# Query Log Processor

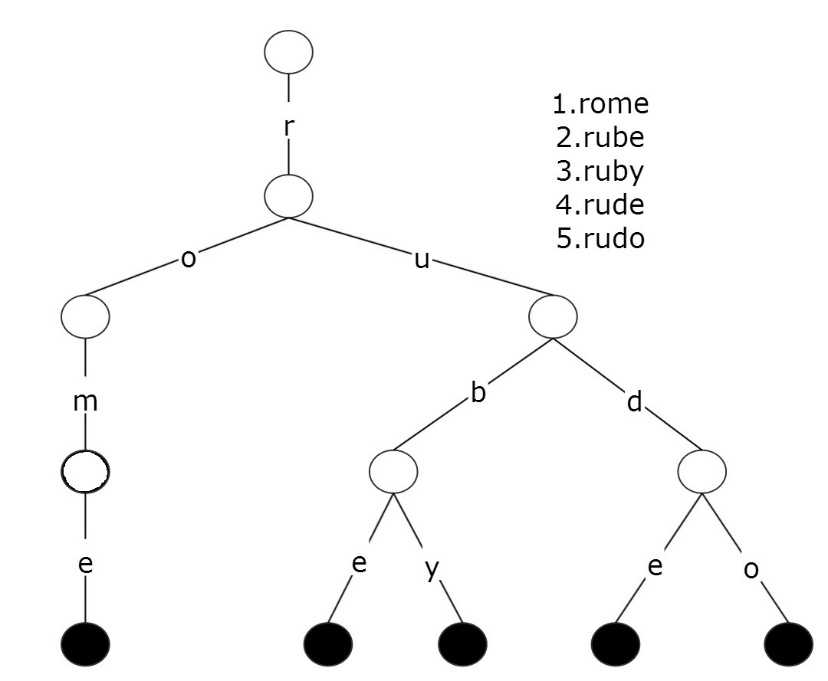
What is the problem?

* Huge amount of query needs to be stored and processed each day
* Weight of a query changes dynamically from time to time thus the ability to detect such changes
* Automated Suggestion and information retrieval needs to be done almost instantly
* Relevant and accurate information needs to be retrieved

Compressed Trie : A Possible Solution

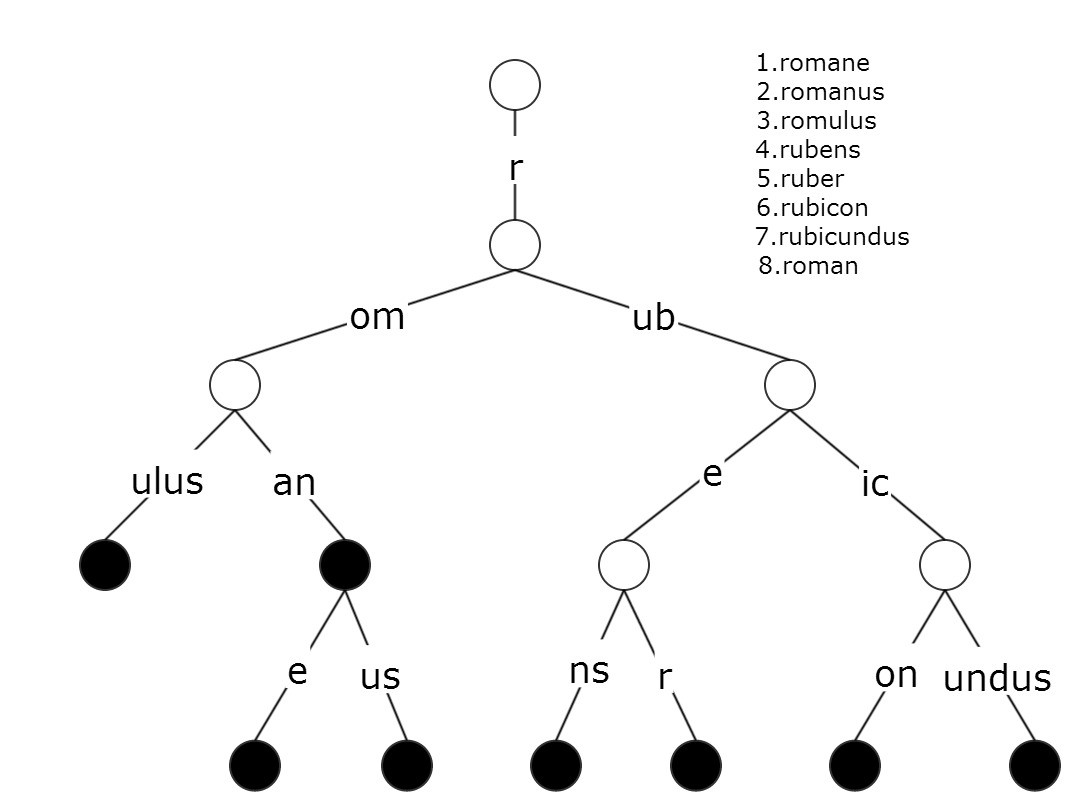
Inorder to understand compressed trie we must introduce trie data structure which is a very ancient data structure

