# CS512 FUN Project - Spell Checking System


Shengjie Li
Rutgers University
Piscataway, NJ, USA
Email: shengjie.li@rutgers.edu

Weikang Li
Rutgers University
Piscataway, NJ, USA
Email: wl494@scarletmail.rutgers.edu

Junlin Lu
Rutgers University
Piscataway, NJ, USA
Email: jl2364@scarletmail.rutgers.edu



***Abstract*—** This article mainly introduces the achievement of a word spell checking system. This system is a handy tool for every user who wants to check whether there are errors in their text. In the system, we use a less memory-consuming version of trie — ternary search tree for storing the dictionary. We use several methods including edit distance, similarity keys, and n-grams to improve the accuracy of word correction suggestion.


## I. Project Description

The type of this project is implementation of an algorithm not covered in class. There are a lot of ways to detect spelling mistakes and correct the word. This system uses dictionary searching to detect possible correct spelling. Spelling checking plays an important part in our daily life, especially for international students whose mother language is not English, therefore developing a spell-checking system helps a lot and is extremely useful. We use a Trie to store the whole dictionary. More specifically, we use a ternary search tree, which is one kind of implementation of Trie. The correction of a word is a process of finding the most similar word. We achieve it using a combination of several methods like edit distance, similarity keys, and n-grams. Each one in our group is proficient in data structures and algorithms, so it is feasible for us to complete it within a semester. By avoiding some number of calculations and combining several different ways to calculate the difference between two words, we can somehow improve the efficiency and accuracy of this system. This is the novel part of this system. We have not encountered any stumbling block yet. But we think the improvement of the accuracy of the spelling correction would be very hard after a certain point.

The project has four stages: Gathering, Design, Infrastructure Implementation, and User Interface.

### A. *Stage1 - The Requirement Gathering Stage.*

- The general description of this project's deliverables: This project is a spell checking system which is able to detect spelling errors in a text inputed by users.
- The three types of users (grouped by their data access/update rights):
  1) **Administrator**: Administrators are able to manage database, manage user privileges and roles, and check system logs.
  2) **Normal users**: The basic users of this system. They use the system to check whether their words are correct.
  3) **System maintenance personnel**: System maintenance personnel is responsible for the maintenance of the dictionary.
- The user's interaction modes: A user typically uses a keyboard to input text, and uses mouse clicks to interact with this GUI spell checking system. The user just needs to type their text in the text area, click the detect button, then several correction suggestions will be revealed on the right. The user is also able to report their mistakes corresponding to the correct word.
- The real world scenarios:
  - Scenario1 description: A general user, like a student who is learning English, and he wants to know if the words that he writes is correct.
  - System Data Input for Scenario1: Words or sentences
  - Input Data Types for Scenario1: String
  - System Data Output for Scenario1: Recommendations for modification of wrong words
  - Output Data Types for Scenario1: String
  - Scenario2 description: An author who wants to check if all words he used are correct
  - System Data Input for Scenario2: Sentences
  - Input Data Types for Scenario2: String
  - System Data Output for Scenario2: Sentence with highlights of wrong words
  - Output Data Types for Scenario2: String
  - Scenario3 description: A system maintenance person who wants to add some new words
  - System Data Input for Scenario3: Words
  - Input Data Types for Scenario3: String
  - System Data Output for Scenario3: A message of whether the operation was successful or not
  - Output Data Types for Scenario3: String
  - Scenario4 description: A system maintenance person who wants to delete some words or correct some wrong words
  - System Data Input for Scenario4: Words
  - Input Data Types for Scenario4: String
  - System Data Output for Scenario4: A message of whether the operation was successful or not
  - Output Data Types for Scenario4: String
  - Scenario5 description: An administrator wants to add a new system maintenance person

