UNIVERSITY
of York

Department of Electronics

Data Structures and Algorithms Assessment

Predictive Text

Y3839090

January 23, 2017

Contents

1 What is a predictive text system and how do they work?

In order to create a predictive text system, their behaviour has to be understood.

The role of a predictive text system (also known as auto-complete or word completion) is to take a partial word and return a list of suggestions. They use some form of dictionary of known words to offer these suggestions.

1.1 How users interact with predictive text systems.

Users interact with predictive text systems via some kind of text input device. Most users interact with some kind of predictive text system on a daily basis on either their PC or mobile device. The UI for both desktop and mobile devices follow the same basic pattern. The user begins by entering text normally via a keyboard or touch screen and once there are a sufficient number of letters for the system to make a reasonable guess (often two letters), the system presents the user with a list of predicted words. The user is then able to press a key to select which of the suggestions they want to use or continue typing and ignore the systems suggestions. The suggested word replaces the partial word they were typing so that they can move on the the next word.

1.2 The behaviour of predictive text systems.

Predictive text system often return result that fall into a few broad categories.

The input. One of the options in a predictive text system is always to keep the partial word that you already have. This is often achieved by returning the input as one of the items in the suggestion list. This behaviour can also happen if the partial word is actually a valid word from the predictive text systems dictionary.

A word prefixed by the partial word. The most common results from predictive text systems are words that are prefixed by the partial word the user has entered. An example of this is that “hel” may return “hel*p*”, “hel*l*” or “hel*lo*”.

Shared Prefixes Words that share a common prefix with the partial word. An example of this is that “applez” may return “apple”, “apples” or“app”.

Others More sophisticated system may also use frequency analysis techniques, lexicographical distance algorithms or even Markov chains to find suggestions. These are out side of the scope of this project and require other types of data sets. The word list available doesn’t contain any frequency data or phrase data.

2 Requirements

Now that a predictive text systems behaviour has been defined, a set of success criteria can be derived.

1. The system must use ISO C99/C11 C (“ANSI C”) .
2. The system must compile with minGW (gcc)
3. The system must compile and execute correctly on the university lab machines in PT108.
4. The system must load a word dictionary from file. *(provided file words.txt)*
5. The system must be capable of adding the loaded words into a data structure to store them whilst the program is running
6. The system must store these words in a space efficient data structure.
7. The system must be able to access these words in a time efficient manner.

8. The system must be able to check to see if a partial word is contained in that data structure.
9. The system must be able to check to see if a partial word that is stored in the data structure is a complete word from the dictionary.
10. The system must be able to make suggestions of words that are prefixed by a partial word.
11. The system must let the user enter a string of text.
12. The system must be able to present the user with a suggestion mode where they will be presented with a set of suggestions which they can chose between.
13. The system should be able to make suggestions of words that share a common prefix with a partial word.
14. The system should let the user select the one of these suggestions.
15. The system should replace the partial word with the selected word.
16. The system should let the user delete characters with backspace.
17. The system could be extended to deal with punctuation.
18. The system could be extended to deal with Capitalisation.
19. The system could present the users with suggestions as they are typing.

3 Data Structures

A predictive text program needs a to be able to access a set of common words. Storing and accessing these words efficiently is one of the challenges in creating a fast predictive text engine. After researching the topic, Tries seem to be the best data structure for this role in this system.[2].

3.1 Trie

3.1.1 What is a trie

The trie is a tree-like data structure that is often used for storing strings (also used for IP addresses). Each node in the trie represents a single letter in a string and each node has a fixed number of child nodes like a binary search tree. Unlike the binary search tree the trie does not have two child nodes, it has one for each letter of the alphabet. This allows the trie nodes to not store their value because their position determines it. A node's children share a common prefix, that prefix is the value of there parent node.

There are many features of tries that make them a good choice for a predictive text system. One of them is that values can be look up by their prefixes. Predictive text systems use a partial word (a prefix) to search for possible full words. So it's important that prefix look-ups are fast and efficient.

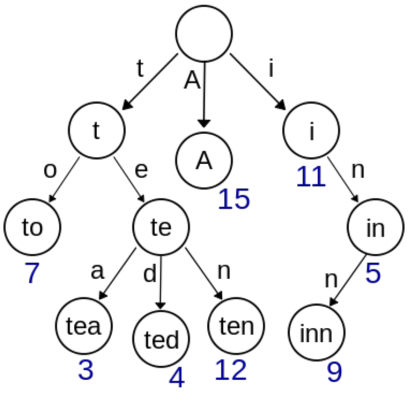


Figure 1: A trie for keys ”A”,”to”, ”tea”, ”ted”, ”ten”, ”i”, ”in”, and ”inn”[1].