The word trie is an infix of the word "re**trie**val" because the trie can find a single word in a dictionary with only a prefix of the word. The main idea of the trie data structure consists of the following parts:

The trie is a tree where each vertex represents a single word or a prefix.

The root represents an empty string (""), the vertexes that are direct sons of the root represent prefixes of length 1, the vertexes that are 2 edges of distance from the root represent prefixes of length 2, the vertexes that are 3 edges of distance from the root represent prefixes of length 3 and so on. In other words, a vertex that are k edges of distance of the root have an associated prefix of length k.

Let **v** and **w** be two vertexes of the trie, and assume that **v** is a direct father of **w**, then **v** must have an associated prefix of**w**.The next figure shows a trie with the words "rome", "rube", "ruby", "rude", "rudo".

In Trie data structure each prefix of a word is stored only once but for each new string it stores nodes exactly the size of the string.Thus Trie has some certain disadvantage as we can see from the figure.It stores too many unnessary nodes which in our case is very fatal.As we have said query sentences are huge, if we store all the query sentences in a normal trie our memory would run out.

Thus the solution of this problem is Compressed Trie. We will try to simulate compressed trie with the following figure.

Compressed Trie:

A compressed trie is a trie with one additional rule:

1. Each internal node has   ≥ 2   children
2. Obtained from standard trie by compressing chains of redundant nodes

So inorder to implement Compressed Trie we must stop using redundant nodes which from a redundant chain.Related information on compressed trie can be found on this link: <http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Text/trie02.html>.

Advantages of using Compressed Trie:

1. We can seek out the required information extremely fast, at constant time i.e. O(1) complexity.
2. Prefix matching can be implemented in compressed trie which is in our case absoloutely necessary.
3. The total storage required by compressed trie is extremely efficient when storing large quantity of query sentences as compressed trie compresses number of nodes used.
4. Insert and Delete operation is extremely fast in compressed trie.
5. For Weighted Information Retrieval Compressed Trie can be modified.
6. Compressed Trie allows us to backup the whole tree which is important if we are to intrigate this service in any online search engine as their server is prone to crashing.
7. Number of Suggestions retrieved in a compressed trie can be easily controlled.

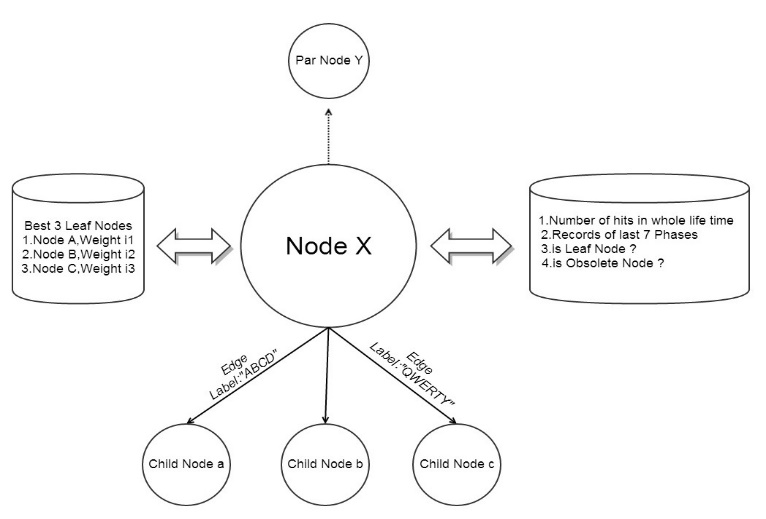
Challenges of implementing Compressed Trie:

1. How we seek out the most relevant information from Compressed Trie Tree.
2. How to calculate the weight of a query string and implement it on such tree.
3. Weight of a leaf node in a compressed trie tree changes dynamically , How to implement this?
4. What information needs to be stored which gives us all the information we require and all this at minimal storage requirement.

