**Data structure:** I used an array of pointers to lists. I decided to implement it this way because arrays allow instant access (O(1)) and the total number of buckets won’t change, so there isn’t any need to worry about slow insertion or deletions. The array contained a pointers to lists because lists allow easy resizing (O(1)) and the lists were unsorted, so I would always have to iterate through the entire list. Therefore, lists were ideal since it’s easy to insert items.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

m\_words (array of pointers to lists)

m\_words[nHash] (pointer to list)

|  |
| --- |
| “tide” |
| “diet” |
| “edit” |

list

const unsigned long MAX\_BUCKETS = 49999; //Found that dividing by a prime will result in

less collisions for non-anagrams

const unsigned long PRIME\_NUMS[26] = {2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101};

const unsigned long AAA = static\_cast<unsigned long>('a'); //ASCII value of 65

class DictionaryImpl

{

public:

DictionaryImpl();

~DictionaryImpl();

void insert(string& word);

void lookup(string& letters, void callback(string)) const;

private:

unsigned long hash(const string& key) const;

list<string>\* m\_words; //Array of pointers to lists

};

unsigned long DictionaryImpl::hash(const string& key) const

{

Iterate through the string (character-by-character) and multiply them all together

Each letter is associated with one of the first 26 prime numbers

(hash = hash \* PRIME\_NUMS[static\_cast<unsigned long>(key[i])-AAA];)

Return the product mod with the max number of buckets

(49,999, which is also a prime)

}

NOTE: NOT ALL FUNCTIONS ARE SHOWN!

DictionaryImpl::DictionaryImpl()

{

m\_words = new list<string>[MAX\_BUCKETS];

}

void DictionaryImpl::lookup(string& letters, void callback(string)) const

{

If callback is NULL

Return;

Remove nonletters

If letters is empty

Return;

Find the hash key of the letters.

If there are no words in the list at the hash key location,

Return;

Sort the letters.

Iterate through the list at the hash key location

If the length of letters == length of current word in list

Create a copy of the word in the list and sort it.

If letters (sorted) == word in list (sorted copy)

callback(word in list);

}

void DictionaryImpl::insert(string& word)

{

Remove all non-letters

If the word isn’t empty:

Add the word to the list at the hash key location;

}