Indian Institute of Information Technology and Management – Kerala (IIITM-K)

M.Sc. in Computer Science

Data Structures and Algorithms

(Project 3)

ENTITLED

(An English Sentence Generator using Tree Data Structure)

(Access Code Here)

Submitted By:

Ravi Prakash

Group 1

M.Sc. Computer Science (Cyber Security)

Dated on: December 12th, 2020

ACKNOWLEDGEMENT

It is my privilege to express my sincerest regards to my project coordinator, ***Mr. Alex P James****, for his valuable inputs, able guidance, encouragement, whole-hearted cooperation and constructive criticism throughout the duration of my group project. I deeply express my sincere thanks to* ***Miss Aswani A R*** *&* ***Miss Jessy Scaria*** *for being there for any adverse situation while the development of the project on the topic* ***"An English Sentence Generator using Tree Data Structure"*** *towards the Mini-Project 1 of the subject as the Data Structures and Algorithms. I take this opportunity to thank all my lecturers who have directly or indirectly helped in the project. Last but not least I express my thanks to my whole team for their cooperation and active interaction that build the base of the project.*

Introduction

We were provided with the following problem statements:

Problem Statement

*"Read any **50 sentence** written in English, and rearrange the words in those sentences until **at least 5 new meaningful sentences** are formed. From the generated new sentences, it should be also possible to reconstruct the original sentences. It is also required to efficiently store the 50 sentences for easy access with minimal memory usage.*

*You are required to use a **tree data structure** to implement this. "*

Required parts of submission:

1. Code
2. *Explanation of code*
3. *Discussion on the solution*
4. *Limitations of the solution*
5. *Conclusion*

Discussion on Solutions & Their Limitations

Planning

During the planning phase, we (the whole team) came across many questions and doubts. Some most frequent questions were:

Frequently Asked Questions:

1. How will we read the data?
2. Should we read only and exactly 50 sentences?
3. Do we have to consider only the sentences that are strictly grammatically correct?
4. What is meant by the reconstruction of the sentences?
5. Do we need to prove or verify the reconstruction?
6. Do we also need to handle complex sentences along with the slang?
7. How do we define a meaningful sentence?
8. Do we necessarily need to handle the negative and interrogative sentences?
9. What should be the structure of the nodes of the tree?
10. Which tree will be best to store our sentences?
11. Should we store sentences in the nodes as a whole?
12. Why splitting sentences in several parts can be a good approach to store in a tree?
13. On what basis, we should split every sentence?
14. Does every sentence need to be splitted in a fixed number of sub-parts?
15. How many nodes and the levels should be there in the tree?
16. How will we choose the best approach?

etc..

After finding the answers to these questions, we came up with many different solutions for the given problem.

Proposed Solutions

We came up with many different solutions. The main variation was in:

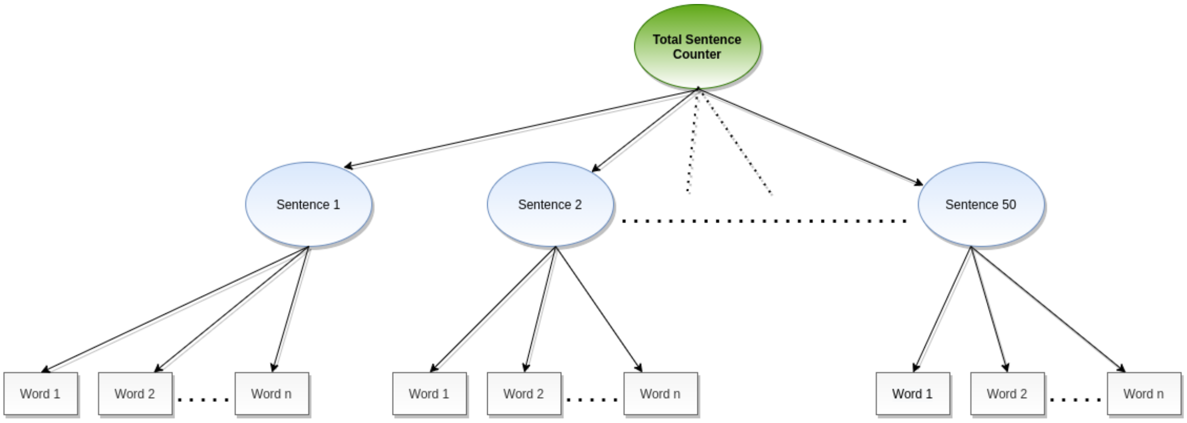
1. Number of children of a Parent
2. What should be kept in Root
3. How many levels/internal nodes can be possible at max.
4. Flexibility of Size of Tree (number of input sentences)
5. Time taken to store & retrieve the data
6. Memory Usage

Following are the 4 solution that I and the whole group brought forward:

1. Each Word Alone
2. **The BTree Storage** (prefered by the majority in the group)
3. Forest of Helping-Verbs
4. **Smart Hybrid Binary cum BTree** (my prefered solution)

Though Solution 3 was not discussed in the group as before that, the group had finalized the approach. Now, here is the description of all of the four approaches:

Each Word Alone



Features

1. Root Node stores a count of the total input sentences
2. There should be 50 internal nodes to store 50 sentences (n, for n sentences), at Level-1 (after root)
3. Leaf nodes stored every sentence again, but wordwise. So, for every sentence, leaves could differ in numbers.
4. **Height** = 3, **Total Nodes** = NA
5. **Time Complexity** to Generate a Sentence = O(n+m) (in best case)

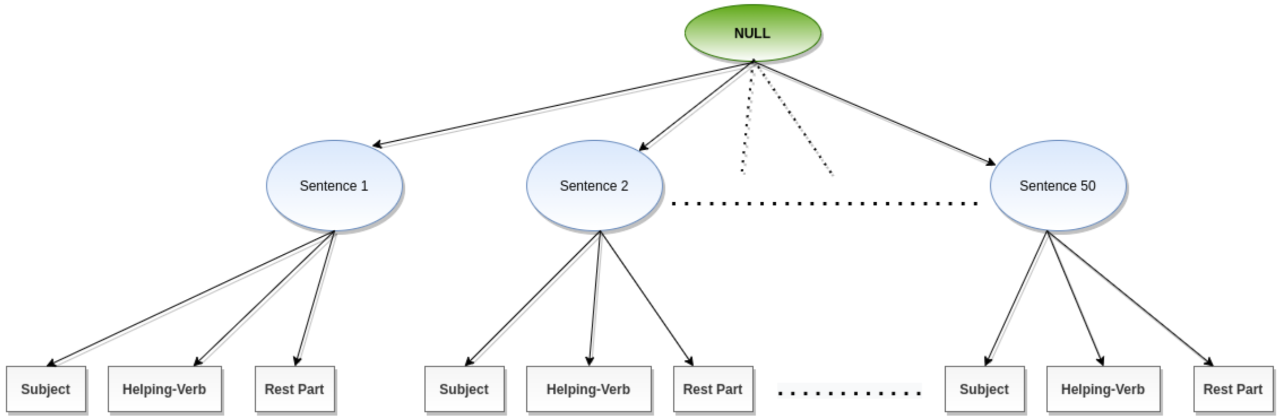
Pros

1. Easy to implement
2. Input sentences can directly be retrieved.
3. All the words are kept separately.
4. Higher number of unique sentences can be generated.

Cons

1. Extra memory Usage by storing same data in internal and leaf node
2. Every sentence needed a separate sub-tree, hence as the size increases,traversing through the tree becomes resource consuming.
3. No mechanism to deal with the duplicate sentences.
4. No mechanisms to ensure whether a sentence is meaningful or not.
5. High Time Complexity because of higher number of edges.

The BTree Storage



*This approach **BTree** and it was chosen as the **final and best solution by the group**.*

.

Features

1. Root Node stores nothing
2. There should be 50 internal nodes to store 50 sentences
3. For every internal node, there should be 3 distinct leaves
    a. To store Subject
    b. To store Helping-Verb (can be blank)
    c. To store rest of the part
4. **Height** = 3, **Total Nodes** = 200 + 1 = **201**
5. **Time Complexity** to Generate a Sentence = O(n+m) here, n = 7, m = 6