3.1.2 Time complexity of a trie

Tries are structure in such a way that they perform look ups rather than searches.

The look up time for a value in a trie is dependant upon $len(w)$, where $len(w)$ is the length of the word to look up[3]. This gives the trie a $O(1)$ search time in regards to n (the number of words). Meaning that searches in a trie do not depend on the number of items in the trie. This is the best big-O time complexity achievable. A predictive text function relies upon look ups. Every time a user presses a key the system will preform multiple look ups, so it is vital that these searches be fast.

The time complexity of deletion in a trie is also constant in regards to n. The time complexity of deletions are $|A| * len(w)$, where $len(w)$ is the length of the word to look up and $|A|$ is the size of the alphabet [3]. This is also a $O(1)$ operation. In the use case as a Predictive text engine deletions will be very rare or non-existent. A simple implementation of predictive text may not even allow the user to delete words form the dictionary.

The time complexity of insertion in a trie is the same as for deletions. Insertion time complexity is $|A|*len(string)$, where $len(string)$ is the length of the word to look up and $|A|$ is the size of the alphabet[3]. This, again, is a $O(1)$ operation in regards to the number of words (n). When used in a predictive text engine most of the insertion into the trie will be made when loading the dictionary from file.

3.1.3 Space complexity of a trie

The space complexity of a trie is $|A|\Sigma_i len(w_i)$ where there are strings w_0 to w_n and $|A|$ is the size of the alphabet [3]. This is saying that the space complexity of a trie is equal to the size of the alphabet by the total length of all the strings. Expressed in therms of n this becomes $|A| * n * l$ where $|A|$ is the size of the alphabet and l is the average length of the words. In this form it is easy to see that the space required to store is governed by a $O(n)$ relationship ship. The big-O time complexity is the same as nearly all data structure [6], But the space complexity of each node in a trie is quite large due to the array of pointer to children.

3.1.4 Space vs Time

The tire data structure sacrifices a good space complexity for time complexity that doesn't depend on the number of strings that it is storing. There are ways to modify the trie data structure to have a smaller space complexity.

The simplest method of doing this is to reduce size if the alphabet. A naive implementation of a trie may use a pointer for each possible value in Extended ASCII character set. This is 256 pointers per node. But most of these character will not be used in normal typing. The only characters that are normally used are a-z and 0-9. This means alphabet size can be reduced from 256 to 36. You may also want to include A-Z and common punctuation but your over all space complexity of each node will still be massively reduced.

There are methods that get rid of the child pointer completely by using lists or hash tables to store the children of each node , but both of these methods increase the time complexity of look ups by a factor of $|A|$. With space complexity reducing by that same factor.

Radix trees or compressed tries could also be used, as well as deterministic acyclic finite state automaton. But these are much more complicated data structures.

3.2 Alternative Data structures

This section details some of the alternative data structures that were be considered for storing the word list. Tries were chosen over these data structures due to haveing better time and space complexity's.

3.2.1 Binary Search Trees vs Tries

Both BST (Binary search trees) and Tries are ordered trees. Unlike tries each node in a BST contains a maximum of 2 children. All of a nodes children on the 'left' side have a smaller value than the node and all the nodes on the “right” have a large value (identical values are normal omitted from BST). BST differ from tries in that each node in a BST stores it's value.

When considering space complexity BST can be shown to have a complexity of $O(n)$ [4]. The big-O space complexity of a trie and BST is the same.

When considering search speed Tries are much faster than BST. For a balanced binary search tree the average time complexity of a search is $O(n)$. This is much worse than the trie's $O(1)$ time complexity.

Predictive text systems preform look ups via prefixes. Binary search trees don't offer any particular advantage in this area whilst tries do (All children of a node are prefixed by that node).

It is also worth noting that BST require a lot of comparisons. Comparisons of two strings is not an atomic operation and its worst case time complexity is $O(l)$, where l is the length of the string.

Tries are a better choice for the purpose of a predictive text system.

3.2.2 Hash Table vs Tries

Hash tables are an associative data structure where a key can be used to look up a value and the value is used to generate the key.

They require the use of a hash function. A good hash functions time complexity is not constant in regards to the length of the input. This is important when comparing hash tables are tries.

