Theeraj **Project 2 -Design Document** 16/10/2023

21045628

Classes:

In my code, I have three main classes—Hashtable, HashNode, and resizableArray that work together. The Hashtable class serves as the core structure for storing and retrieving words and tokens, handling hash table operations, and addressing collisions using linked lists. I use HashNode objects within the hash table to represent individual words and their tokens within these linked lists. Additionally, the resizableArray class dynamically stores words and their tokens.

1) Hashtable Class

- The Hashtable class represents a hash table that uses a hash function to map words to indexes in an array. It handles operations like inserting words, tokenizing them, and retrieving words by their tokens.
- It contains an array of pointers to HashNode objects, where each pointer points to the head of a linked list. These linked lists are used for collision resolution, ensuring that multiple words that hash to the same index can be stored and retrieved correctly.
- The constructor, Hashtable(size), initializes the hash table. It takes an integer size as a parameter to specify the initial size of the hash table. The constructor creates the array, initializes variables for tracking the size, and sets up the hash table for use.
- The destructor for the Hashtable class cleans up the memory allocated for the hash table and linked lists when the Hashtable object is no longer needed. This is to ensure that there are memory leaks by freeing the memory resources.
- 'int hash(const std::string& key)' calculates and provides the hash index that determines the position of the word in the hash table. Takes in the parameter 'key' which is a const reference to a string to be hashed. Returns an integer representing the index for the hash table.
- 'bool lookup(const std:: string& key, int& index)' searches for a word in the hash table and returns true if found, also sets the index parameter to the index of the word. Parameters are key and index, return type is Boolean (true if word is found, false if not found)
- 'bool insert(const std::string& word)' inserts a word into the hash table, it calculates the hash for the word, checks for collisions and adds the word to the appropriate linked list in the hash table. Parameter is 'word'- a const reference to a string(word) to be inserted. Return type is Boolean (true if the insertion is successful, false if the word already exists).
- 'bool tokenizeFile(std::ifstream& fin)' tokenizes the contents of a file and adds unique words to the hash table. It reads words from a file, checks if they are alphanumeric, tokenizes them if necessary and inserts them into the hash table. Parameter is 'fin'-a reference to std::ifstream for reading from the file. Return type is Boolean (true if at least one word is added, false if none are added).
- 'int tokenizeWord(const std::string& word)' uses the hash index to search for a word in the hash table and returns its token. Parameter is 'word'-a const reference to a string (word) to be tokenized. Return type is Integer (the token for the word, or 0 if the word is not in the dictionary).
- 'void tokenizeString()' Reads words from input, tokenizes them using the tokenizeWord function, and prints their tokens to the output.
- 'void printChain(int k)' prints the linked list (chain) at a specific index in the hash table. The Parameter is 'k'- An integer representing the index of the chain to be printed.
- 'std::string retrieveWord(int token)' Accesses the resizableArray using the token as an index and returns the corresponding word. Parameter is token and Return type is string.
- 'std::string tokensToString()' Reads tokens from input, converts them to words using the retrieveWord function, and returns the words as a space-separated string. No parameter, returns a space separated string corresponding to tokens.

2) HashNode Class

- The HashNode class represents a node in a linked list within the hash table. Each HashNode object stores a word, its token, and a pointer to the next node in the linked list.
- The constructor takes parameters for word, token, and a pointer to the next node. It initializes the object with the provided values.
- Since HashNode class doesn't allocate any dynamic memory in my code, there's no need for a custom destructor.