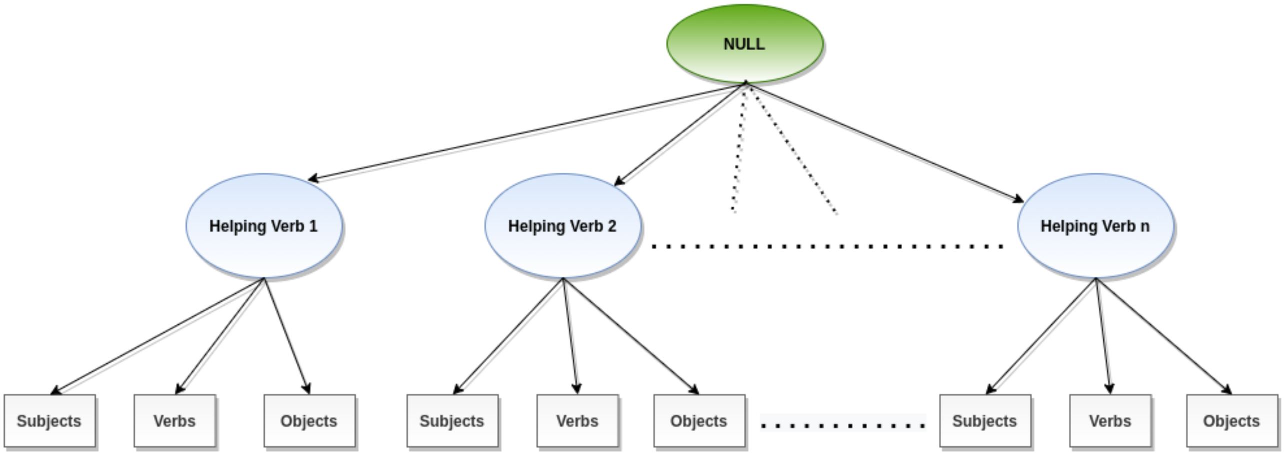
Pros

1. Easy to implement
2. Input sentence can directly be retrieved

Cons

1. Extra memory Usage by storing same data in internal and leaf node
2. Every sentence needed a separate sub-tree, hence as the size increases,traversing through the tree becomes resource consuming.
3. No mechanism to deal with the duplicate entry.
4. In case of generating new sentences with 3 separate parts, 3-different paths need to be traversed till the full depth.
5. No mechanisms to ensure whether a sentence is meaningful or not.
6. High Time Complexity because of higher number of edges.

Forest of Helping-Verbs



Features

1. Data is stored only in the internal nodes.
2. Sentences were stored in separate subtrees, grouped by the helping-verbs.
3. Every leaf consists of an array of similar parts of the sentences.
4. There were maximum 3 leaves for every internal node.
    a. To store Subject
    b. To store Verb (can be blank)
    c. To store Object Field (can be blank)
5. Follows in-order traversal
6. **Height** = 3, **Total Nodes** = NA (depending on the variations found in helping-verbs)
7. **Time Complexity** to Generate a Sentence = O(n)

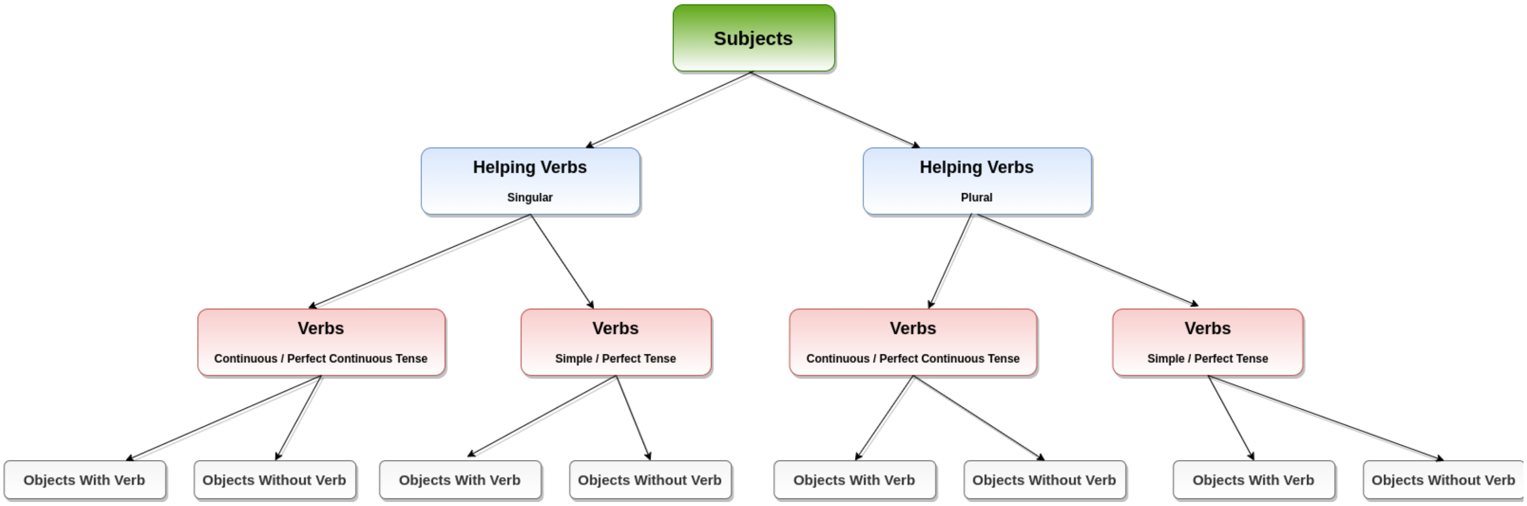
Pros

1. Duplicate entries are rejected.
2. As the number of input sentences grow, the number of internal nodes become fixed.
3. The main data resides in the bottom 2 levels.
4. Due to grouping by helping-verbs, probability to generate meaningful words is higher.
5. Better time complexity than previous approaches.

Cons

1. Complex than Previous two approaches to implement.
2. Can not handle negative sentences perfectly.
3. We can not be very sure about the maximum possible number of internal nodes.

Smart Hybrid Binary cum BTree



I found this method of mine to be the best among the all above four (4) possible approaches. Hence, I personally chose to implement this method as the solution of the given problem.

Features

The tree has a total of **4 levels** and holds data in a Python list. It resembles some of the properties of **BTree** (as nodes hold multiple data) and **Binary-Tree** (as the maximum number of possible children for any parent node is only 2). It not only fulfills the requirements but also provides many other advanced features, making the probability of newly generated sentences to be more meaningful.
The solution makes **extensive use of Tree Data Structures** not only to store the data efficiently but also by covering many other concepts like 3 different traversal techniques at required places. The solution is implemented in such a way that wastage of resources (memory and time) can be avoided as much as possible.

To generate a sentence, it follows *depth-first traversal*.

Height of the tree = 4

Maximum possible nodes = $2^4 - 1 =$ **15**
Time Complexity = **O(log n)** (as in a traditional binary tree)

Pros and Cons of this solution are listed on the next page.

Pros

1. Only **stores** completely **unique input sentences** (and avoids if given input is possible to be generated)
2. **The maximum number of sentences** that can be accommodated are very **flexible**.
3. Makes **minimal memory** usage.
4. Tree can have a maximum height of 4.
5. The maximum number of nodes (including root) can be 15.
6. Best time complexity.
7. Traversing on a single path can provide a meaningful sentence, Making the Time complexity very low.
8. It can easily **detect various types of sentences** and **classify them accordingly**. Viz.
   a. Tense of the sentence
   b. Singular/Plural
   c. Abbreviations
   d. Negation in Sentences etc..

Cons

1. Does not take **interrogative sentences** into an account.
2. Few of the leaves can have a **blank list**.
3. **Complex to implement**, as compared to the initial two approaches.

Traversal Techniques

The program implements three different ways of traversal for different purposes.

1. **Depth-First Traversal** (generating a sentence)
2. **Post-Order** (for counting the total possible unique sentences and printing the tree)
3. **Breadth-First Traversal** (to check if a sentence can be reconstructed or not)

   Although while generating a sentence, it follows only one path from the root (top) to leaf (bottom), only once.

Because of having only **4-levels** and single path depth-first traversal, the **time complexity to generate a sentence** is found to be **O(log n)**.

1.

Explanation of Code

Classes Used

We have splitted our finally implemented program in 3 Python classes:

Structure of the sentenceSplitter -

Parameters:	**Datatype:**
1. __articles	1. List of the articles
2. __pluralPronouns	2. List of plural pronouns
3. __singleIPronouns	3. List of singular pronouns
4. __conjunctions	4. List of common conjunctions
5. __allHelpingVerbs	5. List of helping verbs (H.V.)
6. __firstPriority	6. Most common H.V.
7. __secondPriority	7. Second most common H.V.
8. __thirdPriority	8. Third most common H.V. that are often used along with other helping verbs too
9. __forthPriority	9. Supplements to the H.V.
(these all have constant values)	
Methods:	**Significance:**
1. sentenceSplitter()	1. Constructure
Private Methods	
1. __hasWordsFrom(l1, l2)	1. Loads provided images
2. __removeBlanks(wordList)	2. Removes unnecessary blank spaces
3. __commonPresentPast(sen, HV)	3. - 6. Classifies the sentences among present/past/future and returns a list ['Sub', 'H.V.', 'Verb', 'Obj']
4. __perfectPresentPast(sen, HV)	
5. __futureTense(sen, HV)	
6. __someContinuousPlus(sen, HV)	
7. __splitSimple(sentence)	7. Detects Simple present/past sentences which don't have H.V.
8. __hasSinglarForm(wordList)	8. Classifies sentences among singular/plural
Public Methods	
1. initCleaning(sentence)	1. Lowercase conversion and blank-space removal
2. resolveAbbr(sentence)	2. extends abbriviations
3. removeStops(sentence)	3. Removes end dots
4. removeInterjections(sentence)	4. Removes interjections
5. multipleSentences(sentence)	5. Detects & separates sentences joined by interjections
6. tenseIdentifier(sentence)	6. Implements above all

It splits the fed sentences into a list of four elements : **[Subject, Helping-Verb, Verb, Object]**

Here few of the parts (Helping-Verb, Verb or Object) can be blank.