- System Data Input for Scenario5: Username and password
- Input Data Types for Scenario5: String
- System Data Output for Scenario5: A message of whether the operation was successful or not
- Output Data Types for Scenario5: String
- Scenario6 description: An administrator wants to check the log of the system
- System Data Input for Scenario6: Username and password
- Input Data Types for Scenario6: String
- System Data Output for Scenario6: The log of the system
- Output Data Types for Scenario6: String

- Project Time line and Divison of Labor.
  - Stage 1: Before Oct. 26
    - Tasks of Shengjie Li:
      - Format designing using LaTeX
      - Writing an abstract of the project
      - Writing the timeline and division of the project
    - Tasks of Weikang Li:
      - Writing six real-world scenarios of this system
    - Tasks of Junlin Lu:
      - Writing general description of this project
      - Writing 3 types of users of this system
      - Writing the user's interaction modes of this system
  - Stage 2: Before Nov. 9
    - Tasks of Shengjie Li:
      - Writing a brief description of algorithms and data structures
      - Drawing flow diagram
    - Tasks of Weikang Li:
      - Writing a short textual project description
      - Writing flow diagram major constraints
    - Tasks of Junlin Lu:
      - Writing high-level pseudo code
  - Stage 3: Before Nov. 23
    - Tasks of Shengjie Li:
      - Implementing the system
    - Tasks of Weikang Li:
      - Testing and evaluating the system
    - Tasks of Junlin Lu:
      - Writing documentation
  - Stage 4: Before Dec. 7
    - Tasks of Shengjie Li:
      - Designing a GUI
    - Tasks of Weikang Li:
      - Writing a project report
    - Tasks of Junlin Lu:
      - Preparing for a power point presentation

### *B. Stage2 - The Design Stage.*

- Short Textual Project Description.
  First, users input text(String), then we get the input and divide the input into words.
  For each word, we firstly check if this word exists in our dictionary. If this word is in our dictionary, then this word is correctly spelled. If we cannot find this word in our dictionary, then this word is wrong.
  For each wrong word, we first modify this word 3 times at most, if the modified word can be found in our dictionary, we give users a suggestion of modification. If we do not find any modified words within edit distance of 3 in our dictionary, we will tell users that we could not give any suggestion with respect to this word, and they should think about this word again carefully.
  Overall average time complexity: $O(\log n)$ for each operation, including lookup, insertion and deletion, where $n$ stands for the number of nodes in the ternary search tree.
  Overall space complexity: $O(n)$, where $n$ stands for the number of nodes in the ternary search tree, which is also the number of prefixes. E.g. there are 6 prefixes for word 'trie' and 'tree'('t', 'tr', 'tre', 'tree', 'tri', 'trie').

- Flow Diagram.

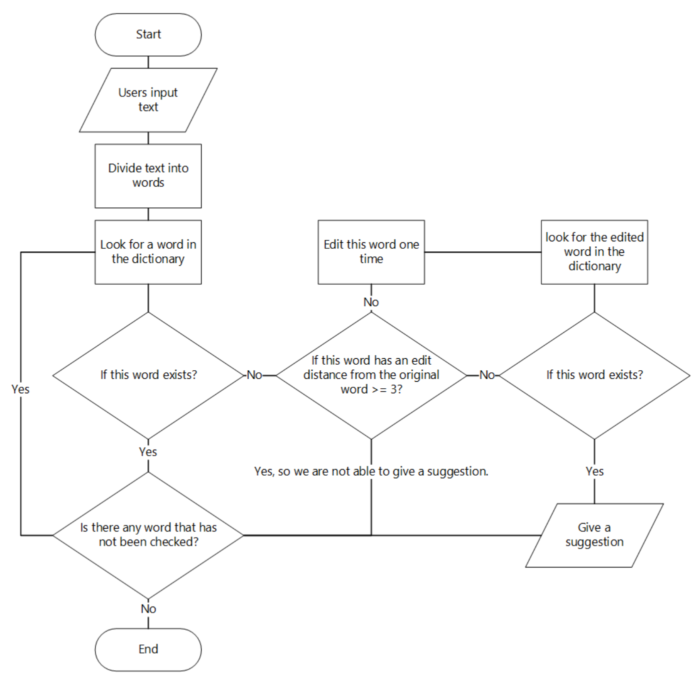



Fig. 1. Flow Diagram

- High Level Pseudo Code System Description.
  Pseudo codes are shown as follows.