The space complexity of a hash map is $O(n)$ which is the same big-O complexity as a trie. But the total space complexity of a trie is larger than that of a hash table as it has to store a set of pointer of each node.

The time complexity of look up in a hash table is $O(1)$ if there are no collisions but could be $O(n)$ in the worst case. In the best case Tries and Hash tables have the same time complexity to look up a value. But in the worst case Tries still have $O(1)$ performance which is better than the hash table.

There are also other notable conditions in which a hash table preforms worse than a trie. Consider attempting to look up a word (e.g. ”zoologist”) starting with a letter (”z”) in a hash table or trie that contain no words starting with that letter (”z”). The hash table would evaluate the hash of that word which takes time proportional to the length of the word. Where as the trie would make a single check and know the word isn't is contained within.

A standard hash table can not preform prefix based look ups. This is because similar words produce dissimilar hashes.

3.2.3 Array Lists vs Tries

Array Lists are dynamically realizable arrays. They behave exactly like a standard array, but if an addition/insert is done when the list is full, it automatically increases its size.

The space complexity of an array list is dependant upon the number of elements in the array and the number of allocated but free elements. This means that the space complexity of an array list is $O(n)$. Array lists and tries have the same big-O space complexity. But an array list will have a smaller space foot print in practice as its space usage $n * l + k$ where as a trie uses $n * l * |A|$, where l is the average word length, k is the unused elements and $|A|$ is the alphabet size.

The time complexity of a search in an array list depends on the searching algorithm. For efficient searches the array must be sorted. So search complexity cannot be considered without considering the complexity or sorting the data.

In the use case of a predictive text system, words will be inserted infrequently but searches are done often. This means that in this use case the time complexity of sorts matter much less than the time complexity of searching. The best time complexity of a sorting algorithms is $O(n * log(n))$ and the time complexity of a binary search is $O(log(n))$. Since the words may only have to be sorted once for hundreds or thousands of searches it is better to consider the amortized cost. In this use case searching will take $O(log(n))$ time. Whilst this is a good time complexity it doesn't match the tries $O(1)$ time complexity.

The time complexity of inserting a value into an array tree is $O(1)$ to insert at the end and $O(n)$ to insert in the middle or beginning.That is if the number of items in the array list is smaller or equal to its size. But if the array has to resize this may not be the case. The performance can be $O(n)$ if the reallocation of the array fails (due to memory fragmentation) and a new array has to created in different area in memory. Because all of the items in the array have to be copied to the new array. The tries time complexity for insertion is constant time and so is better than the array lists linear time.

Considering the amortized time complexity, deletions in a array list are the same as insertions $O(n)$. This worst than tries.

Array lists have some advantages over hash maps and binary search trees when it comes to prefix look up. Because the words in an array list would be sorted any words that share a prefix will be guaranteed to be neighbours. Which is similar to the trie, but not quite as structured.

4 Design

4.1 Structure

The first thing to when designing a system is to decide upon the structure. In the case of a software project, like this, UML class diagrams are a good choice. The full diagram can be found in appendix B. The Class diagram helped show that the predictive text system can be spilt into 4 distinct layers. These are the application/UI layer, the predictive text engine layer, the dictionary storage layer and a library layer.

The application/UI layer will the the entry point of the program and will be responsible of creating and managing the predictive text engine. It will also manage the UI and inputs from the user. Partial words from the user input will be passed to the predictive text engine, which will respond with suggestions for complete words.

The predictive text engine layer will be responsible for creating an managing the dictionary storage layer. It will read the dictionary from the word list text file "word.txt" and add the words to the storage layer. The application layer will use this layer to query the storage layer. The predictive text engine will query the data storage layer to see if words are contained in storage and weather there are any words prefixed by a partial word in storage.

The dictionary storage layer will be responsible for storing and accessing the large set of words (>25k). It will use a Trie data structure to do this. It will provide fast look ups and prefixed based searching. It will use Library classes such as queues and stacks to simplify its own logic.

The library layer contains stack and list classes that can be used by the other layers. (standard c libraries such as stdlib, stdbool, string are not shown in the class diagram).

4.2 How suggestions are made

When a user has typed in section of text the application/UI layer separates the last word from the rest of the string. It does this by scanning the string backwards for the for first occurrence of a word separator (the first occurrence of a separator when scanning backwards is the last separator in the string). It then knows that the last word is between that separator and the end of the string. It then passes the last word to the text engine layer along with the number of suggestions it wants.