A few of the examples can be:

1. I am _____ Robin. -----> Verb is absent
2. We are talking _____. -----> Object is absent
3. I ____ go ___. -----> Helping-Verb and Object is absent

An ideal sentence is considered to be like:

“He is walking on the road.”

a. Subject = He
b. Helping-Verb = is
c. Verb = walking
d. Object = on the road

Hence, about the sentence, gives: **[‘He’, ‘is’, ‘walking’, ‘on the road’]**

If any of the four-part(s) is/are absent in the sentences, the **sentence splitter** simply puts a blank string at its place. like:

“I go.” ------> **[‘I’ , ‘’ , ‘go’ , ‘’]**

To store the data in a tree, the **node** had 3 requirements:

1. data
2. left (child)
3. right (child)

The structure of the Node class is described ahead.

Structure of the Tree-Node -

Parameters:	**Datatype:**
1. data	1. List of 2 lists
2. left	2. Node
3. right	3. Node
Methods:	**Significance:**
1. Node()	1. Constructure
2. __thisSenList(sentence)	2. splits large sentences in parts
3. addData(partOfSen, direction)	3. adds data into (nodes of) the tree
4. varifySentence(partOfSen, left)	4. checks if a sentence can be reconstructed from the available data in the tree
5. randomWord()	5. generates a sentence picking different parts
6. printNode()	6. prints data, available in a node
7. possibleSenCount()	7. finds number of all uniques sentences that can be formed
8. splitObj(level)	8. splits all the long sentences of leaf nodes

Every Node in the tree has the above-mentioned parameters and the given functions can be used with them. Multiple similar Nodes collectively form our storage tree.
In the implemented solution, there can be a **maximum of 15 such Nodes**.

The complete solution is implemented using **wordTree** class, which actually above two classes in it.

Structure of the wordTree -

Parameters: 1. thisSplitter 2. treeRoot 3. hierarchyTree 4. __maxSize 5. __currentSize (highlighted part is only for making program more understandable)	**Datatype:** 1. sentenceSplitter 2. Node 3. Node 4. int 5. int
Methods: 1. wordTree() Private Methods 1. __setMaxSize() 2. __updateSize() 3. __isInsertionPossible() 4. __buildTree() 5. __wordJoiner() Public Methods 1. ifTreeExists(node) 2. convertToList() 3. storeInTree() 4. generateSentence() 5. reconstructSentence() 6. printTree() 7. printHierarchy()	**Significance:** 1. Constructure 1. Upper bound for input sentences 2. Updates current number of inputs 3. Checks if total input = __maxSize 4. Creates & returns treeRoot 5. Joins ['Sub', 'H.V.', 'Verb', 'Obj'] to form one sentence 1. Looks for empty tree 2. Splits sentences using thisSplitter object's methods 3. Stores a sentence in tree 4. Generates a sentence from data in tree 5. Verification of input sentences is possible or not 6. Prints tree with data in different levels 7. Prints a sample tree to understand the storage hierarchy

The driver function implements the solution using an object of **wordTree** class.

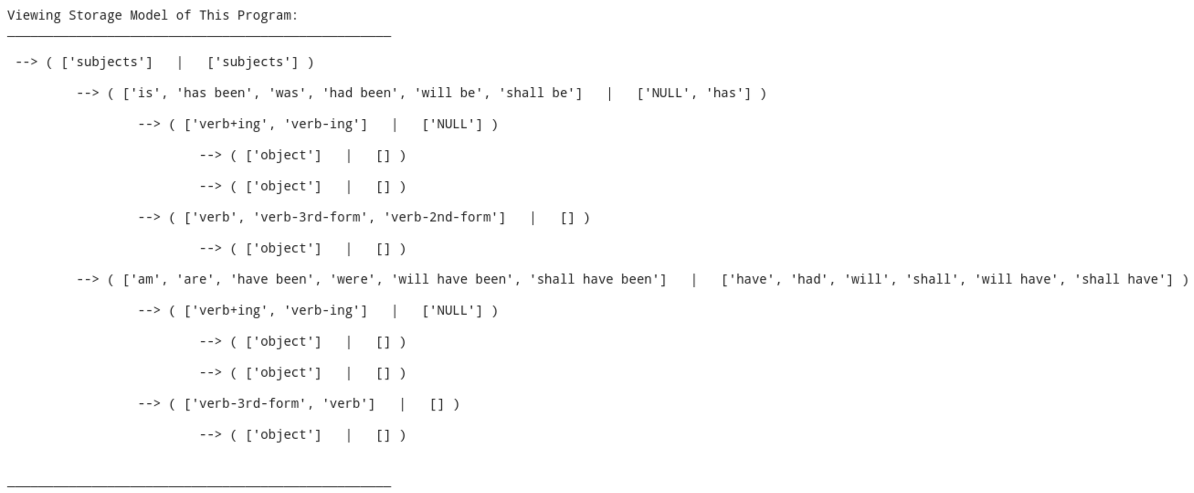
About the Code

The solution is implemented in **Python** language with over **1000 lines of code**.
OOPs concepts:

1. Program consists of 3 Classes (Encapsulation)
2. It requires two functions from two external modules i.e. random (randint()) and re (match())
3. An additional function i.e. clear_output(), from IPython.display, is used to clear the output console only.
4. It provides 5 different to the user.

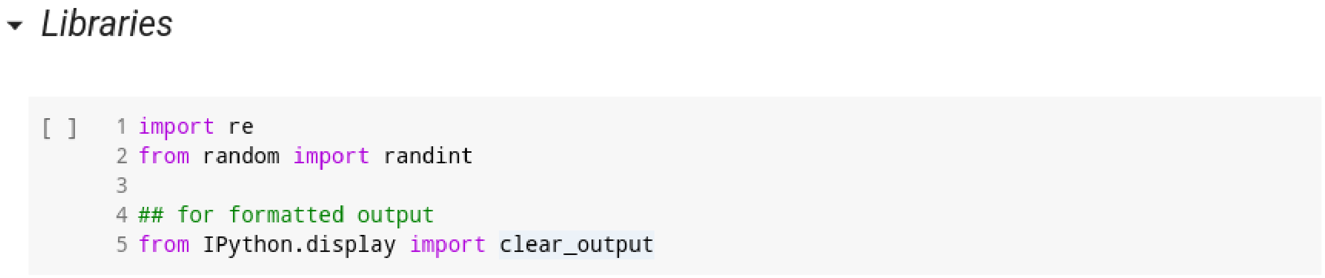
The most challenging part of the project was to detect the tense of the sentence and split it accordingly.

The **storage hierarchy** can be displayed within the program itself, which **makes users understand** how actually the data is being stored in the tree. A sample output for the same looks like:



```
Viewing Storage Model of This Program:
________________________________________________

 --> ( ['subjects']   |   ['subjects'] )

         --> ( ['is', 'has been', 'was', 'had been', 'will be', 'shall be']   |   ['NULL', 'has'] )

                 --> ( ['verb+ing', 'verb-ing']   |   ['NULL'] )

                         --> ( ['object']   |   [] )

                         --> ( ['object']   |   [] )

                 --> ( ['verb', 'verb-3rd-form', 'verb-2nd-form']   |   [] )

                         --> ( ['object']   |   [] )

         --> ( ['am', 'are', 'have been', 'were', 'will have been', 'shall have been']   |   ['have', 'had', 'will', 'shall', 'will have', 'shall have'] )

                 --> ( ['verb+ing', 'verb-ing']   |   ['NULL'] )

                         --> ( ['object']   |   [] )

                         --> ( ['object']   |   [] )

                 --> ( ['verb-3rd-form', 'verb']   |   [] )

                         --> ( ['object']   |   [] )

________________________________________________
```

