# SOFTWARE PROGRAMMING FOR PERFORMANCE

**QWERTYUIOP**
Anishka Sachdeva (2018101112)
Satyam Viksit Pansari (2018101088)

**Project Problem Statement** :  Build a In-Memory Key-Value Storage Software in C++

## Data structure to be used  :  TRIES

### 1. About Tries:

Trie is a **tree based data structure** which can be used for efficient retrieval of a key in a huge set of data/strings. Using Trie, search complexities can be brought to optimal limit (key length). Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as end of word node.

### 2. Implementation of the TRIE data structure:

The Trie will store the **keys** of the data structure. Since each key can be of 64 bytes, the maximum length of the trie will be equal to the maximum length a key we can have (<=64 bytes).

There exists a head node which can have 52 child pointers in total ( 26 for uppercase letters and 26 for lowercase letters).

Each node of the trie data structure will be a struct node consisting of the following values:

- Pointers to Child Nodes (Array of pointers)
- Number of children
- Boolean value to store the end of the string
- Number of true boolean values (Number of keys below a node including the node)
- A pointer to store the **value** of the key

### 3. Implementation of the APIs:

let length of the tree = h

1. **get (key)** - Returns value for the key
   ***Implementation:***

   *Worst Case Complexity = O(h)*

   Search a key in a Trie using recursion.
   In searching a key, only compare the characters of the key and move down. The search can terminate due to the end of a string or lack of key in the trie. In the former case, if the boolean value to store the end of the string of the last node is true, then the key exists in the trie. In the second case, the search terminates without examining all the characters of the key, since the key is not present in the trie.

2. **put (key, value)** - Add key-value, overwrite existing value
   ***Implementation:***

   *Worst Case Complexity : O(h)*

   An efficient approach is to treat every character of the input key as an individual trie node and insert it into the trie. The children are an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array of children. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of the key, and mark end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word.

3. **delete (key)** - Deletes a key along with its value
   ***Implementation:***

   *Worst Case Complexity = O(h)*

   To delete a key, we need to first search a key (compare the characters of the key and move down). The search can terminate due to the end of a string or lack of key in the trie. In the former case, if the boolean value to store the end of the string of the last node is true, then the key exists in the trie. In the second case, the search terminates without examining all the characters of the key, since the key is not present in the trie. If the key exists, we put the value poninter = Null Pointer(i.e. delete the value of the key basically) and boolean value = False. Now if this node has no children pointers, we simply delete the node (as the node is of no use now) and then check the same for the parent recursively. If this node has some child pointers, we do not delete the node.

4. **get(int N) -** Returns Nth key-value pair
   ***Implementation:***

   *Worst Case Complexity = O(h * log52)* where log 52 is for doing the **binary search**

   Trie will store the keys in alphabetical order as children(which are basically characters/alphabets) will be arranged in alphabetical order from left to right (increasing index) .In this arrangement Nth key in the alphabetical order can be retrieved by simply doing a BFS using the number of true boolean values . In essence we will iterate through children nodes and keep adding the number of true boolean values and find the required node in which the exact n(sum of number of true boolean values) lies.

5. **delete(int N)** - Delete Nth key-value pair
   ***Implementation:***

   *Worst Case Complexity = O(h * log52)* where log 52 is for doing the **binary search**

   Trie will store the keys in alphabetical order as children(which are basically characters/alphabets) will be arranged in alphabetical order from left to right (increasing index) .In this arrangement Nth key in the alphabetical order can be retrieved by simply doing a BFS using the number of true boolean values. In essence we will iterate through children nodes and keep adding the number of true boolean values and find the required node in which the exact n(sum of number of true boolean values) lies. After finding the key, we put the value pointer = Null Pointer(i.e. delete the value of the key basically) and boolean value = False. Now if this node has no children pointers, we simply delete the node (as the node is of no use now) and then check the same for the parent recursively. If this node has some child pointers, we do not delete the node.