**Algorithm 1:** checkWord

```
Data: Inputed text
Result: Suggestions
split the input into words;
suggestions = an empty list of strings;
foreach word in words do
    suggestList = [];
    if not checkDictionary(word) then
        for i from 1 to 3 do
            if len(suggestList) > 5 then
                break;
            end
            editOneList = editDisOne(word);
            foreach candidate in editOneList do
                if len(suggestList) > 5 then
                    break;
                end
                if checkDictionary (candidate) then
                    suggestList.append(candidate);
                    suggestions[word] = suggestList;
                end
            end
        end
    end
end
return suggestions;
```

**Algorithm 2:** checkDictionary

```
Data: Word to be checked
Result: If this word is in dictionary
if word is in the ternary search tree then
    return true;
else
    return false;
end
```

**Algorithm 3:** editDisOne

```
Data: Word to be edited
Result: A list of edited words
splits = an empty list of pairs;
deletes = an empty list of strings;
transposes = an empty list of strings;
replaces = an empty list of strings;
inserts = an empty list of strings;
result = an empty list of strings;
for i from 1 to len(word) - 1 do
    splits.append([word(1, i), word(i + 1, len(word))]);
end
foreach pair [a, b] in splits do
    if len(b) > 1 then
        deletes.append(a + b(2, len(b)));
    end
    if len(b) > 2 then
        secondhalf = b(2, 2) + b(1, 1) + b(3, len(b));
        transposes.append(a + secondhalf);
    end
    if len(b) > 1 then
        foreach letter in alphabet do
            replaces.append(a + letter + b(2, len(b)));
        end
    end
    foreach letter in alphabet do
        replaces.append(a + letter + b(2, len(b)));
    end
end
put all candidates in deletes, transposes replaces and
  inserts into result;
return result;
```

- Algorithms and Data Structures.
  In this project, we are using ternary search tree, which is a type of trie(or prefix tree). Compared to the 26-ary trie and binary tree, it has at most three children. So it's more space-efficient than trie.
  Each node in ternary search tree stores a character, an indicator and up to three pointers pointing its children. The character is the data we stored. The indicator is a boolean value which tells us whether the node is the end of a word. The left child and the right child of the ternary seach tree act as the lowerbound and upperbound of prefix, help us easily find the prefix we actually want. Ternary search tree supports tree operations: Insertion, search and deletion. The time complexity, as stated before, are $O(\log n)$.
  I also want to highlight the search operation. We process the string we want to search letter by letter. Everytime we try to look for only one letter in the ternary search tree. If the letter is equal to the character of current node, we move to the center child of current node and try to process next letter. If it is not, we move to the left or right child depending on the relation between the letter and the character of current node. If all letters have been

processed and we are in a node with indicator true, we could say that we found the string in the ternary search tree. The pseudo code is shown as Algorithm 4.
As Fig. 2 shows, it is a ternary search tree of word 'TERNARY', 'TREE', 'TRIE' and 'SEARCH'.