Libraries Used



Libraries

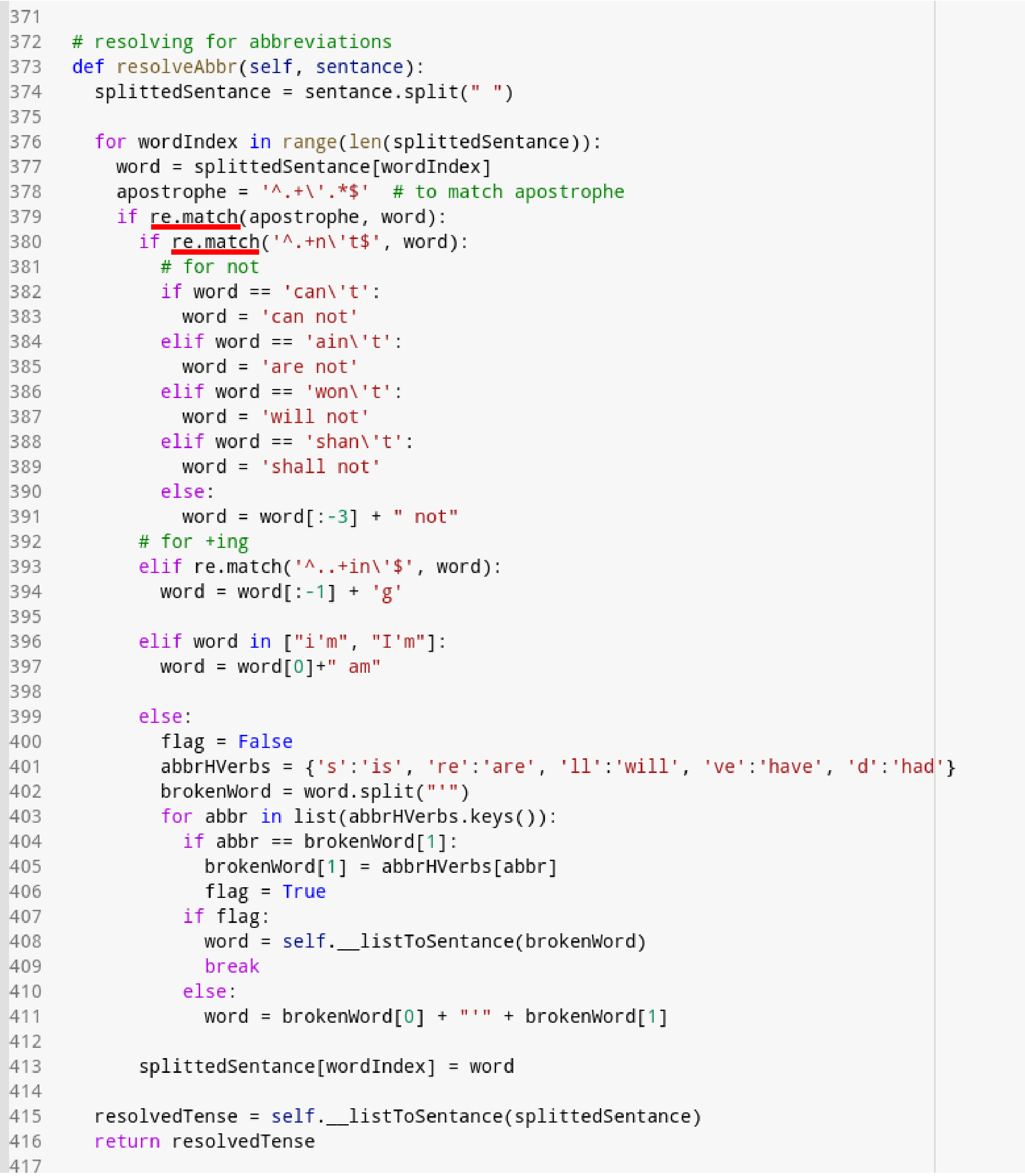
```
import re
from random import randint

## for formatted output
from IPython.display import clear_output
```

Let’s start with the explanations of the code.

1. **re** (regular expression)

The **regular expressions** were required to detect the abbreviations which is very helpful to detect the form and tense of the sentence. Hence, **re** is used here to detect the abbreviated helping verbs.

eg.: *We'll go there.* ---> We will go there. (Simple Future Tense)

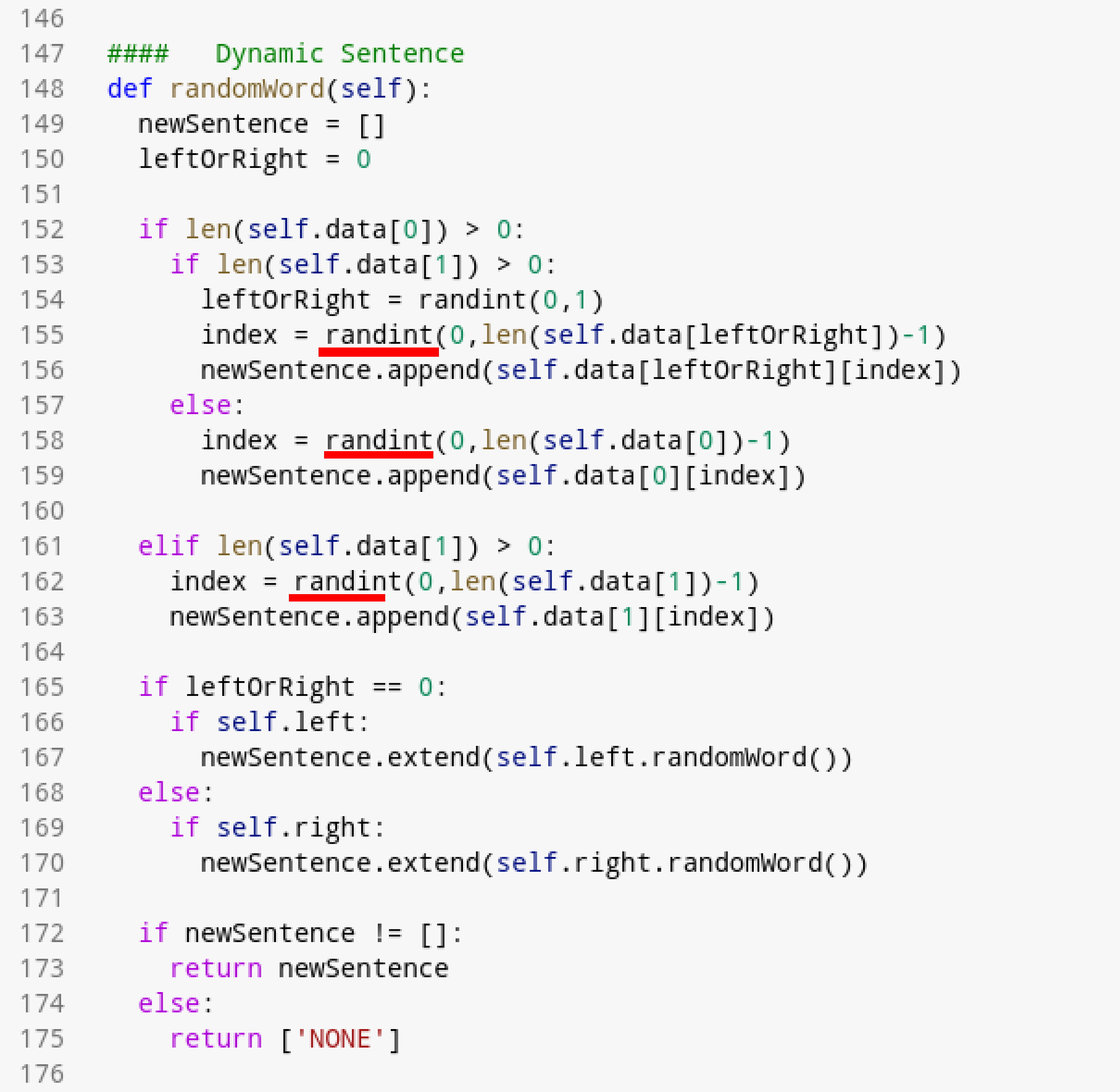


```python

# resolving for abbreviations
def resolveAbbr(self, sentance):
  splittedSentance = sentance.split(" ")

  for wordIndex in range(len(splittedSentance)):
    word = splittedSentance[wordIndex]
    apostrophe = '^.+\'.*$'  # to match apostrophe
    if re.match(apostrophe, word):
      if re.match('^.+n\'t$', word):
        # for not
        if word == 'can\'t':
          word = 'can not'
        elif word == 'ain\'t':
          word = 'are not'
        elif word == 'won\'t':
          word = 'will not'
        elif word == 'shan\'t':
          word = 'shall not'
        else:
          word = word[:-3] + " not"
      # for +ing
      elif re.match('^..+in\'$', word):
        word = word[:-1] + 'g'

      elif word in ["i'm", "I'm"]:
        word = word[0]+" am"

      else:
        flag = False
        abbrHVerbs = {'s':'is', 're':'are', 'll':'will', 've':'have', 'd':'had'}
        brokenWord = word.split("'")
        for abbr in list(abbrHVerbs.keys()):
          if abbr == brokenWord[1]:
            brokenWord[1] = abbrHVerbs[abbr]
            flag = True
          if flag:
            word = self.__listToSentance(brokenWord)
            break
          else:
            word = brokenWord[0] + "'" + brokenWord[1]

      splittedSentance[wordIndex] = word

  resolvedTense = self.__listToSentance(splittedSentance)
  return resolvedTense

```

(a method of ***sentenceSplitter*** *class)*

2. **random** (to generate a random index)

To generate new sentences by the composition of the word from different fed sentences, we picked those words randomly. Different parts of sentences (like subject, verbs etc..) were stored at different levels of the tree, we needed to generate a random index to pick one word from each pool of different parts.
To get a random index, **randint** function of a random library is used.

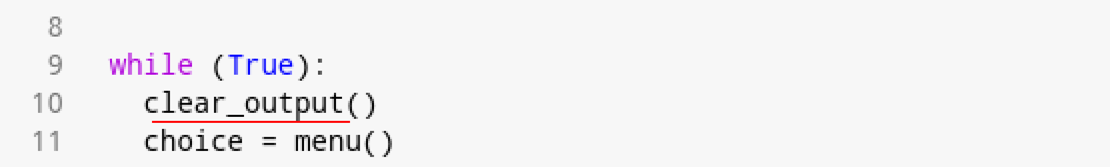


```python

####   Dynamic Sentence
def randomWord(self):
  newSentence = []
  leftOrRight = 0

  if len(self.data[0]) > 0:
    if len(self.data[1]) > 0:
      leftOrRight = randint(0,1)
      index = randint(0,len(self.data[leftOrRight])-1)
      newSentence.append(self.data[leftOrRight][index])
    else:
      index = randint(0,len(self.data[0])-1)
      newSentence.append(self.data[0][index])

  elif len(self.data[1]) > 0:
    index = randint(0,len(self.data[1])-1)
    newSentence.append(self.data[1][index])

  if leftOrRight == 0:
    if self.left:
      newSentence.extend(self.left.randomWord())
  else:
    if self.right:
      newSentence.extend(self.right.randomWord())

  if newSentence != []:
    return newSentence
  else:
    return ['NONE']

```

*(a method of **Node** class)*

3. **IPython.display** (for formatting the console output)

clear_output function is used to clean the output from the console as soon as the Menu reappears. Although, this function does not make any impact on the main program.

