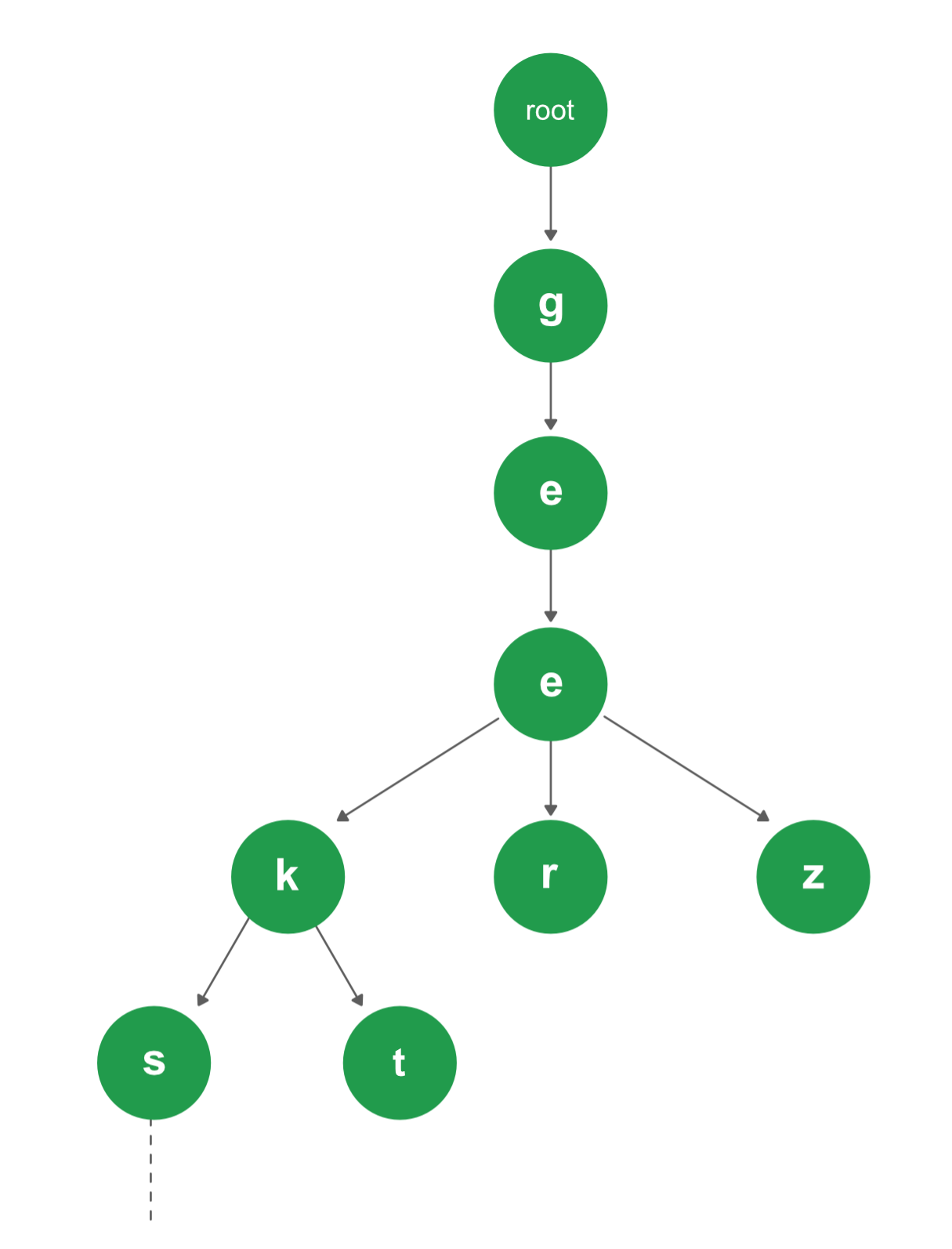
**Topic 1: Insertion and Search in Trie Data Structure**

The Trie data structure is an efficient information **re-*trie*-val** data structure. The Trie data structure is used to efficiently search for a particular string key among a list of such keys. Using the trie, the lookup operation can be performed in time complexity of **O(key\_length)**.  
  
**Representation**:  
A trie is represented as a tree where each node contains 26 pointers which is equal to the number of characters in the English alphabet. In Trie basically, the common prefix of all strings are represented as a common path in the tree.  
  
For Example, look at the below sample Trie:  
  
There are 4 different strings in the Trie above: [ "geeks", "geekt", "geer", "geez" ]. Since the prefix **"gee"** is common among all of the strings, so it is a common path in the Trie as well. Again for the strings "geeks" and "geekt", the common prefix is "geek", so after dividing paths from "gee", the node with character "k" is common to these two strings.  
  
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 the end of the word node. A Trie node field *isEndOfWord*is used to distinguish the node as the end of word node. A simple structure to represent nodes of the English alphabet can be as following,

// Trie node  
struct TrieNode  
{  
 struct TrieNode \*children[ALPHABET\_SIZE];  
  
 // isEndOfWord is true if the node  
 // represents end of a word  
 bool isEndOfWord;  
};

**Insertion in a Trie**

Inserting a key to **Trie**is a simple approach. Every character of the input key is inserted as an individual Trie node. Note that 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. The key length determines Trie depth.

