup words and their translations.
* Password managers: Password managers store usernames and passwords for various websites and applications. Using a hash table-based implementation of a dictionary, password managers can store and retrieve login information quickly and efficiently.

**Features**

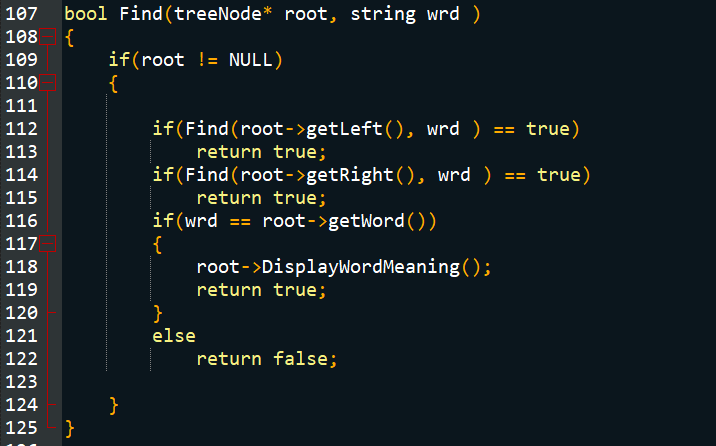
* **DJBhash():** This function takes a string word as input and returns the hash value for that word. The hash value is calculated using the built-in hash() function, which generates a unique integer value for each input string.



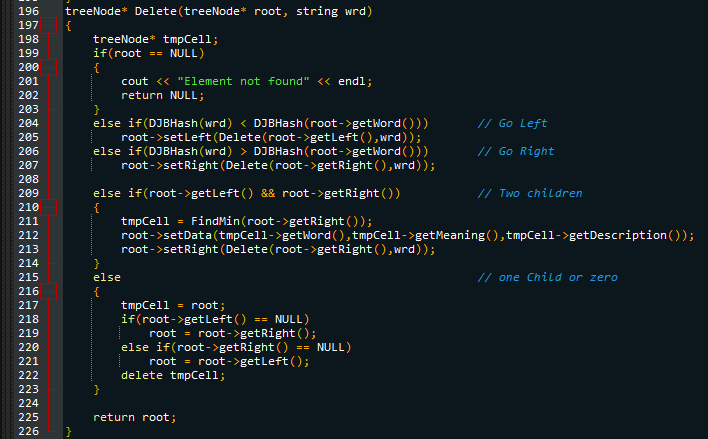
* **insert():** This function inserts a new word into the dictionary. It first calculates the hash value for the word using the DJBhash function, and then uses the hash value as an index to store the word in the dictionary. If there is already a word stored at that index, it uses linear probing to find the next available index to store the new word.



* **search():** This function searches for a word in the dictionary. It first calculates the hash value for the word using the DJBhash() function, and then uses the hash value as an index to check if the word is stored at that index. If not, it uses linear probing to check the next index until it finds the word or reaches an empty index.



* **delete():** This function deletes a word from the dictionary. It first searches for the word using the search() function to find its index in the dictionary. If the word is found, it sets the value at that index to None to delete the word. If the word is not found, it raises a ValueError with an error message.



Overall, the program implements a dictionary using a hash table with linear probing for collision resolution. This data structure is commonly used for fast lookup of key-value pairs, and can be useful in many applications where efficient searching is required.