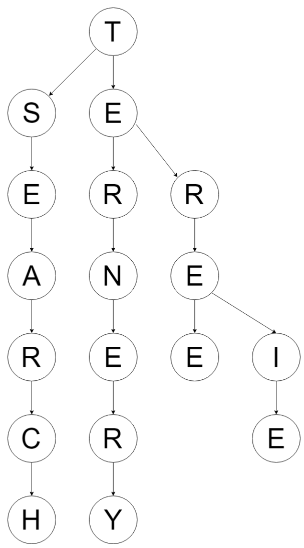



Fig. 2. A ternary search tree

**Algorithm 4:** Search in ternary search tree

**Data:** Word to be searched
**Result:** Whether the search is successful
p ← the root of ternary search tree;
idx ← 0;
**while** *p is not null* **do**
  **if** *word[idx] < p.character* **then**
    p ← p.leftchild;
  **else if** *word[idx] > p.character* **then**
    p ← p.rightchild;
  **else**
    **if** *idx == len(word) - 1 and p.indicator is true* **then**
      return true;
    **end**
    idx ← idx + 1;
    p ← p.centerchild
**end**
return false;

- Flow Diagram Major Constraints. Please insert here the integrity constraints:
  - Input Constraint:
    Description: In our project, we would check the spelling of English words, which means that the input must be English words, no matter if they are right or wrong.
    Justification: All characters of each word inputed should be letters of alphabet.

### C. Stage3 - The Implementation Stage.

We are using Python 3.7 as our programming languange for this project. The programming environment is as follows:

*a) Operating System:* Ubuntu 18.04.1 LTS 64-bit

*b) Processor:* Intel® Core™ i7-8700K CPU

*c) Memory:* 16 GB DDR4-3200 Memory

*d) Graphics Card:* GTX 1070ti

The deliverables for this stage include the following items:

- Sample small data snippet:
  "Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is taht the frist and lsat ltteer be at the rghit pclae. The rset can be a toatl mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe."
- Sample small output:
  **If we use the program's auto correction:** "Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it doesnt matter in what order the ltteers in a word are, the only iprmoetnt thing is that the frist and last letter be at the right place. the set can be a total mess and you can still read it outhit problem. this is abusee the human mind does not read every letter by istle, but the word as a wolve."
  **If we choose the best suggestion in the list:** "Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it doesnt matter in what order the ltteers in a word are, the only iprmoetnt thing is that the frist and last letter be at the right place. The rest can be a total mess and you can still read it outhit problem. this is abusee the human mind does not read every letter by itself, but the word as a wolve."
- Working code:
  **The code of Trie (Ternary Search tree)**

```
class Trie:
  root = None
  def __init__(self):
    self.root = None
  def insert(self, word, rank):
    self.root = insert(self.root,
      word, rank)
  def find(self, word):
    return find(self.root, word)

class Node:
  leftChild = None
  rightChild = None
  centerChild = None
  indicator = False
  character = None
  value = 99999
  def __init__(self, character):
    self.character = character

def insert(node, word, rank):
```

```
    if len(word) == 0:
      return node
    if node is None:
      node = Node(word[0])
    if word[0] < node.character:
      node.leftChild = insert(
        node.leftChild, word, rank)
    elif word[0] > node.character:
      node.rightChild = insert(
        node.rightChild, word, rank)
    else:
      if len(word[1:]) == 0:
        node.indicator = True
        node.value = int(rank)
      else:
        node.centerChild = insert(
          node.centerChild,
          word[1:], rank)
    return node

def find(node, word):
    if node is None or len(word) == 0:
      return (False, 99999)
    if word[0] < node.character:
      return find(node.leftChild, word)
    elif word[0] > node.character:
      return find(node.rightChild, word)
    else:
      if len(word) == 1 and
         node.indicator == True:
        return (True, node.value)
      return find(node.centerChild,
            word[1:])
```