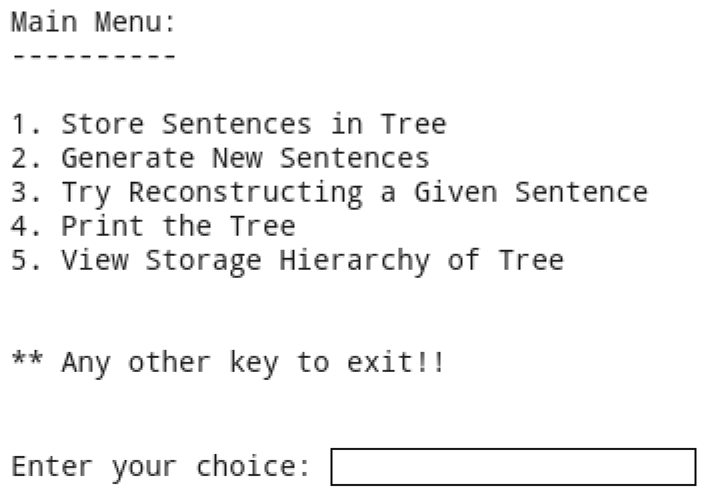


```python

while (True):
  clear_output()
  choice = menu()
```

*(a snippet from the **Driver Function**)*

Main-Menu

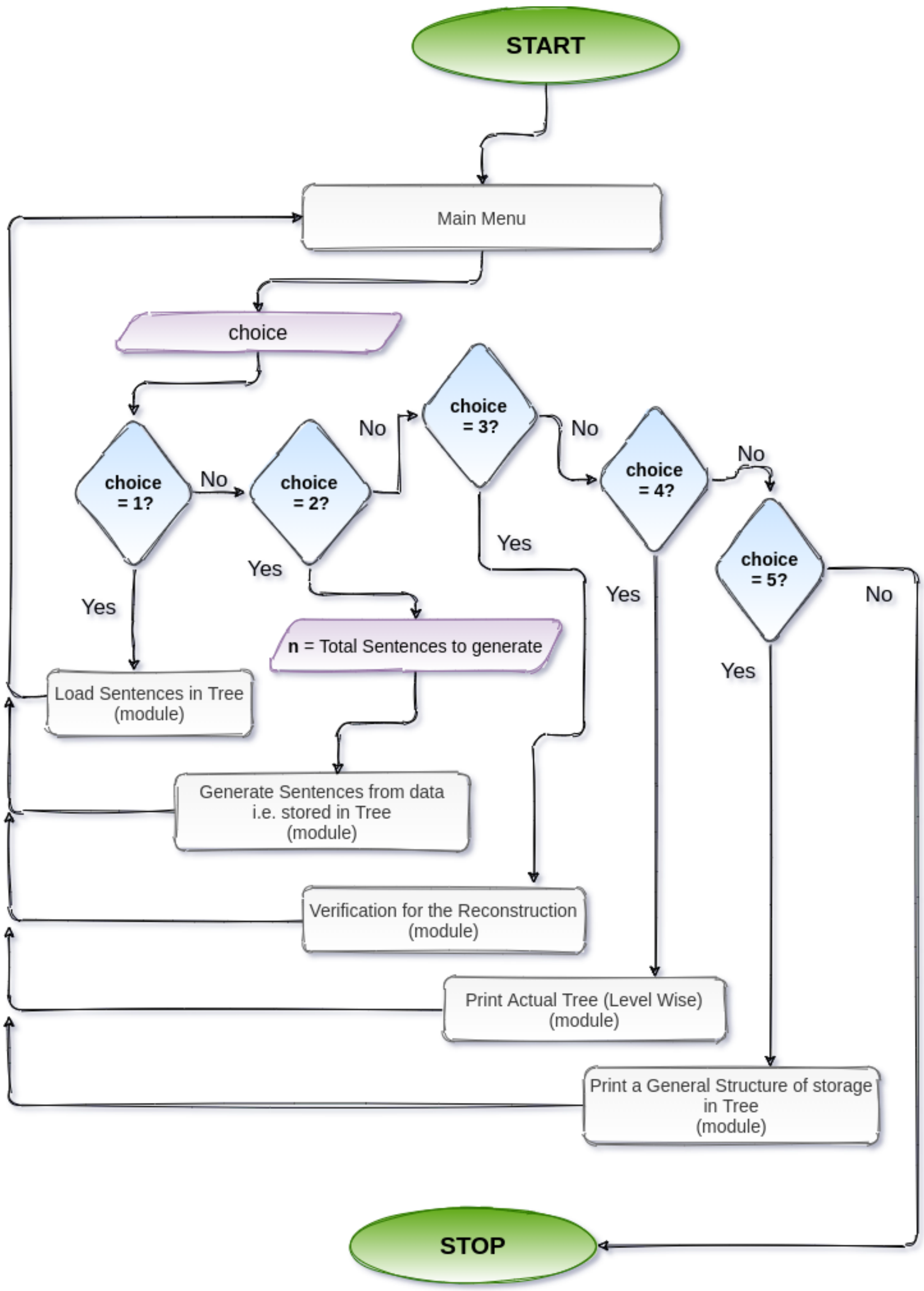
The program greets with the following Main-Menu:



```
Main Menu:
----------

1. Store Sentences in Tree
2. Generate New Sentences
3. Try Reconstructing a Given Sentence
4. Print the Tree
5. View Storage Hierarchy of Tree


** Any other key to exit!!


Enter your choice:
```

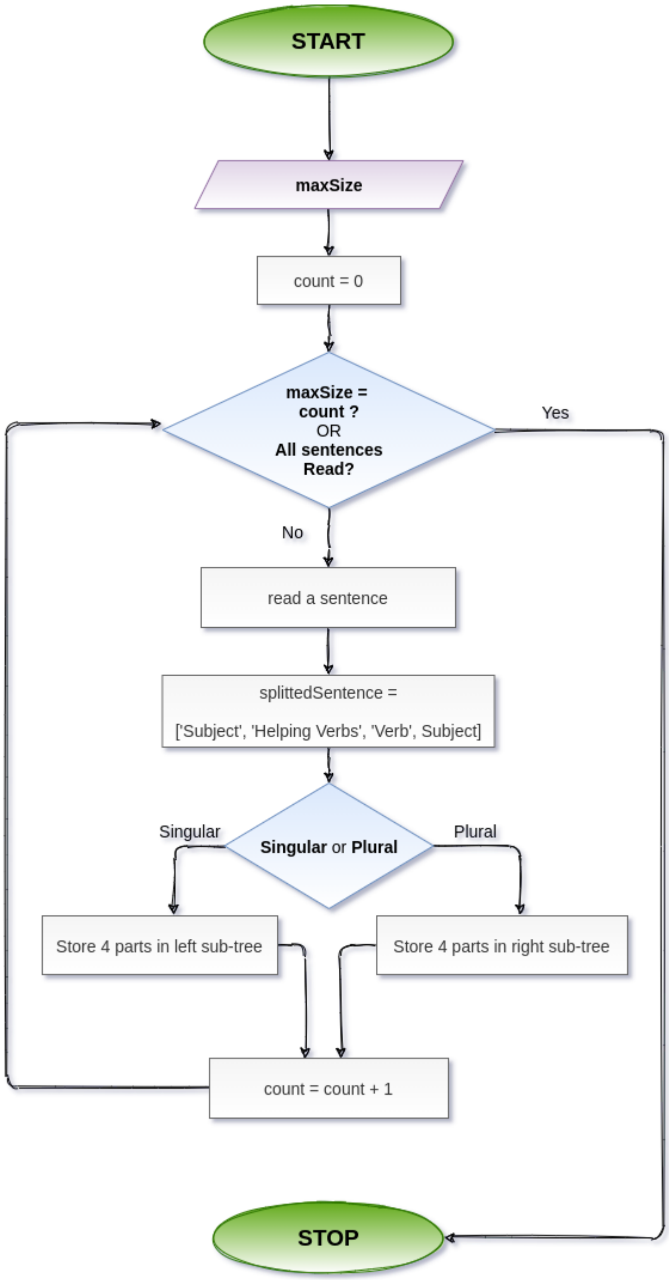
Here, the explanation of the implementation of each of the above-provided features is provided.