Proposed Solution and Short Description of Implementation:

Inorder to solve all the required challenges and we have devised an interesting solution.To understand our proposed solution you must know what type of information we are storing in each node in a compressed Trie tree.First of Each node in our Compressed Trie Tree should have some specific required information which are:

* Link to parent Node
* Link to all childs
* Global number of hits of this node in its Life time.
* Records(number of hits) of last 7 Phases of this current node.
* Best Weighted 3 or 4 nodes who are child of this current node (immeadiate or not)
* Current node leaf node or not
* Current node obsolete node or not

Why such information is stored will be discussed in brief shortly. These specific information is enough for our purpose of retrieving information from the Compressed Trie Tree.

**Weight Calculation of a Node :**

Before simulating the information retrieval process we must clearly define how we are defining weights of each node and show some examples.

Case 1.

Weight of a specific query can go up and down due to many reasons. One of them is due to change of time.An Useful Example is given below.

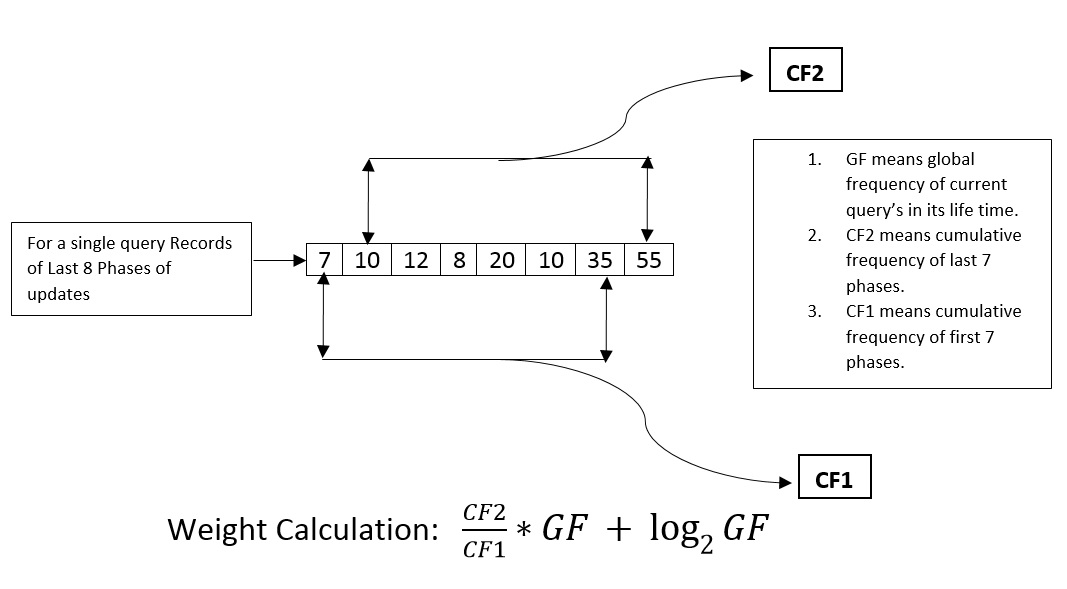
|  |  |  |
| --- | --- | --- |
| Popular Queries in Summer | Popular Queries in Winter | Popular Queries in All Seasons |
| সুতির পাঞ্জাবী | সোয়েটার এর দাম | আজকের খবর |
| আইস ক্রিম | হিটার পাবো কোথায়? | বাংলাদেশের দর্শনীয় জায়গা গুলো কি কি? |

Case 2.

There are some queries which can gain instantly become very much popular and within a short interval of time can lose its popularity.

|  |
| --- |
| Queries that gain and lose popularity in short interval |
| আইফোন ৫ এর দাম বাংলাদেশে কত? |
| মেসি ফর্ম এ নেই কেন? |
| বাংলাদেশ ক্রিকেট দলের খেলার ফলাফল? |

By analysing above requirements we have mathematically defined weight of every single query that is every single leaf node.Suppose we have records of last 8 phases of every single query.We also know the global frequency GF of every single leaf node.

