**1. Design document**

**Introduction:**

In this program, we will implement an algorithm that makes the program read words and check if it is spelled correctly. Here we will use a bag that will serve as a dictionary that contains a collection of correctly spelled words. We provide the program with a text file with words in it that need to be checked. To check if a word is spelled correctly at first, we will check to see if the word is in the dictionary.

**Data structures:**

The program uses a doubly linked list class called DoublyLinkedBag where each node in the doubly linked list will have an item (in this case a word), reference to the next node, and reference to the previous node. With public functions like add, remove, and search, the class allows us to add, delete, and search for words respectively.

**Functions:**

In the main program, we have two functions one is the loadDictionary function which loads all the words from the dictionary text file we provide and stores in a bag which the program refers to as the dictionary to check words. The program also has a function that checks if a particular word is in the dictionary.

**The main program:**

The main program first loads the dictionary which should be located at the same directory as the program. It then asks the user to enter the name of the text file in the form of “filename.txt” which should also be in the same directory. Once the user enters the file name the program displays all the words misspelled in the file.

**2. Code list**

/\*\*BagInterface.h\*\*/

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* Listing 1-1.

@file BagInterface.h \*/

#ifndef \_BAG\_INTERFACE

#define \_BAG\_INTERFACE

#include <vector>

using namespace std;

template<class ItemType>

class BagInterface

{

public:

/\*\* Gets the current number of entries in this bag.

@return The integer number of entries currently in the bag. \*/

virtual int getCurrentSize() const = 0;

/\*\* Sees whether this bag is empty.

@return True if the bag is empty, or false if not. \*/

virtual bool isEmpty() const = 0;

/\*\* Adds a new entry to this bag.

@post If successful, newEntry is stored in the bag and

the count of items in the bag has increased by 1.

@param newEntry The object to be added as a new entry.

@return True if addition was successful, or false if not. \*/

virtual bool add(const ItemType& newEntry) = 0;

/\*\* Removes one occurrence of a given entry from this bag,

if possible.

@post If successful, anEntry has been removed from the bag

and the count of items in the bag has decreased by 1.

@param anEntry The entry to be removed.

@return True if removal was successful, or false if not. \*/

virtual bool remove(const ItemType& anEntry) = 0;

/\*\* Removes all entries from this bag.

@post Bag contains no items, and the count of items is 0. \*/

virtual void clear() = 0;

/\*\* Counts the number of times a given entry appears in bag.

@param anEntry The entry to be counted.

@return The number of times anEntry appears in the bag. \*/

virtual int getFrequencyOf(const ItemType& anEntry) const = 0;

/\*\* Tests whether this bag contains a given entry.

@param anEntry The entry to locate.

@return True if bag contains anEntry, or false otherwise. \*/

virtual bool contains(const ItemType& anEntry) const = 0;

/\*\* Empties and then f ills a given vector with all entries that

are in this bag.

@return A vector containing all the entries in the bag. \*/

virtual vector<ItemType> toVector() const = 0;

}; // end BagInterface

#endif

/\*\*DoubleNode.h\*\*/

#ifndef Double\_NODE\_

#define Double\_NODE\_

template<class ItemType>

class DoubleNode

{

private:

ItemType item; // A data item

DoubleNode<ItemType>\* next; // Pointer to next node

DoubleNode<ItemType>\* previous; //Point to previous node //new

public:

/\*Default constructor\*/

/\*@pre none

/\*@post intializes default values to data member\*/

DoubleNode();

/\*Constructor to intialize item\*/

/\*@param anItem passed by reference

/\*@pre none

/\*@post intializes the item\*/

DoubleNode(const ItemType& anItem);

/\*Constructor which intializes item, next, and previous\*/

/\*@param anItem passed by reference

/\*@param nextNodePtr passed by reference

/\*@param previousNodePtr passed by reference

/\*@pre none

/\*@post intializes the all three data members\*/

DoubleNode(const ItemType& anItem, DoubleNode<ItemType>\* nextNodePtr, DoubleNode<ItemType>\* previousNodePtr);

/\*Sets item\*/

/\*@param anItem passed by reference

/\*@pre none

/\*@post sets the next item\*/

void setItem(const ItemType& anItem);

/\*Sets next\*/

/\*@param nextNodePtr passed by reference

/\*@pre none

/\*@post sets the next\*/

void setNext(DoubleNode<ItemType>\* nextNodePtr);

/\*Sets previous\*/

/\*@param previousNodePtr passed by reference

/\*@pre none

/\*@post sets the previous\*/

void setPrevious(DoubleNode<ItemType>\* previousNodePtr);

/\*Gets item\*/

/\*@pre none

/\*@post returns the item\*/

ItemType getItem() const ;