Program Flow


START
Main Menu
choice
choice = 1?
No
choice = 2?
No
choice = 3?
No
choice = 4?
No
choice = 5?
No
Yes
Yes
Yes
Yes
Yes
Load Sentences in Tree (module)
n = Total Sentences to generate
Generate Sentences from data i.e. stored in Tree (module)
Verification for the Reconstruction (module)
Print Actual Tree (Level Wise) (module)
Print a General Structure of storage in Tree (module)
STOP

Storing The Sentences in Tree

1. Loading/Storing Sentences in Tree -


START
maxSize
count = 0
maxSize = count ? OR All sentences Read?
Yes
No
read a sentence
splittedSentence = ['Subject', 'Helping Verbs', 'Verb', Subject]
Singular
Singular or Plural
Plural
Store 4 parts in left sub-tree
Store 4 parts in right sub-tree
count = count + 1
STOP

Explanation

1. Initially, the maximum storage limit of the tree is set for the tree as **maxSize**

2. Then sentences are read and **stored** in the tree in a **sequential order**

3. Every sentence is stored in parts.

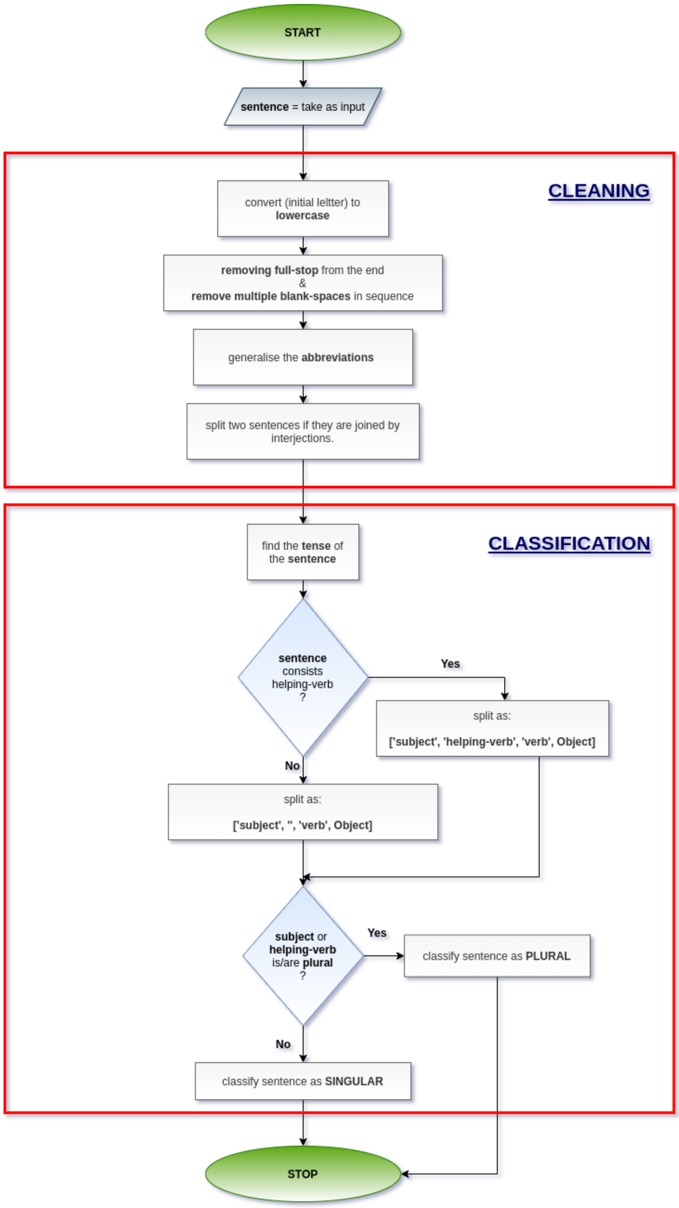
4. Sentences are split into 4 parts before storing into the tree.

5. Parts of the sentence:
   a) *Subject*
   b) *Helping Verb*
   c) *Verb (main)*
   d) *Object*

6. All these 4 parts are stored at 4 different levels of the tree.

7. So we can divide the whole tree in 4-zones:
   a) *Subject Zone* ------------> **Level-1**
   b) *Helping Verb Zone* -----> **Level-2**
   c) *Verb Zone (main)* -------> **Level-3**
   d) *Object Zone* --------------> **Level-4**

2. Splitting a Sentence - (the most challenging part)


START
sentence = take as input
CLEANING
convert (initial leltter) to lowercase
removing full-stop from the end & remove multiple blank-spaces in sequence
generalise the abbreviations
split two sentences if they are joined by interjections.
CLASSIFICATION
find the tense of the sentence
sentence consists helping-verb ?
Yes
split as: ['subject', 'helping-verb', 'verb', Object]
No
split as: ['subject', '', 'verb', Object]
subject or helping-verb is/are plural ?
Yes
classify sentence as PLURAL
No
classify sentence as SINGULAR
STOP

[Subject, Helping-Verb, Verb, Object]

Explanation

1. It was the most challenging thing throughout the project.

2. We needed to split the sentences into a fixed number of parts i.e. **4**

3. Challenges:
   a) Different case (lower upper) etc..
   b) More than 1 space between two words.
   c) Negation
   d) Absence of Helping Verbs
   e) Abbreviation

4. The whole process was divided in 2 parts:
   a) cleaning
   b) classification

5. Solutions:
   a) Conversion to lowercase (for processing)
   b) Reduced spaces in to 1 (between words)
   c) 'Not' was considered as a part of helping-verbs
   d) For Simple present/past sentences initial one or 2 (based on the presence of articles) words were kept as **Subject**
   e) Most common abbreviations were expended

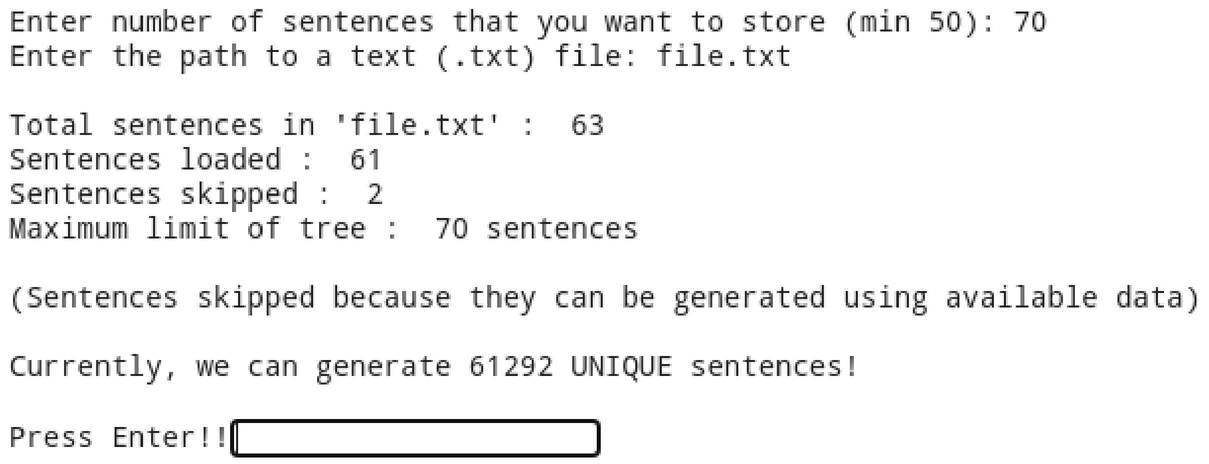
6. Based on the nature of the **Helping Verbs**, sentences were classified as **singular/plural**

7. In case of the absence of the helping verbs, the helping-verb part was kept as **'NULL'** in the tree

*Fully implemented by **sentenceSplitter** class.*

Output

Loading sentences to the tree:

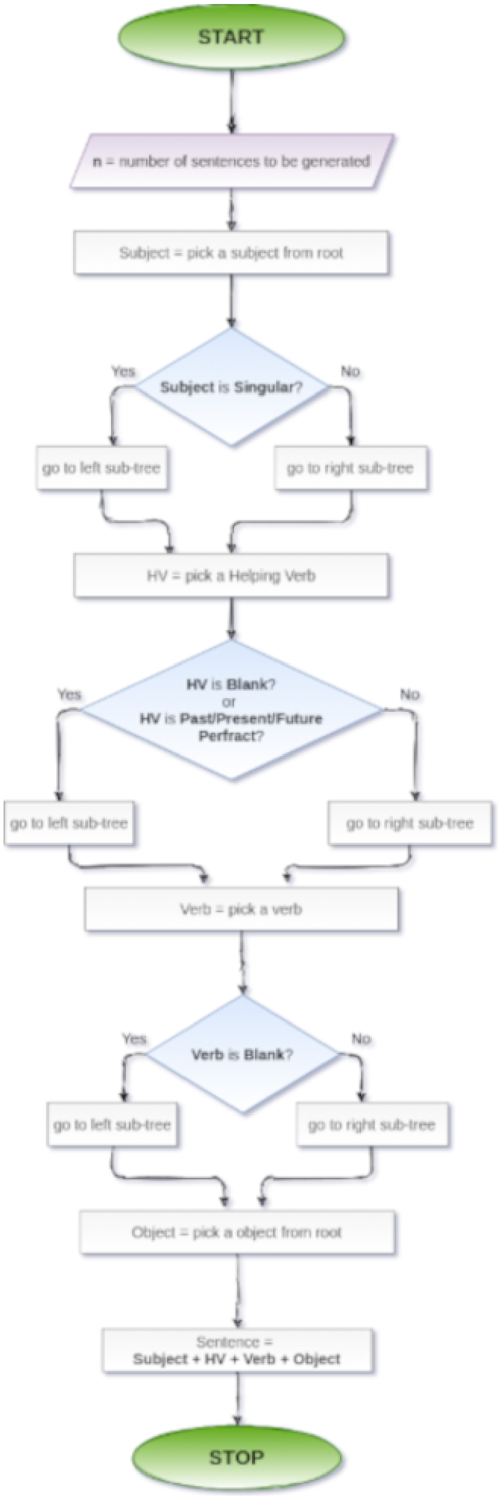


```
Enter number of sentences that you want to store (min 50): 70
Enter the path to a text (.txt) file: file.txt

Total sentences in 'file.txt' :  63
Sentences loaded :  61
Sentences skipped :  2
Maximum limit of tree :  70 sentences

(Sentences skipped because they can be generated using available data)

Currently, we can generate 61292 UNIQUE sentences!

Press Enter!!
```

