CSC340 Advanced Project Report

Fast Spelling check Dictionary with Separate-Chaining Hash-Table

Leo Wang

Fall 2016

Fast Spelling check Dictionary with Separate-Chaining Hash-Table

Table of Contents

1. Introduction
2. Dictionary Word List
3. Test Case
4. ADT
5. Hash Key and Hash Array size
6. Program Structure, Compilation and Execution
7. Summary
8. Appendix : Listing

1. Introduction

For the CSC340 extra credit advanced project, my proposal is to create a fast spelling checker application with separate-chaining hash table. The following is the summary of the project.

The proposed checker program will read a dictionary file of >110K words and records the words in the dictionary file into the data structure. Then it will parse through document in text format and output a log to screen showing words that are not in the dictionary.

The hash table ADT is a very powerful one with fast search performance of big-O(1). That's a desired feature for dictionary search. The separate-chaining hash table can simplify the long hashing routine. The hashing algorithm is very interesting but it requires two separate hash table arrays. Since the maximum length of the chain is well controlled, it still provide O91) performance.

It's developed with Object Oriented Design principles and practices learned in class by using Unified Modeling Language (UML) and Virtual Methods. I applied the methodology learned in the CSC340 class and coding homework to implement the project. I create or use existing abstract data type (ADT) and Standard Template Library (STL) like hash table, linked list, array, and file I/O.

2. Dictionary Word List

Many of the low cost commercial spelling check devices have several hundred thousands of words built-in. I plan to make this spelling checker dictionary with similar storage capability. My original goal was to use the wordlist of more than 350k words but I found the my XCODE environment has capacity issue. For each word stored in the separate-chaining hash table, the storage needed is a pointer in the hash table array and one linked list (one string and one pointer). The maximum number of words my XCODE environment can handle is about 136000 words. The word list I proposed has about 350K words and beyond the XCODE can handle. In order to continue the project, I found the following word list which has 109588 words and is a very representative good quality word list. It will not affect the scope and effort of the goal of the project

This is a list of about 110,000 English words transcribed orthographically. I obtained it from The Interociter bulletin board in Dallas (214/258-1832). The original read.me file said that the list came from Public Brand Software. The original list contained 146,440 words, but I discovered that there were thousands of duplicate words. I resorted the list and removed the duplicates using the Unix utility uniq. The total number of words is now 109,588.

The following is the word list dictionary.txt used in this project. The first entry is the number of words and followed by one word per line.

```
109588
a
aah
aahed
aahing
aahs
aardvark
aardvarks
aardwolf
ab
abaci
aback
abacus
abacuses
abaft
abalone
abalones
abandon
abandoned
abandonedly
abandonee
```

3. Test Case

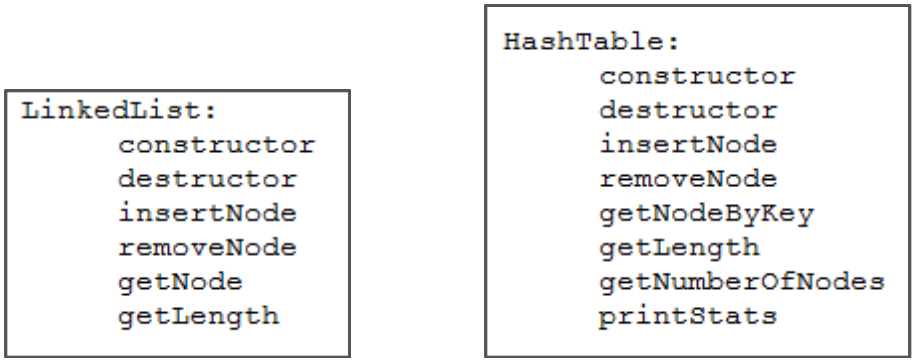
I used many news articles to test the program. Most of them are from CNN.com, CBS.com, and USA today websites. I saved them in text format and name it “document.txt” before performing the spelling check with my dictionary.