/\*Gets next\*/

/\*@pre none

/\*@post returns the next\*/

DoubleNode<ItemType>\* getNext() const ;

/\*Gets previous\*/

/\*@pre none

/\*@post returns the previous\*/

DoubleNode<ItemType>\* getPrevious() const ;

};

#include "DoubleNode.cpp"

#endif

/\*\*DoubleNode.cpp\*\*/

#include "DoubleNode.h"

#include <cstddef>

template<class ItemType>

DoubleNode<ItemType>::DoubleNode() : next(NULL)

{

} // end default constructor

template<class ItemType>

DoubleNode<ItemType>::DoubleNode(const ItemType& anItem) : item(anItem), next(NULL)

{

} // end constructor

template<class ItemType>

DoubleNode<ItemType>::DoubleNode(const ItemType& anItem, DoubleNode<ItemType>\* nextNodePtr, DoubleNode<ItemType>\* previousNodePtr) :

item(anItem), next(nextNodePtr), previous(previousNodePtr)

{

} // end constructor

template<class ItemType>

void DoubleNode<ItemType>::setItem(const ItemType& anItem)

{

item = anItem;

} // end setItem

template<class ItemType>

void DoubleNode<ItemType>::setNext(DoubleNode<ItemType>\* nextNodePtr)

{

next = nextNodePtr;

} // end setNext

template<class ItemType>

void DoubleNode<ItemType>::setPrevious(DoubleNode<ItemType>\* previousNodePtr)

{

previous = previousNodePtr;

} // end setPrevious

template<class ItemType>

ItemType DoubleNode<ItemType>::getItem() const

{

return item;

} // end getItem

template<class ItemType>

DoubleNode<ItemType>\* DoubleNode<ItemType>::getNext() const

{

return next;

} // end getNext

template<class ItemType>

DoubleNode<ItemType>\* DoubleNode<ItemType>::getPrevious() const

{

return previous;

} // end getPrevious

/\*\*DoublyLinkedBag.h\*\*/

#ifndef \_Doubly\_LINKED\_BAG

#define \_Doubly\_LINKED\_BAG

#include "BagInterface.h"

#include "DoubleNode.h"

template<class ItemType>

class DoublyLinkedBag : public BagInterface<ItemType>

{

private:

DoubleNode<ItemType>\* headPtr; // Pointer to first node

int itemCount; // Current count of bag items

// Returns either a pointer to the node containing a given entry

// or the null pointer if the entry is not in the bag.

DoubleNode<ItemType>\* getPointerTo(const ItemType& target) const;

public:

/\*Default constructor\*/

/\*@pre none

/\*@post intializes default values to data member\*/

DoublyLinkedBag();

/\*Copy constructor\*/

/\*@param aBag instance of DoublyLinkedBag

/\*@pre none

/\*@post copies the bag into a new object\*/

DoublyLinkedBag(const DoublyLinkedBag<ItemType>& aBag);// Copy constructor

/\*Destructor\*/

/\*@pre none

/\*@post deallocates memory\*/

virtual ~DoublyLinkedBag();// Destructor should be virtual

/\*Member function\*/

/\*@pre none

/\*@post gets the size of the bag\*/

int getCurrentSize() const;

/\*Member function\*/

/\*@pre none

/\*@post checks to see if the bag is empty\*/

bool isEmpty() const;

/\*Member function\*/

/\*@param newEntry

/\*@pre none

/\*@post adds an item to the bag\*/

bool add(const ItemType& newEntry);

/\*Member function\*/

/\*@param anEntry

/\*@pre none

/\*@post removes an entry from the bag\*/

bool remove(const ItemType& anEntry);

/\*Member function\*/

/\*@pre none

/\*@post clears the bag\*/

void clear();

/\*Member function\*/

/\*@param anEntry

/\*@pre none

/\*@post checks to see if an item is in the bag\*/

bool contains(const ItemType& anEntry) const;

/\*Member function\*/

/\*@param anEntry

/\*@pre none

/\*@post computes the number of times an item appears in the bag\*/

int getFrequencyOf(const ItemType& anEntry) const;

/\*Member function\*/

/\*@pre none

/\*@post dynamic array able to resize itself\*/

vector<ItemType> toVector() const;

};

#include "DoublyLinkedBag.cpp"

#endif

/\*\*DoublyLinkedBag.cpp\*\*/

#include "DoublyLinkedBag.h"

#include "DoubleNode.h"

#include <cstddef>

template<class ItemType>

DoublyLinkedBag<ItemType>::DoublyLinkedBag() : headPtr(NULL), itemCount(0)

{

} // end default constructor

template<class ItemType>

DoublyLinkedBag<ItemType>::DoublyLinkedBag(const DoublyLinkedBag<ItemType>& aBag)

{

itemCount = aBag.itemCount;