Once the partial word gets to the predictive text engine layer, the first thing the engine does is check to see if the partial word is contained in the data storage layer.

The data storage layer will then respond indicating weather the partial word is contained in the trie. This will tell the engine weather the partial word is either in the trie and starred (is a word), in the trie but not starred (a prefix) or not in the trie.

If the partial word is contained in storage and is a full word then we will add the partial word to the list of found suggestions. If the application/UI layer only requested one suggestion then we return the list now, if not we continue onwards.

If the partial word was in storage we can look for words that are prefixed by the partial word (e.g. if input was “Dat” the we can look for words like “Data” and “Date”). The predictive text engine passes the partial word to the dictionary storage layer, to preform a breadth first search of words prefixed by the partial word.

Before starting a breadth first search, the data structure layer walks the trie to find the node that represents the partial word. All words that are prefixed by the partial word will be children of this node. This node will be the starting point for the breadth first search.

[NOT CURRENTLY USING BREADTH FIRST SEARCH] Once the starting node has been found a breadth first search is preformed. The current node is added to a queue and we repeat the process below until the queue is empty.

The first thing done is de-queueing an item from the queue this is now the current node. The current node is check to see weather it is starred (is a compleate word). If the current node is then we add it to the list of found suggestions. If enough suggestions have been then found then the loop is broken. Then all of the children of the current node are added to the queue in order.

For the loop to have finished either enough suggestions were found or there we no more nodes to check. This means we now have a list of the shortest possible suffixes to the partial word. These are returned to the predictive text engine.

The predictive text engine now has a list of suggestions that may contain the partial word its self and words prefixed by the partial word. The predictive text engine then returns this list tp the application/UI layer where they are printed to the screen, next to the key needed to chose the particular suggestion.

The application/UI layer then lets the user either pick a suggestion or continue typing.

5 Testing Methodology

The testing of the predictive test system will be split up into three distinct sections. The first of these sections will involve testing the time complexity of the system and seeing if it matches expectations. The second section is the unit testing of the system. The final section of testing will be testing the user interacts of the system and comparing the behaviour against the behaviour defined in the success criteria.

5.1 Complexity

In order to test the time complexity of our system relative to the number of words in our word list we will have to create multiple word lists each of different size. The file "words.txt" that was provided is a list of 25,143 words. I have also obtained lists of the 10, 100, 1000 and 10000 most common words (from Google's Trillion word data set [5]). These are contained in the files named "10words.txt" , "100words.txt" , "1000words.txt" and "10000words.txt".

This means we can run a function on each of these data sets then use the resulting times to estimate the real word time complexity of the function.

5.2 Unit Testing

Every module (.c file) used within the system (excluding main.c, the unit tests themselves and the unit test runner) has a corresponding unit test file (ending with "_UT.c"). These unit tests evaluate the behaviour of all of the functions a module makes public via its interface (.h file). The module UnitTester is responsible for running each set of unit tests. Each unit test is written to evaluate the behaviour of a specific part of the module under test.

To run these unit tests the Symbol UTEST has to be defined at compile time. This can be done by switching from the debug/release configuration to the unittest configuration. If UTEST is defined the UnitTester module is ran, and preforms all unit tests and prints out weather the tests are failing or passing.

5.3 User interactions

Unlike the other two sections these tests are not strictly quantitative. Each of the success criteria that relate to user input or interaction will be evaluated.

6 Expected Results

6.1 Complexity

The complexity of the trie data structure is expected to be the same as discussed in Section 3.1 about Tries.

- Space complexity is $|A|\Sigma_i len(w_i)$. In terms of n this is $|A| * n * l$. And in big-O notaion this is $O(n)$.
- Time complexity of lookups is $len(w)$. In big-O notaion this is $O(1)$.
- Time complexity for inserts is $|A| * len(string)$. Which is $O(1)$ in big-O notation.

6.2 Unit Testing

Because the system was unit tested though out its development, all of the bugs found from unit tested were dealt with as they were found. This means that all of the unit tests are expected to pass.

6.3 User interactions

The completed system is expected to meet all of the "must do" criteria and some of the "should do" and "could do" criteria. Below are the criteria where the completed system is expected to fall short.

Criteria 13 The system should be able to make suggestions of words that share a common prefix with a partial word.

This functionality is currently not implemented in the system.

Criteria 17 The system could be extended to deal with punctuation.