The following is the example from CBS.com about the popular news of recent election. This file is listed in the appendix and uploaded in GitHub.

```
U.S. Officials: Putin Personally Involved in U.S. Election Hack
by William M. Arkin, Ken Dilanian and Cynthia McFadden
U.S. intelligence officials now believe with "a high level of
confidence" that Russian President Vladimir Putin became
personally involved in the covert Russian campaign to interfere
in the U.S. presidential election, senior U.S. intelligence
officials told NBC News.
Two senior officials with direct access to the information say
new intelligence shows that Putin personally directed how hacked
material from Democrats was leaked and otherwise used. The
intelligence came from diplomatic sources and spies working for
U.S. allies, the officials said.
Putin's objectives were multifaceted, a high-level intelligence
source told NBC News. What began as a "vendetta" against Hillary
Clinton morphed into an effort to show corruption in American
politics and to "split off key American allies by creating the
image that [other countries] couldn't depend on the U.S. to be a
credible global leader anymore," the official said.
Ultimately, the CIA has assessed, the Russian government wanted
to elect Donald Trump. The FBI and other agencies don't fully
endorse that view, but few officials would dispute that the
Russian operation was intended to harm Clinton's candidacy by
leaking embarrassing emails about Democrats.
```

4. ADT

The following ADT has been implemented in this project. The relationship between ADT hashTable and linkedList is private inheritance. The following is the illustration.



UML specification is used to document the public methods of hashTable and linkedList ADTs.

```
// Adds a node to the Hash Table.
// @pre private data "array" is a reference to an array of Linked Lists and "length" cis the size of the Hash Table
    array.  HashTable is the array of linked list
// @post If the operation was successful, a node is added to the Hash Table.
// @param nodeKey is the data to be filled into the private data "key" of node struct.
// @return No return value needed.
void insertNode( string nodeKey ) ;         // Adds a node to the Hash Table.

// Deletes a node by key from the Hash Table. Returns true if successful.
// @pre private data "array" is a reference to an array of Linked Lists and "length" cis the size of the Hash Table
    array.  HashTable is the array of linked list
// @post If the operation was successful, a node is removed tfrom the Hash Table.
// @param nodeKey is the data to be filled into the private data "key" of node struct.
// @return True if the removal is successful or false if not.
bool removeNode( string nodeKey ) ;         // Deletes a node by key from the Hash Table. Returns true if successful.

// Check Hash Table by key or null if not found.
// @pre private data "array" is a reference to an array of Linked Lists and "length" cis the size of the Hash Table
    array.  HashTable is the array of linked list
// @post If the operation was successful, the reference of the linked list in the hash table is returned
// @param nodeKey is the data to be filled into the private data "key" of node struct.
// @return the reference (pointer) of the match
node * getNodeByKey( string nodeKey ) ;     // Check Hash Table by key or null if not found.

// Prints statistics of the Hash table.
// @pre private data "array" is a reference to an array of Linked Lists and "length" cis the size of the Hash Table
    array.  HashTable is the array of linked list
// @post No change to the private data of the hash table and linked list used in hash table.
// @param No input parameter needed
// @return No return value needed.
void printStats() ;                         // Prints statistics of the Hash table.

// Returns the number of locations in the Hash Table.
// @pre private data "array" is a reference to an array of Linked Lists and "length" cis the size of the Hash Table
    array.  HashTable is the array of linked list
// @post No change to the private data of the hash table and linked list used in hash table.
// @param No input parameter needed
// @return the value of private data "length" in datatype integer.
int getLength() ;                           // Returns the number of locations in the Hash Table.
```

The methods in ADT hashTable are all redefined and it will not use dynamic bonding to use the method of the same name in ADT linkedList. As the class relationship definition, the private inheritance will hide all public, protected, and private members of the ancestor ADT.

The following methods for ADT linkedList:

```
node * linkedList::getNode( string nodeKey ) // Searches for a node by its key.
{
    node * p = head;
    node * q = head;
    while (q)
    {
        p = q;
        if ((p != head) && (p -> key == nodeKey))
            return p;
        q = p -> next;
    }
    return NULL;
}

int linkedList::getLength()   // Returns the length of the list.
{
    return length;
}
```

```
bool linkedList::removeNode( string nodeKey ) // Removes a node from the list by node key.
{
    if (!head -> next) return false;
    node * p = head;
    node * q = head;
    while (q)
    {
        if (q -> key == nodeKey)
        {
            p -> next = q -> next;
            delete q;
            length--;
            return true;
        }
        p = q;
        q = p -> next;
    }
    return false;
}
```

```
void linkedList::insertNode( string nodeKey ) // Inserts a node at the end of the list.
{
    node * newNode = new node { nodeKey, NULL };
    if ( (head->next) == NULL )
    {
        head -> next = newNode;
        length++;
        return;
    }
    node * p = head;
    node * q = head;
    while (q)
    {
        p = q;
        q = p -> next;
    }
    p -> next = newNode;
    newNode -> next = NULL;
    length++;
}
```