**Searching a Key in Trie**

Searching for a key is similar to the insert operation, however, we only compare the characters 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 *isEndofWord* field 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.  
  
The following picture explains construction of trie using keys given in the example below,



In the picture, every character is of type *trie\_node\_t*. For example, the *root* is of type trie\_node\_t, and it's children *a*, *b* and *t* are filled, all other nodes of root will be NULL. Similarly, "a" at the next level is having only one child ("n"), all other children are NULL. The leaf nodes are in blue.  
  
  
**Implementation**:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// C++ implementation of search and insert

// operations on Trie

#include <bits/stdc++.h>

using namespace std;

const int ALPHABET\_SIZE = 26;

// trie node

struct TrieNode

{

struct TrieNode \*children[ALPHABET\_SIZE];

// isEndOfWord is true if the node represents

// end of a word

bool isEndOfWord;

};

// Returns new trie node (initialized to NULLs)

struct TrieNode \*getNode(void)

{

struct TrieNode \*pNode = new TrieNode;

pNode->isEndOfWord = false;

for (int i = 0; i < ALPHABET\_SIZE; i++)

pNode->children[i] = NULL;

return pNode;

}

Run

Java



**Output :**

the --- Present in trie  
these --- Not present in trie  
their --- Present in trie  
thaw --- Not present in trie

**Time Complexity:** Both the insert and search operations in a Trie takes **O(M)** time. Where **M**is the length of the key being inserted or searched for in the Trie.

**Topic 2 : Deletion in Trie**

We had already discussed the representation of a Trie data structure, inserting a key and searching a key in a Trie.  
  
Let us look at the process of deleting a given key from a Trie.  
  
During delete operation, we delete the key in a bottom-up manner using recursion. The following are possible cases which may occur while deleting the key from trie,

* **Case 1**: Key may not be present in the trie. In this case, the delete operation should not modify trie.
* **Case 2**: Key present as a unique key (no part of key contains another key (prefix), nor the key itself is a prefix of another key in trie). In this case, delete all the nodes of that key.
* **Case 3**: Key is a prefix key of another long key in the trie. In this case, simply unmark the leaf node.
* **Case 4**: Key present in the trie, having at least one other key as a prefix key. In this case, delete nodes from the end of the key until the first leaf node of longest prefix key.

**Implementation**:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// C++ implementation of delete

// operations on Trie

#include <bits/stdc++.h>

using namespace std;

const int ALPHABET\_SIZE = 26;

// trie node

struct TrieNode {

struct TrieNode\* children[ALPHABET\_SIZE];

// isEndOfWord is true if the node represents

// end of a word

bool isEndOfWord;

};

// Returns new trie node (initialized to NULLs)

struct TrieNode\* getNode(void)

{

struct TrieNode\* pNode = new TrieNode;

pNode->isEndOfWord = false;

for (int i = 0; i < ALPHABET\_SIZE; i++)

pNode->children[i] = NULL;

return pNode;

}

Run

Java



**Output:**

Yes  
No  
Yes

**Topic 3 : Implementing Auto-Complete Feature using Trie**

The **Auto-Complete** feature is widely useful in showing suggestions when the user types a certain word. In a simpler version, the auto-complete feature lists all of the strings which have a matching prefix queried by a user.  
  
For example, if the dictionary stores the following words **{“abc”, “abcd”, “aa”, “abbbaba”}** and the User types in **“ab”** then he must be shown *{“abc”, “abcd”, “abbbaba”}* as a result as all of them have the prefix *ab*.

**Implementation using Trie**

We can implement this Auto-Complete feature easily using a Trie data structure. We will have to first insert all of the strings initially in a Trie, and then fetch all matching strings based on the user's query.  
  
**Detailed Algorithm**:

1. Insert all of the given strings in a Trie.
2. Search for the given query using the standard Trie search algorithm.
3. If the query prefix itself is not present, return -1 to indicate the same.
4. If the query is present and is the end of the word in Trie, print query. This can quickly be checked by seeing if the last matching node has the *isEndWord*flag set. We use this flag in Trie to mark the end of word nodes for the purpose of searching.
5. If the last matching node of the query has no children, return.
6. Else recursively print all nodes under a subtree of the last matching node.

**Implementation**:

C++



1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

// C++ program to demonstrate auto-complete feature

// using Trie data structure.

#include<bits/stdc++.h>

using namespace std;

// Alphabet size (# of symbols)

#define ALPHABET\_SIZE (26)

// Converts key current character into index

// use only 'a' through 'z' and lower case

#define CHAR\_TO\_INDEX(c) ((int)c - (int)'a')

// trie node

struct TrieNode

{

struct TrieNode \*children[ALPHABET\_SIZE];

// isWordEnd is true if the node represents

// end of a word

bool isWordEnd;

};

// Returns new trie node (initialized to NULLs)

struct TrieNode \*getNode(void)

{

struct TrieNode \*pNode = new TrieNode;

pNode->isWordEnd = false;

for (int i = 0; i < ALPHABET\_SIZE; i++)

pNode->children[i] = NULL;

Run

Java



**Output**:

hel  
hell  
hello  
help  
helping  
helps