**The code of the checker:**

```
def check_word(trie, word):
    suggest_list = []
    suggest_list_without_rank = []
    print(check_dictionary(trie, word))
    check_result, word_rank =
      check_dictionary(trie, word)
    if check_result == False:
      edited_word_list = word
      for i in range(3):
        if len(suggest_list) >= 8:
          break
        edited_word_list = edit_word(
          edited_word_list, i + 1)
        for edited_word in
          edited_word_list:
          if edited_word in
            suggest_list_without_rank:
            continue
          if len(suggest_list) >= 8:
            break
          check_result, word_rank =
            check_dictionary(trie,
              edited_word)
          if check_result == True:
            suggest_list.append(
              (edited_word, i,
                word_rank))
            suggest_list_without
_rank.append(edited_word)
      suggest_list = sorted(suggest_list,
        key=itemgetter(1, 2))
      if len(suggest_list) == 0:
        suggest_list.append(
          ('No suggestion', 9, 99999))
    return suggest_list

def edit_word(word_or_list, edit_distance):
    if edit_distance <= 1:
      return edit_word_once(word_or_list)
    else:
      edited_list = []
      for edited_word in word_or_list:
        if len(edited_list) > 100000:
          break
        edited_list += edit_word_once(
          edited_word)
      return edited_list

def edit_word_once(word):
    splits = []
    delete_list = []
    traspose_list = []
    replace_list = []
    insert_list = []
    result = []
    for i in range(len(word) + 1):
      splits.append((word[0:i], word[i:]))
    for (a, b) in splits:
      if len(b) >= 1:
        delete_list.append(a + b[1:])
        for c in string.ascii_lowercase:
          replaced_word = a + c + b[1:]
          if word == replaced_word:
            continue
          replace_list.append(
            replaced_word)
      if len(b) >= 2:
        second_half = b[1] + b[0] + b[2:]
        traspose_list.append(a +
          second_half)
      for c in string.ascii_lowercase:
        insert_list.append(a + c + b)
    result = traspose_list + delete_list +
      replace_list + insert_list
    return result
```

```python
def check_dictionary(trie, word):
  return trie.find(word)

def load_dictionary_from_json(filepath):
  with open(filepath) as dictionary_file:
    words = set(dictionary_file.read()
.split())
  return words

def load_dictionary_from_txt(filepath):
  with open(filepath) as dictionary_file:
    words = dictionary_file.readlines()
  words = [word.split()
        for word in words]
  return words

def load_dictionary_to_trie(words, trie):
  for word, rank in words:
    trie.insert(word.strip(),
      rank.strip())

def check_text(text):
  global trie_basic
  global trie_235k
  suggest_list_of_all_words = []
  for i in range(len(text)):
    suggest_list_of_all_words.append([])
    if text[i] in string.punctuation:
      continue
    print('Finding suggestions ' +
      'with respesct to ' + text[i])
    if text[i] == 'i':
      suggest_list_of_all_words[i]
.append('I')
      continue
    suggest_list_of_all_words[i] =
      check_word(trie_235k, text[i])
  print(suggest_list_of_all_words)
  return suggest_list_of_all_words
```

- Demo and sample findings
  - Data size: We have around 240,000 English vocabularies storing in a txt file of 2.5 MB. After fully loaded, the program takes around 300 MB of memory.
  - The accuracy of this program heavily depends on the vocabulary we have, and it is hard to do a perfect English spell checking system using merely Trie.

### D. *Stage4 - User Interface.*

- The initial statement to activate your application with the corresponding initial UI screenshot
  To run this program, we need to use python 3. This can be done by typing `python codes/main.py` in the terminal under the root directory of this project.
  Then the main GUI will pop out as Fig 3 shows, after it finished loading the English dictionary.