The following are the methods for ADT hashTable:

```
void hashTable::insertNode( string nodeKey )    // Adds a node to the Hash Table.
{
    unsigned int index = hash( nodeKey );
    array[ index ].insertNode( nodeKey );
}

bool hashTable::removeNode( string nodeKey )    // Deletes a node by key from the Hash Table.
{
    int index = hash( nodeKey );
    return array[ index ].removeNode( nodeKey );
}

node * hashTable::getNodeByKey( string nodeKey )    // Check Hash Table by key.
{
    int index = hash( nodeKey );
    return array[ index ].getNode( nodeKey );
}
```

```
int hashTable::getLength()                    // Returns the number of locations in the chained Hash Table.
{
    return length;
}

int hashTable::getNumberOfNodes()             // Returns the number of nodes in the chained Hash Table.
{
    int nodeCount = 0;
    for ( int i = 0; i < length; i++ )
    {
        nodeCount += array[i].getLength();
    }
    return nodeCount;
}
```

```
void hashTable::printStats()                    // Prints statistics of the Hash table.
{
    int min = 0;
    int max = 0;
    int mean = 0;
    int tmp;
    for ( int i = 0; i < length; i++ )
    {
        tmp = array[i].getLength();
        if (tmp > max ) max = tmp;
        if (tmp < min) min = tmp;
        mean = mean + tmp;
        // cout << i + 1 << " : " << tmp << endl;    // test only
    }
    int lengthOfChain [100];
    for ( int i=0; i<100; i++) lengthOfChain [i] = 0;
    for ( int i = 0; i < length; i++ )
    {
        tmp = array[i].getLength();
        lengthOfChain[tmp] ++;
    }
    cout << endl << "Hash Table Contains " << getNumberOfNodes() << " linked-list nodes in total." << endl << endl;
    cout << "      Maximum chain length : " << max << endl;
    cout << "      Minimum chain length : " << min << endl;
    cout << "      Mean chain length :    " << mean / length << endl ;
    for ( int i=min; i<=max; i++)
        cout << "      chain length = " << i << "   " << lengthOfChain [i] << " hashtable entries " << endl;

}
```

5. Hash Key and Hash Array size

The number of words in the dictionary word list is 110K. The ADT hashTable array size is set the same size.

```
hashTable::hashTable( int tableLength )    // Constructs the empty Hash Table object.
{
    if (tableLength <= 0) tableLength = 16;
    array = new linkedList[ tableLength ];
    length = tableLength;
}
```

```
  hashTable myHashTable(numWords);      // create Hash Table with length of numWords.
```

The search key has to be generated from the character string of the words. The lengths of the words are relatively short (normally 3~8 characters, certainly not long) and the search key generated from the character string won't provide uniform coverage of the hash table array index.

This is the hash table search key generation algorithm suitable for string objects.

```
unsigned int hashTable::hash( const string & nodeKey )      // (Figure 5-4) hash routine for string objects
{
    unsigned int hashValue = 0;
    for( char ch : nodeKey )
        hashValue = 37 * hashValue + ch;  // 37
    return hashValue % length;
}
```

I did some experiment of different hash table array sizes. The smaller the hash table array size I select, the longer the average length of the chains will be resulted. Since the longer separate-chain length means more comparison in the searching, the search speed performance will be degraded.

The following is the statistics of the separate-chain lengths v.s. Hash Table array size.