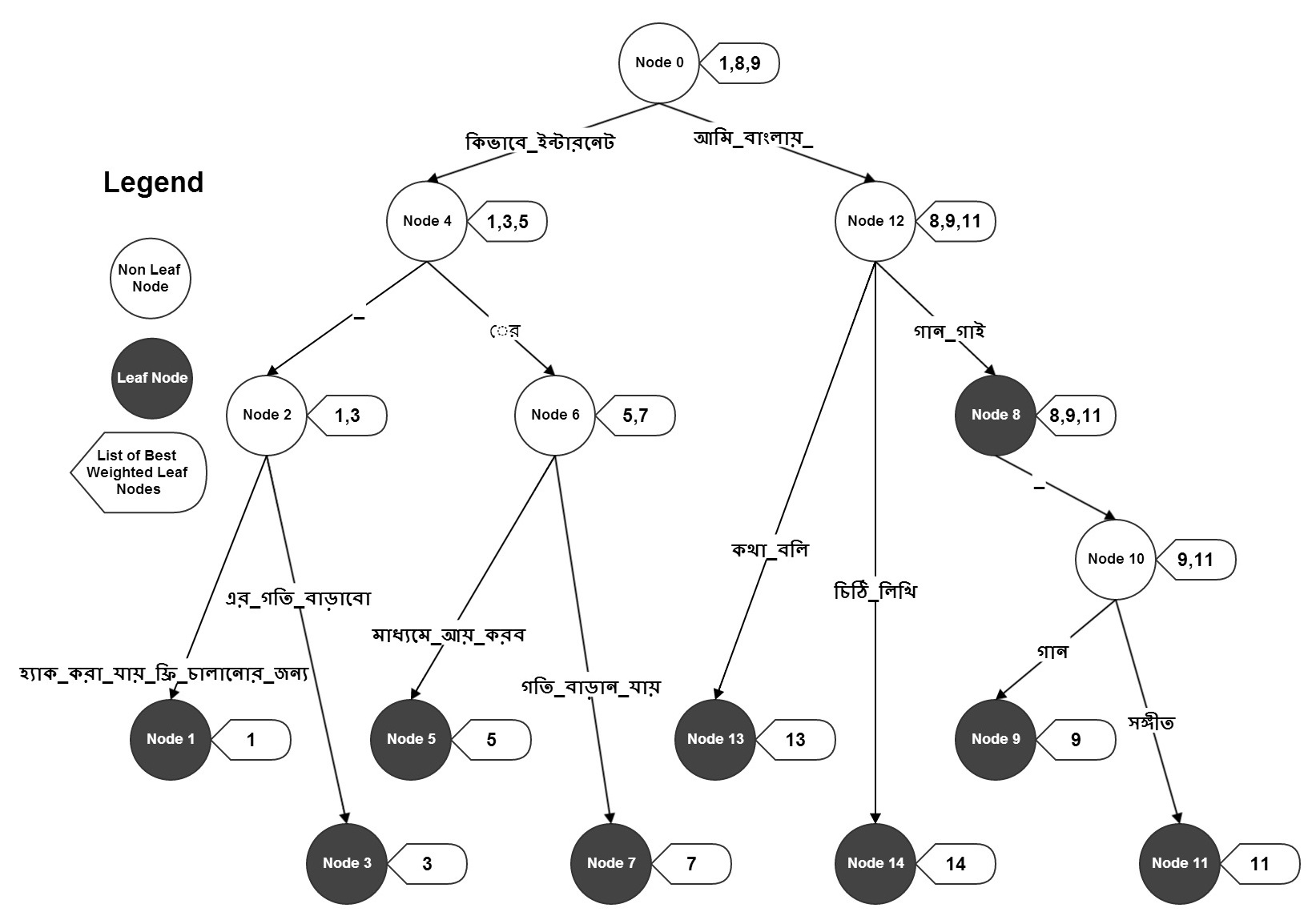


Building The Tree:

For Each query string we update the whole tree and in the leaf node we update

1. Its global frequency
2. Calculate its weight using the Weight Calculation Formula.
3. Update the phase records i.e last record is removed while new record is inserted.