3) resizableArray Class

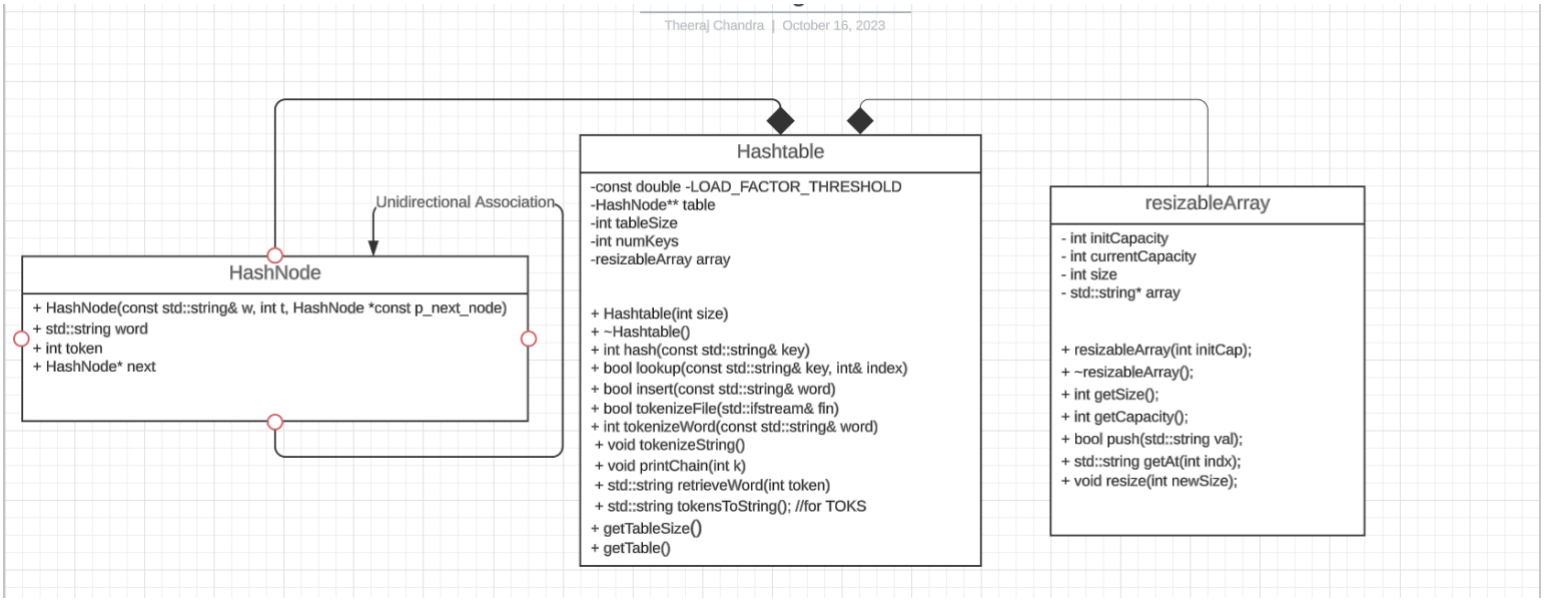
- The resizableArray class is a dynamic array data structure used to store words and their tokens. It allows efficient resizing as needed, also required for efficient word retrieval by the token.
- The constructor takes an integer parameter, initCap, which specifies the initial capacity of the array. It initializes the array, capacity, and size variables. It essentially sets up the initial capacity for the array and allocates memory accordingly.
- The destructor for resizableArray class ensures that the allocated memory is released when the object is destroyed.
- 'int getSize()' retrieves the size (number of elements) in the dynamic array. Has no parameters, return type is integer (size).
- 'int getCapacity()' retrieves the current capacity of the dynamic array. Has no parameters, return type is integer (current capacity).
- 'bool push(std::string val)' inserts a string into the dynamic array, also handles resizing if the array is full. Parameter is 'val'- a string to be added to the dynamic array. Return type is Boolean (true if the push is successful, false if resizing fails).
- 'std::string getAt(int indx)' Retrieves the string at a specified index in the dynamic array. Parameter is 'indx': An integer representing the index to retrieve. Return type is String (the element at the specified index).
- 'void resize(int newSize)' resizes the dynamic array to a new specified size. Parameter is 'newSize': An integer representing the new size for the dynamic array. No return type – void.

In summary, the Hashtable class uses the HashNode class to manage linked lists within the hash table for collision resolution. It also uses the resizableArray class to store words and tokens, thereby allowing efficient retrieval by token. These classes work together to provide a solution for handling words and tokens in my code.

[1] **Runtime Analysis**

1. **TOKENIZE**
   We assume uniform hashing (collisions evenly distributed); My TOKENIZE command involves accessing the linked list at a specific index in the hash table takes constant time in this case. Hence, with uniform hashing, TOKENIZE command is an O(1) operation because the time it takes to access a linked list remains constant and doesn't depend on the size of the data.
2. **RETRIEVE**
   The RETRIEVE command is also a O(1) operation because it directly accesses a specific element in an array, which takes constant time. I use the resizableArray to store the words and their tokens. I use the token as the index to access the corresponding element in the array. Hence, this is O(1) and it does not depend on the number of elements in the array.
3. **INSERT**
   Average Case – Without hash collisions, (assuming hash function is evenly distributed) average time complexity for inserting an element is O(1). This is because in absence of collisions, my program directly places the word into the corresponding index in the hash table.
   Worst Case- Considering the worst case scenario, I would need to traverse this entire linked list to insert a word, which may take O(n) time, where n is the number of words in the linked list. This case occurs when there is a collision at a specific index, leading to a long linked-list of words.

UML Diagram



Citations and References:

[1] ChatGPT, For each of my command in the code, calculate the runtime complexities. OpenAI [Online]. https://chat.openai.com/ (October 16, 2023)