The completed system does not deal with punctuation in any way. It strips it from the loaded words and may replace any punctuation that comes from user input.

Criteria 18 The system could be extended to deal with Capitalisation.

The completed system does not deal with capitalization. Words loaded from file are processed in such away that capitals and lower case letters are mapped to the same space and so all capitalization data is lost.

7 Test Results

7.1 Complexity results

7.2 Results

The results from running the unit tests on the final build can be found in appendix B or in a file named ".\Log\UnitTestLog.txt", to get these results from the program use the method described Section 5.2.

All unit tests passed.

7.3 User interaction results

8 Conclusion

References

[1] Booyabazooka (based on PNG image by Deco). Modifications by Superm401. *Example of a trie.* [Online; accessed 13-January-2017]. 2006. URL: `https://en.wikipedia.org/wiki/File:Trie_example.svg`.

[2] Peter Brass. *Advanced Data Structures.* Cambridge University Press, 2008. Chap. 8.1, pp. 336–356.

[3] Peter Brass. *Advanced Data Structures.* Cambridge University Press, 2008. Chap. 8.1, p. 341.

[4] Thomas A. Standish. *Data Structures and Software Principles in C.* Addison-wesley publishing company, 2008. Chap. 9.7, p. 374.

[5] Google Web Trillion Word Corpus as described by Thorsten Brants et al. *google-10000-english.* [Online; accessed 20-January-2017]. 2012. URL: `https://github.com/first20hours/google-10000-english`.

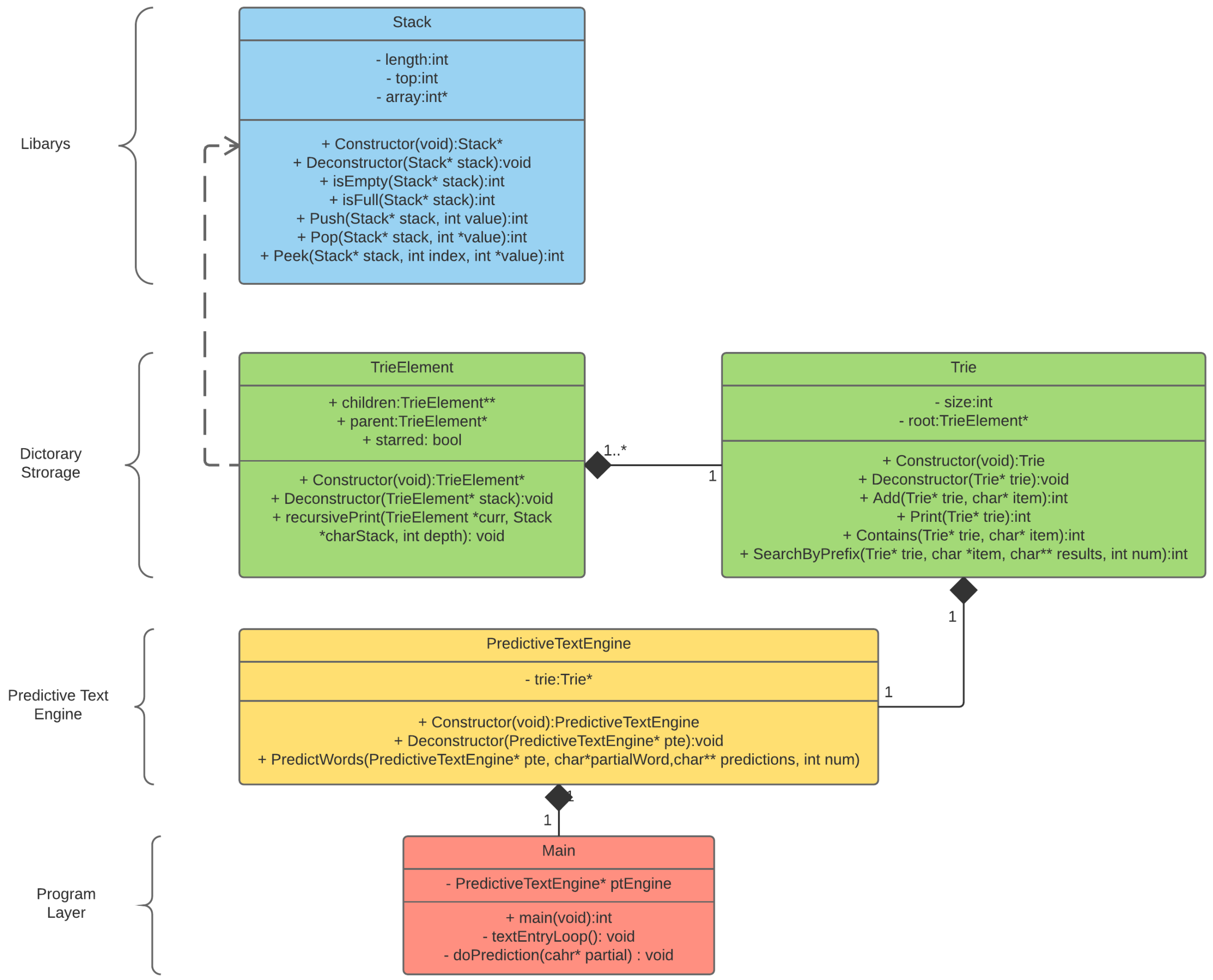
[6] Eric D. Rowell. *Big-O Cheat Sheet.* [Online; accessed 20-January-2017]. URL: `http://bigocheatsheet.com/`.