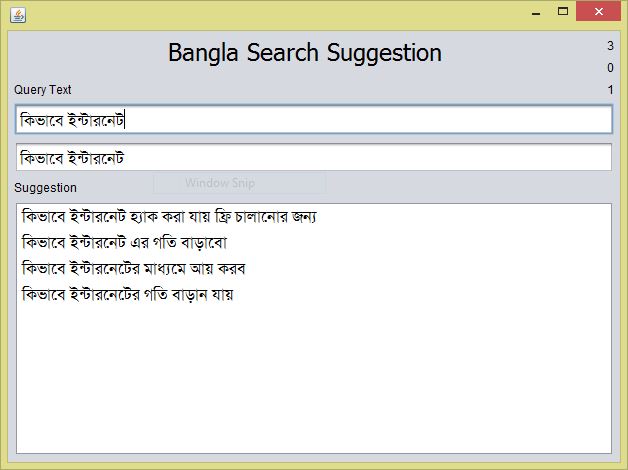
For Every single node other than the leaf node we do all the above things except point 2 and 3.At the same time for a non leaf node we update the list of best leaf nodes which is its immeadiate or non immeadiate child.The list of best leaf nodes is always sorted according to each leaf node’s Weight which is generated at creation of the leaf node.



In the figure we have 8 query sentences.They are given as well as their weight is also given.When we

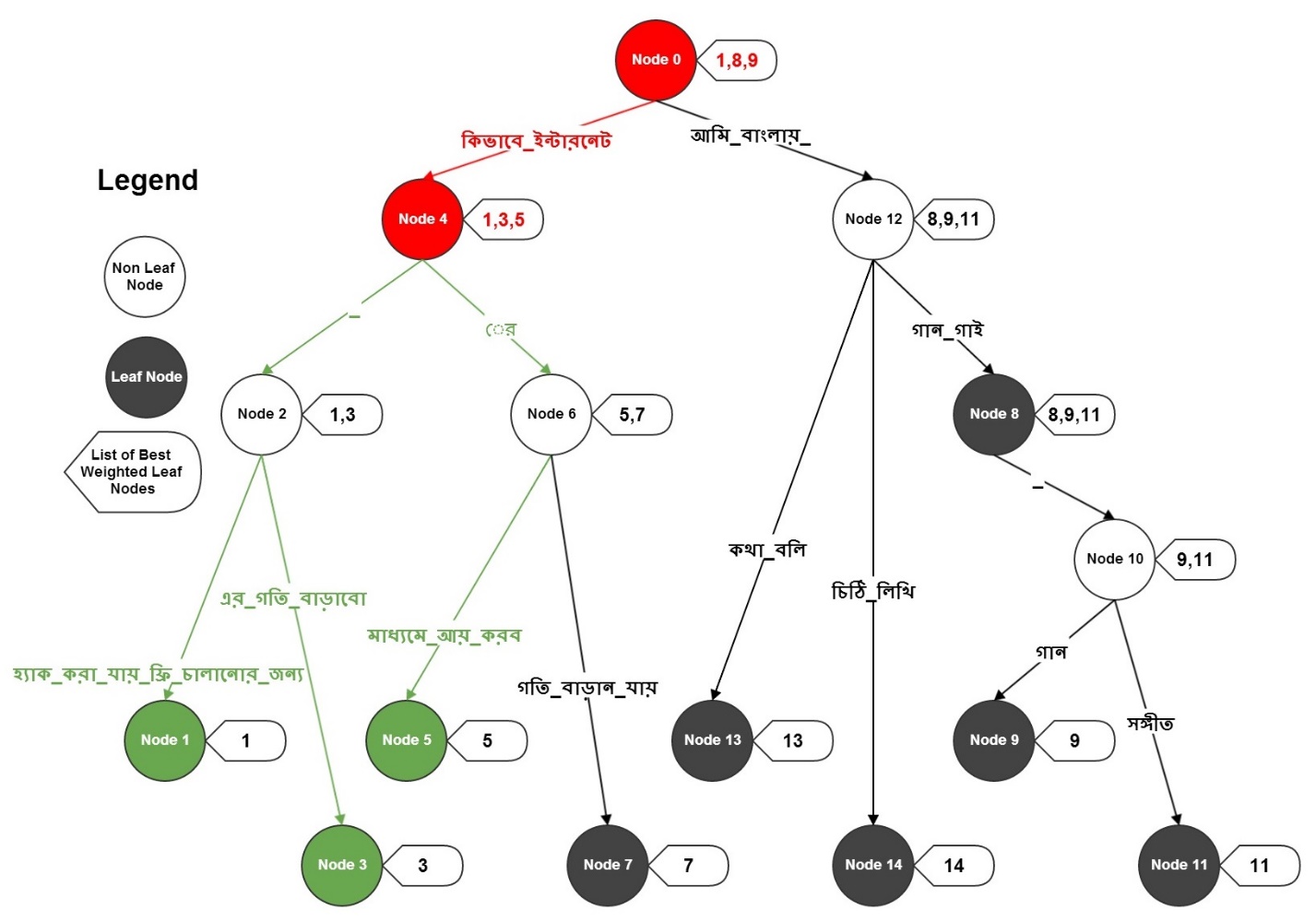
|  |  |
| --- | --- |
| Query Sentence | Weight |
| কিভাবে\_ইন্টারনেট\_হ্যাক\_করা\_যায়\_ফ্রী\_চালানোর\_জন্য | ৫৫ |
| কিভাবে\_ইন্টারনেট\_এর\_গতি\_বাড়াবো | ৮ |
| কিভাবে\_ইন্টারনেটের\_মাধ্যমে\_আয়\_করব | ৬ |
| কিভাবে\_ইন্টারনেটের\_গতি\_বাড়ান\_যায় | ৪ |
| আমি\_বাংলায়\_গান\_গাই | ৫০ |
| আমি\_বাংলায়\_গান\_গাই\_গান | ৩০ |
| আমি\_বাংলায়\_গান\_গাই\_সঙ্গীত | ১৫ |
| আমি\_বাংলায়\_কথা\_বলি | ১০ |
| আমি\_বাংলায়\_চিঠি\_লিখি | ৬ |