DoubleNode<ItemType>\* origChainPtr = aBag.headPtr; // Points to nodes in original chain

if (origChainPtr == NULL)

headPtr = NULL; // Original bag is empty

else

{

// Copy first node

headPtr = new DoubleNode<ItemType>();

headPtr->setItem(origChainPtr->getItem());

// Copy remaining nodes

DoubleNode<ItemType>\* newChainPtr = headPtr; // Points to last node in new chain

origChainPtr = origChainPtr->getNext(); // Advance original-chain pointer

while (origChainPtr != NULL)

{

// Get next item from original chain

ItemType nextItem = origChainPtr->getItem();

// Create a new node containing the next item

DoubleNode<ItemType>\* newNodePtr = new DoubleNode<ItemType>(nextItem);

// Link new node to end of new chain

newChainPtr->setNext(newNodePtr);

// Advance pointer to new last node

newChainPtr = newChainPtr->getNext();

// Advance original-chain pointer

origChainPtr = origChainPtr->getNext();

} // end while

newChainPtr->setNext(NULL); // Flag end of chain

} // end if

} // end copy constructor

template<class ItemType>

DoublyLinkedBag<ItemType>::~DoublyLinkedBag()

{

clear();

} // end destructor

template<class ItemType>

bool DoublyLinkedBag<ItemType>::isEmpty() const

{

return itemCount == 0;

} // end isEmpty

template<class ItemType>

int DoublyLinkedBag<ItemType>::getCurrentSize() const

{

return itemCount;

} // end getCurrentSize

template<class ItemType>

bool DoublyLinkedBag<ItemType>::add(const ItemType& newEntry)

{

// Add to beginning of chain: new node references rest of chain;

// (headPtr is null if chain is empty)

DoubleNode<ItemType>\* nextNodePtr = new DoubleNode<ItemType>();

nextNodePtr->setItem(newEntry);

nextNodePtr->setNext(headPtr); // New node points to chain

// Node<ItemType>\* nextNodePtr = new Node<ItemType>(newEntry, headPtr); // alternate code

headPtr = nextNodePtr; // New node is now first node

itemCount++;

return true;

} // end add

template<class ItemType>

vector<ItemType> DoublyLinkedBag<ItemType>::toVector() const

{

vector<ItemType> bagContents;

DoubleNode<ItemType>\* curPtr = headPtr;

int counter = 0;

while ((curPtr != NULL) && (counter < itemCount))

{

bagContents.push\_back(curPtr->getItem());

curPtr = curPtr->getNext();

counter++;

} // end while

return bagContents;

} // end toVector

template<class ItemType>

bool DoublyLinkedBag<ItemType>::remove(const ItemType& anEntry)

{

DoubleNode<ItemType>\* entryNodePtr = getPointerTo(anEntry);

bool canRemoveItem = !isEmpty() && (entryNodePtr != NULL);

if (canRemoveItem)

{

// Copy data from first node to located node

entryNodePtr->setItem(headPtr->getItem());

// Delete first node

DoubleNode<ItemType>\* nodeToDeletePtr = headPtr;

headPtr = headPtr->getNext();

// Return node to the system

nodeToDeletePtr->setNext(NULL);

delete nodeToDeletePtr;

nodeToDeletePtr = NULL;

itemCount--;

} // end if

return canRemoveItem;

} // end remove

template<class ItemType>

void DoublyLinkedBag<ItemType>::clear()

{

DoubleNode<ItemType>\* nodeToDeletePtr = headPtr;

while (headPtr != NULL)

{

headPtr = headPtr->getNext();

// Return node to the system

nodeToDeletePtr->setNext(NULL);

delete nodeToDeletePtr;

nodeToDeletePtr = headPtr;

} // end while

// headPtr is NULL; nodeToDeletePtr is NULL

itemCount = 0;

} // end clear

template<class ItemType>

int DoublyLinkedBag<ItemType>::getFrequencyOf(const ItemType& anEntry) const

{

int frequency = 0;

int counter = 0;

DoubleNode<ItemType>\* curPtr = headPtr;

while ((curPtr != NULL) && (counter < itemCount))

{

if (anEntry == curPtr->getItem())

{

frequency++;

} // end if

counter++;

curPtr = curPtr->getNext();

} // end while

return frequency;

} // end getFrequencyOf

template<class ItemType>

bool DoublyLinkedBag<ItemType>::contains(const ItemType& anEntry) const

{

return (getPointerTo(anEntry) != NULL);

} // end contains

template<class ItemType>

DoubleNode<ItemType>\* DoublyLinkedBag<ItemType>::getPointerTo(const ItemType& anEntry) const

{

bool found = false;

DoubleNode<ItemType>\* curPtr = headPtr;

while (!found && (curPtr != NULL))

{

if (anEntry == curPtr->getItem())

found = true;

else

curPtr = curPtr->getNext();

} // end while

return curPtr;

} // end getPointerTo

/\*\*project4.cpp\*\*/

/\*\*Program that checks all the words in file and compares it to a dictionary to see

if the spelling is right

/\*Mohammad Aryan Khan

/\*py9242yg

/\*Instructor: Jie Meichsner

/\*Due date:10/06/2023\*/

#include"DoublyLinkedBag.h"

#include<iostream>

#include<fstream>

#include<sstream>

#include<string>

using namespace std;

// Assuming each word in the dictionary is of type string.