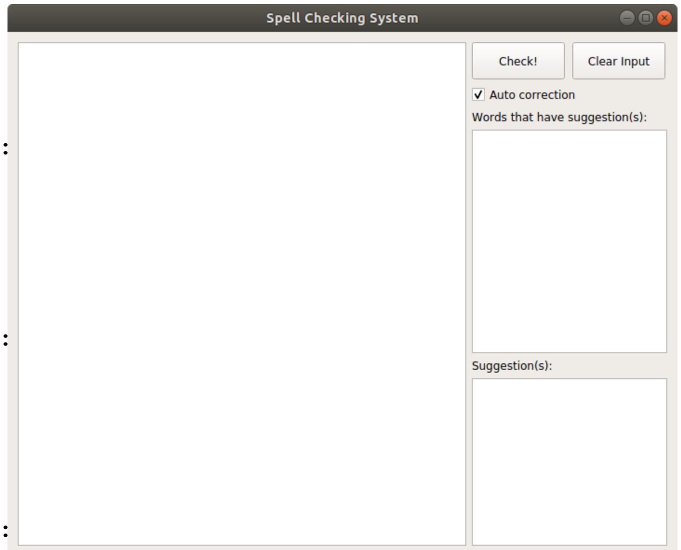
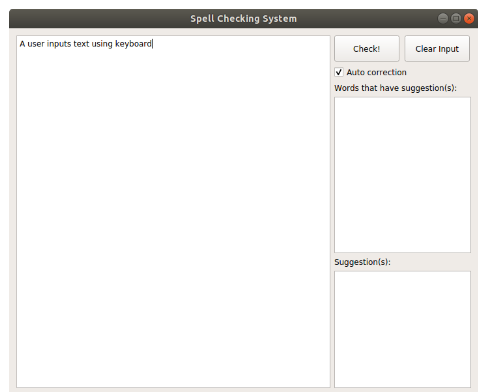
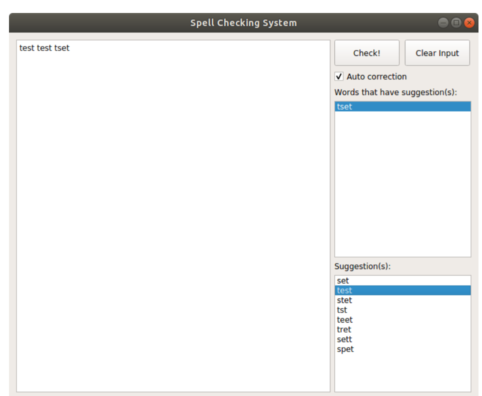



Fig. 3. Main GUI

- Two different sample navigation user paths through the data exemplifying the different modes of interaction and the corresponding screenshots.
  1) A user inputs text in the input area using keyboard.




  2) A user clicks buttons or clicks items in the list using a mouse.




- The error messages popping-up when users access and/or updates are denied (along with explanations and examples):

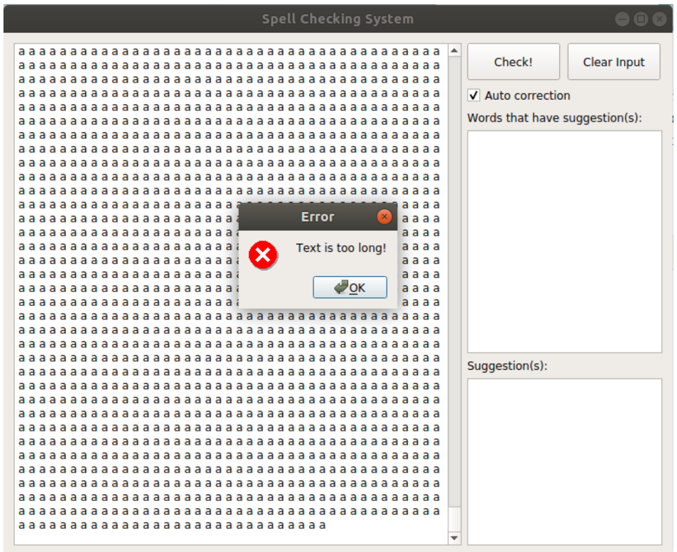



Fig. 4. Error: text is too long.

- The error message: Text is too long!
- The error message explanation (upon which violation it takes place): When a user tries to input more than 50,000 words in the input box, the program will pop out an error message saying that user should not input such a long text.
- The error message example according to user(s) scenario(s): The example is too long to be inserted here, but you can see it from the Fig 4.