It’s clear that the solution implements **efficient storage** and **avoids** any type of **duplicity**, which makes sure that the **reconstruction of fed sentences** is possible from generated sentences.
It can also be seen that the solution has a **flexible maximum limit** as per the requirement (though it can not be changed throughout the program, once specified).

In this way, if the tree has enough space, users can feed the sentences from multiple files.

Generating a Meaningful Sentences



Explanation

1. As we split the **sentence** in **4 parts**, this (hybrid) binary tree consists of **4 levels**.

2. We further can bifurcate the whole tree in 2 parts, *at the **root level***.
   a) **Left** -----> Stores **Singular** Sentences
   b) **Right** -----> Stores **Plural** Sentences

Subject is picked from the root.

3. From the second levels, further bifurcation is done on the type of sentence (from **level two**).
   a) **Left** : Continuous / Perfect Continuous
   b) **Right** : Simple / Perfect

Helping-Verbs is picked from Level-2.

4. From the second levels, further bifurcation is done on the type of sentence (from **level three**).
   c) **Left** : Require Some Main Verb
   d) **Right** : Require NO Main Verb

Verb is picked from Level-3.

5. **Object** part is picked from the leaf node i.e. from the **level 4**.

6. Athe the end all four parts are joined.

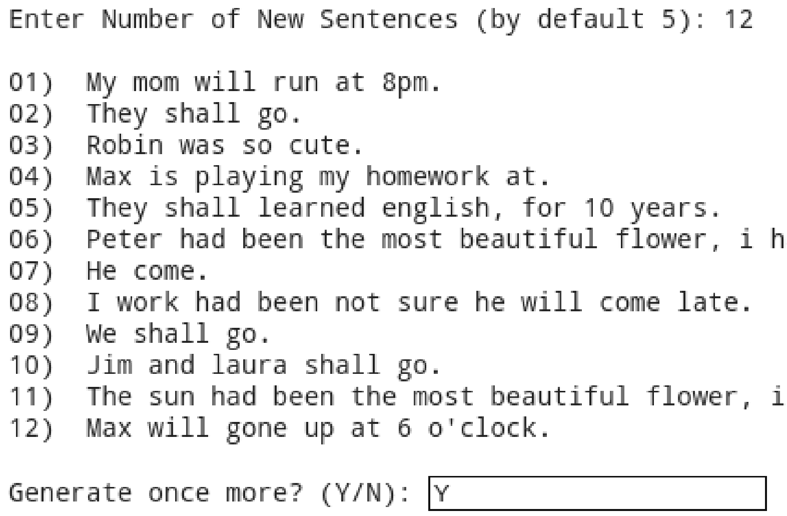
7. The first letter is **capitalized** and a **full stop** is put at the end, and the **new sentence** is returned.

8. Without any much trouble, we **can generate a new sentence** just by travelling from root to the leaf i.e. **in 4-steps**.

Traversal: Depth-First Traversal

Output

This function enables the user to generate the desired number of sentences as many times as he/she wants.

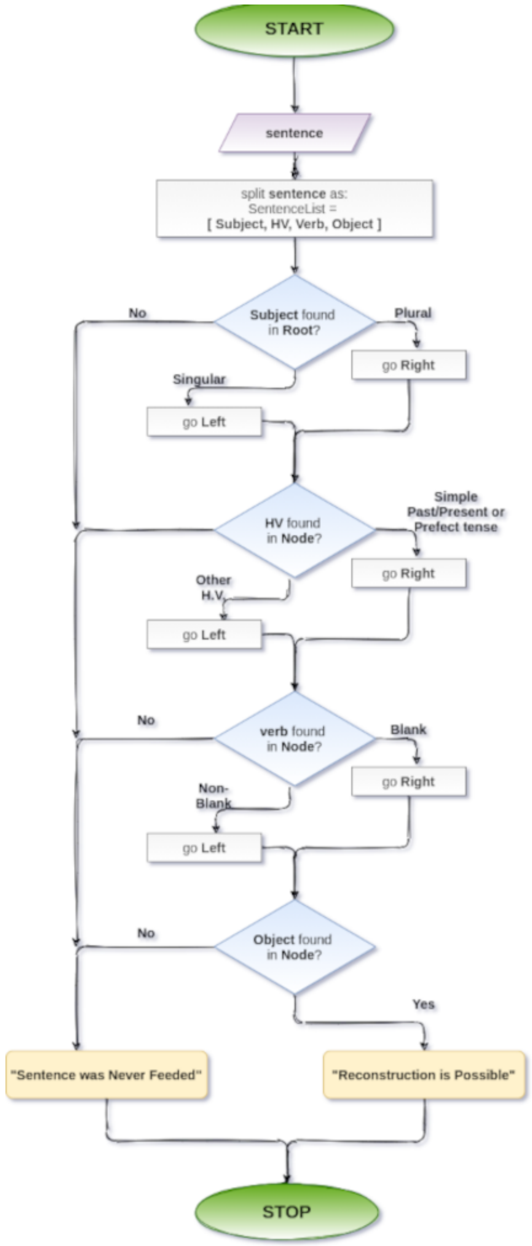


```
Enter Number of New Sentences (by default 5): 12

01)  My mom will run at 8pm.
02)  They shall go.
03)  Robin was so cute.
04)  Max is playing my homework at.
05)  They shall learned english, for 10 years.
06)  Peter had been the most beautiful flower, i have ever seen.
07)  He come.
08)  I work had been not sure he will come late.
09)  We shall go.
10)  Jim and laura shall go.
11)  The sun had been the most beautiful flower, i have ever seen.
12)  Max will gone up at 6 o'clock.

Generate once more? (Y/N): Y
```

It’s clearly visible that the implemented solution produces highly appreciable outputs. We had to generate at least 5 meaningful sentences and here we can see that 7 out of 12 sentences can be accepted.

Although many of the sentences are kind of funny, they can still be considered as meaningful because they are grammatically correct.

Verifying the Capability of Reconstruction



Explanation

1. The provided sentence is split into four (parts) using the code above.

2. Sequentially, every part is picked.

3. The picked part of the sentence is searched in the data, available in the nodes.

4. The search for the part is done only in the node present at the respective level in the tree.

5. Hence, the **element can be searched only in two nodes, at max** (at a time), because of the binary tree.

6. If **everything is matched** to the leaves (4th level), **reconstruction is possible**.

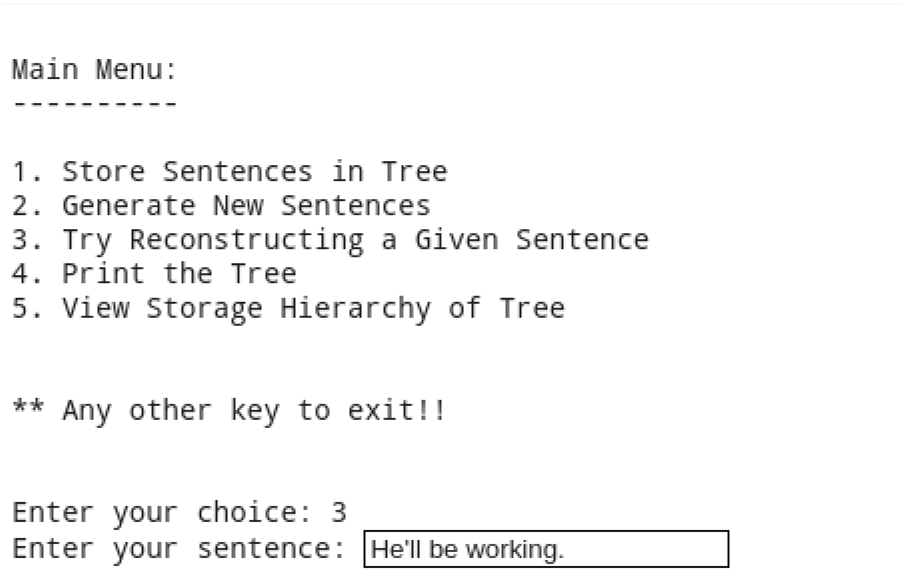
7. If a match is **not found at any single level**, we infer that **reconstruction is NOT possible**.