completed updating all this sentences in Compressed Trie the tree looks like above.Now suppose User Searched for “কিভাবে ইন্টারনেট” then we will get below output.



How this is information is fetched is described below.

* In our compressed trie we try to look up the string “কিভাবে ইন্টারনেট” as close as possible.
* When ever we find an unmatched character we stop right there.
* We know the path from the root to the last accesible node.
* All the information is fetched from the nodes starting from the root and ending at the last node .
* The last node knows the best possible leaf nodes that we could’ve gone to.So this is given the highest priority among all the other nodes retrieved.
* Thus We get a list of leaf nodes along their weight.
* These leaf nodes are sorted according to their weight calculated.
* Among the candidate leaf nodes we pick the best ones according to their weight.
* These leaf nodes are then traversed right back to the root to get the original string.



The Above figure simulates what we have stated.The simulation procedure is stated below

1. Start from root node( Node 0 )
2. Last accesible node is Node 4.
3. Pick the leaf nodes from Node 0 and 4.
4. Node 4 will have higher priority from Node 0 as it is closer to the bottom of the tree.
5. Collect the leaf nodes .They are Node 1,3,5,8,9.
6. But Leaf node 8,9 came from Node 0 thus having less priority.
7. So we select Leaf nodes 1,3,5 sorted according to their weight.
8. Now For each leaf node 1,3,5 traverse backwards right to the top of the tree that is Node 0.
9. We get the result String.

Complexity Analysis and Memory Usage:

**Performance :**

For Each query string checked in Compressed Trie we just need to scan the input string , the results are picked in constant complexity.Thus information is retrieved in O(1) complexity which is the best possible run time for any algorithm.Though there are some constant factors added.For Even further efficiency reader’s can minimize the constant factor to get even faster results.

**Memory Usage:**

For each unique query string inserted in Compressed Trie we may have atmost 2 nodes (worst case).Thus if there are n unique strings worst case memory usage is 2\*n which can stated O(n) in Big-O notation.

Future Work:

There are some possible enhancements that can be done on the above algorithm to get even better accuracy and better run time. Some features can also be included to improve user’s experience.They are

* Query can be personalized for user to user to get improved accuracy
* The Compressed Trie Tree can be stored in distributed servers, thus reducing overhead and storing terabytes of data.
* Zero query search suggestion can be offered to users get better user’s experience.