- The information messages or results that pop-up in response to user interface events.

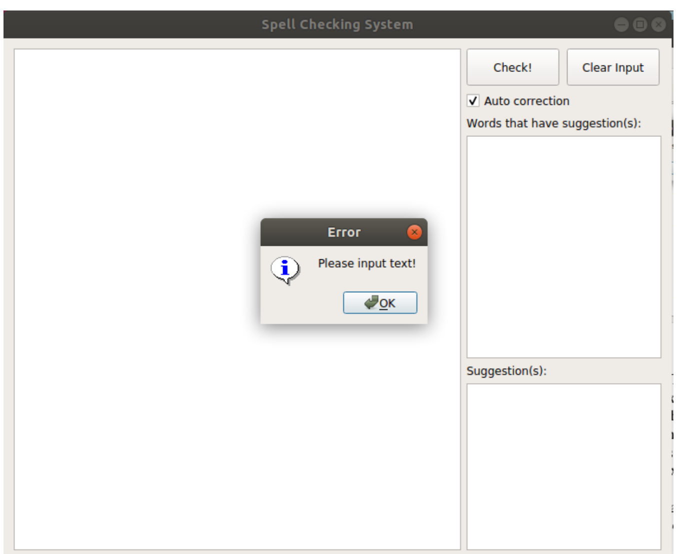



Fig. 5. No text inputed.

- The information message: Please input text!
- The information message explanation and the corresponding event trigger When a user inputs nothing and click 'Check' button, the program will pop out a message telling the user should input something.
- The error message example in response to data range constraints and the coresponding user's scenario

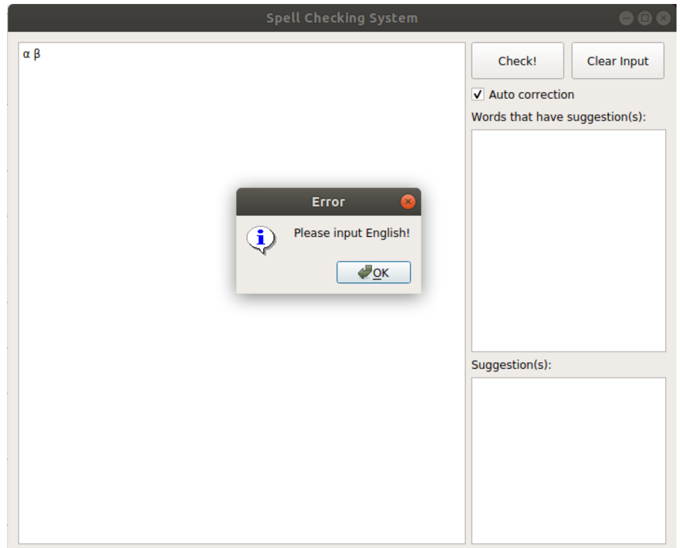



Fig. 6. Please input English!

When a user inputs non-English characters, the program will pop out a message telling the user should input English characters.

The interface mechanisms that activate different views.

- The interface mechanism: After the main interface launches, users can input text in the input area and click buttons. If a user clicks the 'check' button, the program checks if the user's input is valid. If it is valid, the program would try to find errors in the text. If the program found some errors, it would add suggestions into the 'Words that have suggestion(s)' list and 'Suggestion(s)' list. If a user double-clicks any item in the 'Words that have suggestion(s)' list, the program would add suggestions to the 'Suggestion(s)' list. If a user double-clicks any item in the 'Suggestion(s)' list, the text will be corrected by this action. If a user clicks 'Clear Input' button, the input area and both lists will be cleared.

## II. Project Highlights.

- After we finished the project, we did an experiment using errors that are usually made by Wikipedia editors. We randomly chose 300 words from them, and our program made 70% correct corrections. (Data source: https://www.kaggle.com/bittlingmayer/spelling#wikipedia.txt)