8. Very helpful in terms of storage and **memory utilization**.

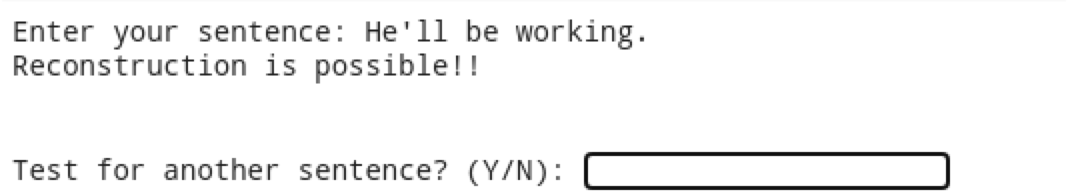
9. ***In other words, it simply tries to pick words from the possible sentences (generated sentences) and checks whether any fed sentence can be reformed or not.***

Traversal: Breadth-First Traversal

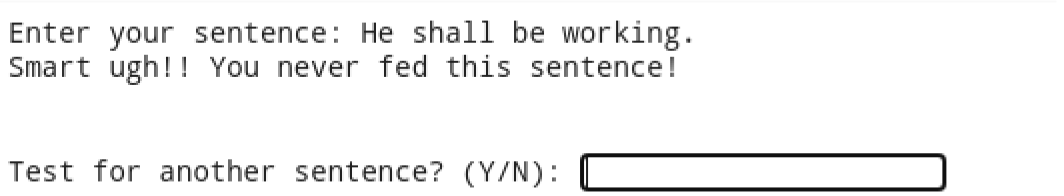
Output



```
Main Menu:
----------

1. Store Sentences in Tree
2. Generate New Sentences
3. Try Reconstructing a Given Sentence
4. Print the Tree
5. View Storage Hierarchy of Tree


** Any other key to exit!!


Enter your choice: 3
Enter your sentence: He'll be working.
```

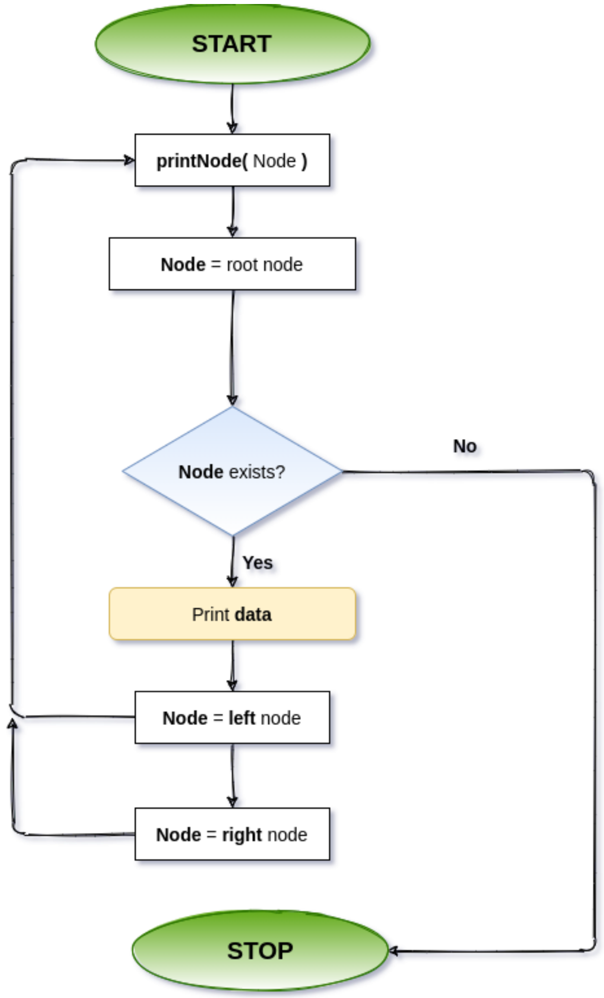


```
Enter your sentence: He'll be working.
Reconstruction is possible!!


Test for another sentence? (Y/N):
```



```
Enter your sentence: He shall be working.
Smart ugh!! You never fed this sentence!


Test for another sentence? (Y/N):
```

Printing the Tree



Explanation

1. It makes fantastic use of recursion.

2. **In-order** traversal is done here.

3. Levels are separated by sufficient space.

4. A **sample tree** can also be printed **to understand** how data is actually stored in the tree..

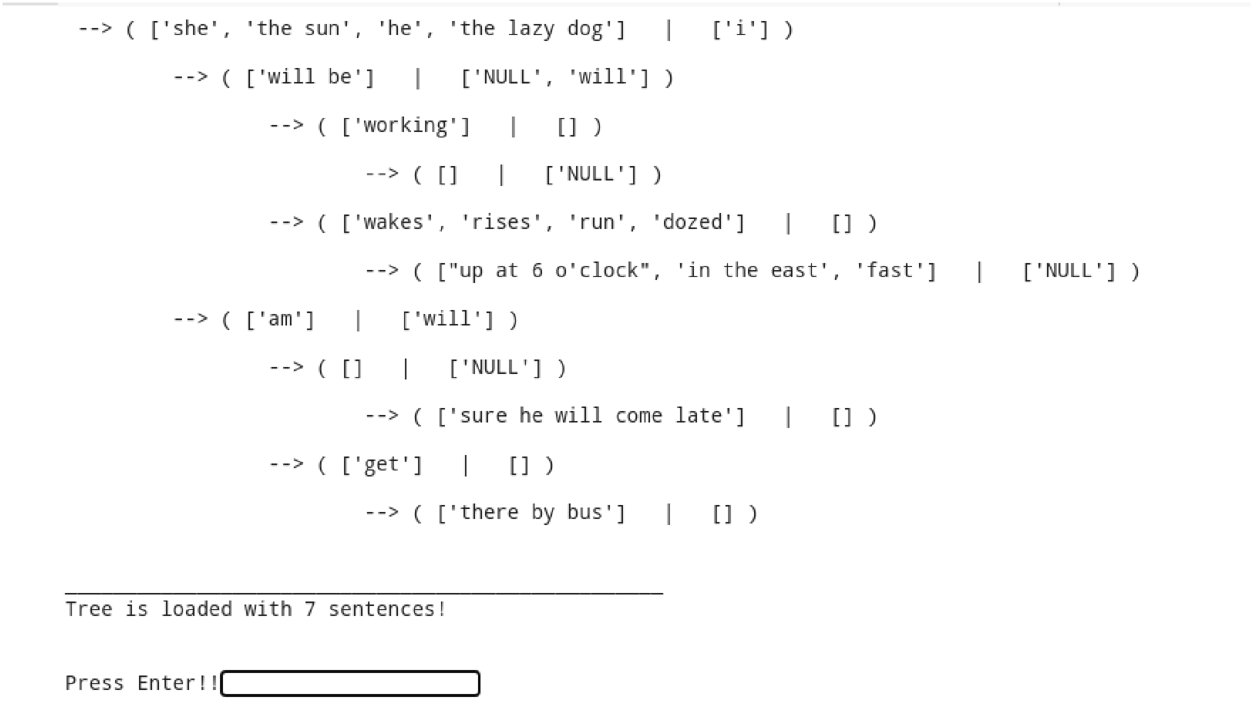
5. the Same method of traversal can be used to give the total **number of** all **possible unique sentences** that can be generated.

6. Time complexity is :

Traversal: Post-Order Traversal

Output

Here, I have tried to print a small tree with just 7 sentences:



```
 --> ( ['she', 'the sun', 'he', 'the lazy dog']   |   ['i'] )

         --> ( ['will be']   |   ['NULL', 'will'] )

                 --> ( ['working']   |   [] )

                         --> ( []   |   ['NULL'] )

                 --> ( ['wakes', 'rises', 'run', 'dozed']   |   [] )

                         --> ( ["up at 6 o'clock", 'in the east', 'fast']   |   ['NULL'] )

         --> ( ['am']   |   ['will'] )

                 --> ( []   |   ['NULL'] )

                         --> ( ['sure he will come late']   |   [] )

                 --> ( ['get']   |   [] )

                         --> ( ['there by bus']   |   [] )

__________________________________________________
Tree is loaded with 7 sentences!

Press Enter!!
```

It can be seen that the tree is printed in a very neat and clean way.
All the levels can be easily seen, clearly.

Conclusion

1. The solution **fulfills all the requirements** without using any text processing library.
2. It can tackle many complex sentences like one that consists of abbreviations, negations, conjunctions, etc..
3. **Fast** and efficient storage with **minimal memory usage**.
4. Awesome user-friendly interface.
5. One **major drawback** is that the program can not handle interrogative sentences.
6. For many times, it does not handle complex sentences (defined above) in all situations.

Exit

Program greets back to the user on exit as below:

```
Good Bye!!
```

As the implemented code is really large enough (**1304 lines of code**) to share in this file, hence I'm providing a link to the Google Colab (Jupyter) Notebook below. Please do visit this link:

A **running instance** of the given solution is available at the following link:

https://github.com/ravi-prakash1907/Data-Structures-and-Algo/blob/main/Submissions/Group%20Projects/Project3/English_Sentence_Generator_by_Ravi.ipynb

Happy Testing!!!