Hash Table array size = word count * 2

```
Hash Table Contains 109588 linked-list nodes in total.

      Maximum chain length : 6
      Minimum chain length : 0
      Mean chain length :    0
      chain length = 0   132982 hashtable entries
      chain length = 1   66369 hashtable entries
      chain length = 2   16664 hashtable entries
      chain length = 3   2796 hashtable entries
      chain length = 4   326 hashtable entries
      chain length = 5   35 hashtable entries
      chain length = 6   4 hashtable entries
```

Hash Table array size = word count

```
Hash Table Contains 109588 linked-list nodes in total.

      Maximum chain length : 7
      Minimum chain length : 0
      Mean chain length :    1
      chain length = 0   40374 hashtable entries
      chain length = 1   40231 hashtable entries
      chain length = 2   20095 hashtable entries
      chain length = 3   6805 hashtable entries
      chain length = 4   1728 hashtable entries
      chain length = 5   298 hashtable entries
      chain length = 6   49 hashtable entries
      chain length = 7   8 hashtable entries
```

Hash Table array size = word count / 2

```
Hash Table Contains 109588 linked-list nodes in total.

      Maximum chain length : 11
      Minimum chain length : 0
      Mean chain length :    2
      chain length = 0   7306 hashtable entries
      chain length = 1   14979 hashtable entries
      chain length = 2   14820 hashtable entries
      chain length = 3   9849 hashtable entries
      chain length = 4   4990 hashtable entries
      chain length = 5   1942 hashtable entries
      chain length = 6   679 hashtable entries
      chain length = 7   175 hashtable entries
      chain length = 8   37 hashtable entries
      chain length = 9   15 hashtable entries
      chain length = 10   0 hashtable entries
      chain length = 11   2 hashtable entries
```

If I use the Hash Table array of twice the size of word count, the maximum length of the chain is shortened from 7 to 6 but majority of array entries are not used. If I use the Hash Table array of half the size of word count, the maximum length of the chain is increased from 7 to 11 and fewer empty array entries.

Since the Hash Table array is array of pointers, the performance cost is too high in order to save storage size for the array. Therefore, the array size of the word count size is reasonable tradeoff.

6. Program Structure, Compilation and Execution

The Main program consists of four parts.

```
//*********************************************************************************************
// Main program consists of following steps
//
// Step 1 : Open dictionary.txt, Take numWord (number of word of the dictionary )
//          create the hash table of the size equal to the number of words in the dictionary
//
// Step 2 : For each word, calculate the hash function and insert into the hash table
//
// Step 3 : Output the statistics (min, max, mean)chain length and statistics of hashtable entries
//
// Step 4 : Open document.txt.
//          Read the words from the document and output wach word that is not in the dictionary
//
//*********************************************************************************************
```

The first step initiates the dictionary.

```
   //*************************************************************************
   // Step 1 : Open dictionary.txt, Take numWord (number of word of the dictionary )
   // create the hash table of the size equal to the number of words in the dictionary
   //*************************************************************************

   inFileName = argv[1];                  // Open "dictionary.txt"
   inFile.open(inFileName, ios::in);
   cout << "Open " << inFileName << endl;

   getline(inFile, key, '\n');
   numWords = stoi(key);
   cout << endl << "*********************************************************" << endl;
   cout << endl << "Start Dictionary Hashtable Insertion, total " << numWords << " words" << endl;
   hashTable myHashTable(numWords/2);      // create Hash Table with length of numWords.
```

The second step inserts the words into the dictionary.

```
for (i=1; i<= numWords; i++)
{

    //*************************************************************************************
    // Step 2 : For each word, calculate the hash function and insert into the hash table
    //*************************************************************************************

    inFile >> key;
    myHashTable.insertNode( key );

}  // for i loop
cout << endl << "Finished Dictionary Hashtable Insertion, total " << i-1 << "words" << endl;

inFile.close();                          // Close "dictionary.txt"
```

The third step analyze the statistics of the Hash Table. The Hash Table analysis method “printStats” can provide summary of how the distribution of the nodes in each hash Table chains.

```
//*********************************************************************************************
// Step 3 : Output the statistics (min, max, mean chain length)
//*********************************************************************************************

myHashTable.printStats();              // print Statistics of the Hash Table.
```