DoublyLinkedBag<string> dictionary;

void loadDictionary(const string &filename) {

ifstream file(filename);

string word;

while (file>>word) {

dictionary.add(word);

}

}

bool isWordInDictionary(const string &word) {

return dictionary.contains(word);

}

int main() {

// Load the dictionary.

loadDictionary("dictionary.txt");

string inputFilename;

cout<<"Enter the name of the file that contains words to check: ";

cin>>inputFilename;

ifstream inputFile(inputFilename);

if (!inputFile.is\_open()) {

cout<<"Failed to open the input file."<<endl;

return 1;

}

cout << "\nThe following words in the file \""<<inputFilename<<"\" are not spelled correctly:\n";

string line, word;

while (getline(inputFile, line)) {

istringstream check(line);

while (check>>word) {

if (!isWordInDictionary(word)) {

cout<<word<<endl;

}

}

}

cout<<"\nThanks for using the spell checker system."<<endl;

return 0;

}

**3. User document**

In this program, we will implement an algorithm that makes the program read words and check if it is spelled correctly. The main program first loads the dictionary which should be located at the same directory as the program. It then asks the user to enter the name of the text file in the form of “filename.txt” which should also be in the same directory. Once the user enters the file name the program displays all the words misspelled in the file.

The project’s name is project4.cpp. It is located at the following directory on CentOS:

/home/STCLOUDSTATE/py9242yg/CSCI301/project4

To compile the program simply enter:

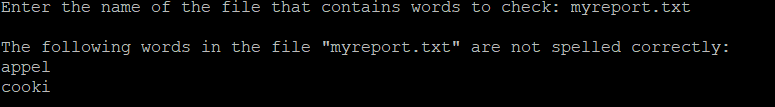
g++ -std=gnu++0x project4.cpp

To run the program, enter:

./a.out

Then respond to the program’s prompts for inputs as specified. The program will continue to prompt the user for specific until the user decides to terminate the program.

For example,



**4. Test data plan**

Valid inputs:

As we saw with the previous example the dictionary text file contained words: apple, cookie, biscuit, and nut.

The myreport text file contained: appel, cooki biscuit, and nut.

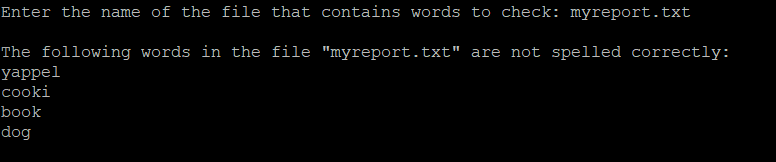
Hence the program output showed that the words misspelled here were appel and cooki.

Boundary inputs:

In the case that a word is included in the myreport text file but not in the dictionary when you run the program, the program does not show an error instead it considers that word as misspelled.

Dictionary text file contains apple, cookie, biscuit, nut, and Book.

Myreport text file contains appel,cooki, biscuit, nut, book, and dog.



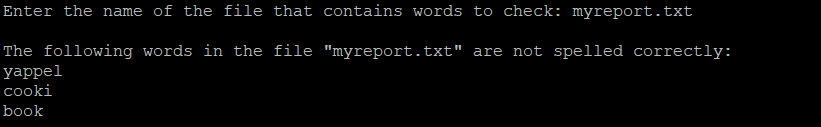
Moreover, the order of the words mentioned in the dictionary and myreport text file does not matter.

Invalid inputs:

The program here is case sensitive. Meaning the program will judge for example “Book” and “book” to be different hence misspelled.

Dictionary text file contains: apple, cookie, biscuit, nut, and Book.

Myreport text file contains: yappel,cooki, biscuit, nut, and book.



Moreover, any input in the dictionary and myreport text that is not a string will throw an error.

**5.Summary**

The C++ program serves as a spell checker and utilizes a custom DoublyLinkedBag class to manage a collection of words efficiently. This DoublyLinkedBag class employs a doubly linked list structure, with each node containing a word (item), references to the next and previous nodes, allowing for operations such as adding, removing, and searching for words. Users can load a dictionary from a text file and check the spelling of words against this dictionary.