Implementation of TRIE Data Structure

Team Members:

1. Saikiran A	**1PI10CS076**
2. Sankarshan Bhat	**1PI10CS078**
3. Vikyath Harekal	**1PI10CS0116**

What is Trie ?

A Trie or a Prefix Tree is an ordered tree data-structure used to store associative arrays where the keys are usually strings. All the descendants of a node have a common prefix of a key associated with that node. A Trie has a lot of advantages over Binary Search Trees and also in some cases, Hash-Tables.

It is commonly used in Dictionary representation as it poses an advantage over space-complexity by its structure.

Why is trie needed ?

The trie can insert and find strings in O(L) time .The set <string> and the hash tables can only find in a dictionary words that match exactly with the single word that we are finding; the trie allow us to find words that have a single character different, a prefix in common, a character missing, etc

Trie can also be used to store phone numbers and hence can be used for implanting the whole telephone directory. Similar to searching of strings, searching of phone numbers will also become faster and better.

The current STL does not have any container that supports efficient for the above to situations. So this implementation provides a container that makes searching/index much better than the current containers.

Interface:

bool insert(const T*) ;

This function, returns true on successful insertion of T, else returns false.

void display() ;

This function displays all the words that are present in the Trie.

bool deleteString(T* str) ;

If T is present in the Trie, this function deletes it and returns true, otherwise returns false.

int size() ;

This function return number of words that are currently in the Trie.

void inorderTraversal()

Prints in-order traversal of the Trie.

Implementation of Generic Trie

The plan to implement trie was to basically work with all generic functions/algorithms that usually work on all the stl containers like list, vector etc.

For this to work, the main thing we had to do was to support iterator for trie.

We also had to overload pre-increment operator, dereference operator and Comparison operators.

Basic Implentation:

The basic implantation of trie contains a node class, which contains a generic data field which can have any type (usually characters or integers) and also two pointers to itself to denote left and right of the sub tree from that node. Additionally it also contains a Boolean field which denotes whether the word is complete or not at that node.

We also have a Trie class, which contains a pointer to the node class, which is used a root node for the trie tree. In our implementation, we consider the root node to be null and then follows the words. We used the friend concept to make Trie a friend of Node so that all the members(even private) of the Node class can be accessed in the Trie class.

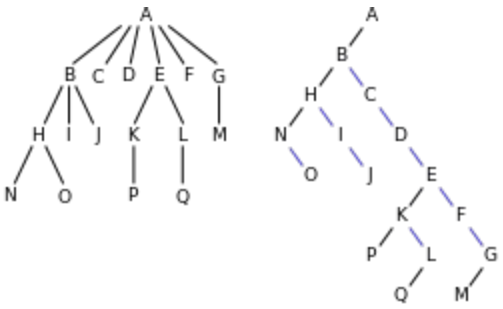
For the trie class, we supported, insert a word, delete a word, display all words, search a word in the dictionary, in-order traversal and few others.

What is LCRS ?:

LCRS is left-child right-sibling, which can be used to implement a general purpose tree(a tree in which a node can have any number of children).

LCRS is basic idea if having child to the left of node and its sibling to the right of node.

As an example a general purpose tree (left) when converted to an LCRS tree would look like the one in the right.



Iterator Support:

In the context of Trie data structure for an eglish dictionary as an example , iterator for single character is meaningless, as trie is a container for storing words. We had to implement iterator for traversing word by word. And also bidirectional iterator has very low significance here, so we implemented a forward iterator which can be used to traverse the entire trie tree. The idea was to implement a list inside the iterator

class as a container for characters which form a word. This list is filled up with the characters of the next word, when the iterator is incremented. The iterator class here was implemented as an inner class of Trie. We also use a stack here to push the node which is part of another word, so that we may need not reinsert it into the list and also not loose the track of words in the trie. Once all the words from that node are traversed, that node is removed from the stack.

Iterator begin is used to get the first word of the trie into the list, whereas iterator end is null, which is used to denote end of the trie.