Appendices

A Class Diagram

PREDICTIVE TEXT UML

Zak West | January 21, 2017


Libarys
Dictorary
Strorage
Predictive Text
Engine
Program
Layer
Stack
- length:int
- top:int
- array:int*
+ Constructor(void):Stack*
+ Deconstructor(Stack* stack):void
+ isEmpty(Stack* stack):int
+ isFull(Stack* stack):int
+ Push(Stack* stack, int value):int
+ Pop(Stack* stack, int *value):int
+ Peek(Stack* stack, int index, int *value):int
TrieElement
+ children:TrieElement**
+ parent:TrieElement*
+ starred: bool
+ Constructor(void):TrieElement*
+ Deconstructor(TrieElement* stack):void
+ recursivePrint(TrieElement *curr, Stack
*charStack, int depth): void
1..*
1
Trie
- size:int
- root:TrieElement*
+ Constructor(void):Trie
+ Deconstructor(Trie* trie):void
+ Add(Trie* trie, char* item):int
+ Print(Trie* trie):int
+ Contains(Trie* trie, char* item):int
+ SearchByPrefix(Trie* trie, char *item, char** results, int num):int
1
1
PredictiveTextEngine
- trie:Trie*
+ Constructor(void):PredictiveTextEngine
+ Deconstructor(PredictiveTextEngine* pte):void
+ PredictWords(PredictiveTextEngine* pte, char*partialWord,char** predictions, int num)
1
1
Main
- PredictiveTextEngine* ptEngine
+ main(void):int
- textEntryLoop(): void
- doPrediction(cahr* partial) : void

B Unit Testing Output

```
Fri Jan 20 19:32:16 2017

====//====//====//====//====//====//====//====//====//====
====                                                  ====
====                 UNIT TESTING                     ====
====                                                  ====
====//====//====//====//====//====//====//====//====//====
==== ==== ==== ==== ==== ==== ==== ==== ==== ====
               UNIT TESTING Stacks
==== ==== ==== ==== ==== ==== ==== ==== ==== ====
 [PASS]          Constructed stack pointer was not null
 [PASS]          Two constructed stacks did not have the same pointer
 [PASS]          Pushing a value to a stack worked
 [PASS]          Pushing multiple values to a stack worked
 [PASS]          Popping a value off a stack worked
 [PASS]          Popping multiple values off a stack worked
 [PASS]          Adding values to one stack doesnt affect the other stack
 [PASS]          Checking if a stack isEmpty worked
 [PASS]          Checking if a stack isFull worked
 [PASS]          Peeking at a value in a stack worked
 [PASS]          Turning a stack into an array worked
 [PASS]          Checking a stacks Height worked
==== ==== ==== ==== ==== ==== ==== ==== ==== ====
 [PASS]         All Unit Tests for Stacks passed !


==== ==== ==== ==== ==== ==== ==== ==== ==== ====
               UNIT TESTING Trie
==== ==== ==== ==== ==== ==== ==== ==== ==== ====
 [PASS]          Constructed trie pointer was not null
 [PASS]          Adding values to a trie worked
 [PASS]          Contains worked
==== ==== ==== ==== ==== ==== ==== ==== ==== ====
 [PASS]         All Unit Tests for Trie passed !


====//====//====//====//====//====//====//====//====//====
 [PASS]          All Unit Tests passed !
```