The fourth step scan the test-case article

```
//*********************************************************************************************
// Step 4 : Open document.txt.
//          Read the words from the document and output wach word that is not in the dictionary
//*********************************************************************************************

inFileName = argv[2];                  // Open "document.txt"
inFile.open(inFileName, ios::in);
cout << endl << "*********************************************************"<< endl;
cout << endl << "Start to use the dictionary to check article text file." << endl << endl;
cout << endl << endl << "Opening article file " << inFileName << endl << endl;
cout << "The following words are NOT found in the speeling check dictionary:" << endl;
i = 0;
j = 0;

while ( inFile >> key )
{

    filterKey = RemoveSpecials(key);
    if ( myHashTable.getNodeByKey( filterKey ) == NULL )
    {
        j++;
        cout << filterKey << " ";
    }
    else
        i++;
}
cout << endl << endl << "Total " << j << " words Not in spelling check dictionary.";
cout << endl << "Total " << i << " words found in spelling check dictionary." << endl << endl;
inFile.close();                        // Close "document.txt"
return 0;
```

The program is developed in XCODE and the following is how to run in the XCODE environment. It should be straight forward to run in other environment or command mode.

```
## Running tests

* To run the program in command line:
main /your-full-path/dictionary.txt /your-full-path/reading.txt <enter>

* To run the program in XCODE
Add the command line arguments in following click path
Product -> Scheme -> Edit Scheme -> Run -> Arguments
For example:
Add :/Users/wang_family/Desktop/C++/CSC340-extra/dictionary.txt
/Users/wang_family/Desktop/C++/CSC340-extra/reading.txt
```

The output to screen will be in the following format.

```
argc : 3
argv[0] : /Users/wang_family/Library/Developer/Xcode/DerivedData/CSC340-extra-emofllkbvzrzvncpzszwqxlxykuw/Build/Products/
    Debug/CSC340-extra
argv[1] : /Users/wang_family/Desktop/C++/CSC340-extra/dictionary.txt
argv[2] : /Users/wang_family/Desktop/C++/CSC340-extra/document.txt


Open /Users/wang_family/Desktop/C++/CSC340-extra/dictionary.txt

**********************************************************

Start Dictionary Hashtable Insertion, total 109588 words

Finished Dictionary Hashtable Insertion, total 109588words

Hash Table Contains 109588 linked-list nodes in total.

Maximum chain length : 7
Minimum chain length : 0
Mean chain length :    1
chain length = 0   40374 hashtable entries
chain length = 1   40231 hashtable entries
chain length = 2   20095 hashtable entries
chain length = 3   6805 hashtable entries
chain length = 4   1728 hashtable entries
chain length = 5   298 hashtable entries
chain length = 6   49 hashtable entries
chain length = 7   8 hashtable entries

**********************************************************

Start to use the dictionary to check article text file.


Opening article file /Users/wang_family/Desktop/C++/CSC340-extra/document.txt

The following words are NOT found in the speeling check dictionary:
putin m arkin dilanian cynthia mcfadden vladimir putin putin putins hillary clinton morphed couldnt dont clintons putins 17
    russias seniormost putin putin putin mcfaul 2012 2014 hillary clinton 2011 presidentelect mcfaul clinton russias putins
    85 putin putin obama

Total 38 words Not in spelling check dictionary.
Total 545 words found in spelling check dictionary.

Program ended with exit code: 0
```

7. Summary

Dictionary is an important tool to look up certain data from a large database. Since the size of database is large, the search performance is the key consideration for implement the algorithm. Regular array or link based implementation could have undesired search performance results like O(n) or O(log n). Hash Table based dictionary is ideal ADT for the optimized search performance of O(1). Even with simple separate-chaining implementation on the Hash Table, the search performance can still be kept at O(1).

I implemented this spelling check dictionary from the ground up. I practiced the Object oriented design methodology using UML specification; clearly redefine all methods in private inheritance ADT relationship. The word list search and environment setting experience teach me on how to identify the issue and try to find the reasonable solution.

The coding was done right before the final week of the semester and I uploaded all files to the GitHub and this is the project report in hardcopy. I appreciate this change to practice the complete project. It's unlike the coding homework assignment in which we only need to finish the "ToDo" portion of the code to complete the run.

Hash Table ADT could be very usable for the trend of "big data" internet searching. Since the size of the web is almost boundary-less, super-efficient search algorithm is a must as long as the Hash Table can be re-hashed off-line to keep current.

Thanks you Professor.

8. Appendix : Listing