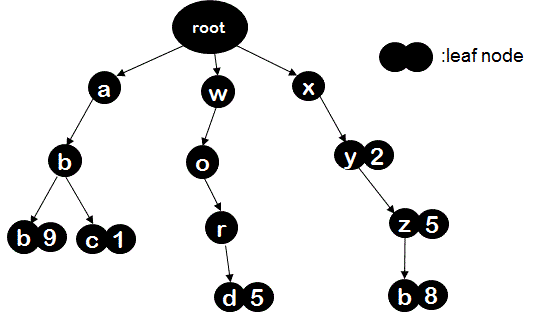
Trie Data Structure | Insert and search

**A trie is a data structure used for efficient retrieval of data associated with keys. If key is of length n, then using trie worst case time complexity for searching the record associated with this key is O(n). Insertion of (key, record) pair also takes O(n) time in worst case.  
  
Trie's retrieval/insertion time in worst case is better than hashTable and binary search tree both of which take worst case time of O(n) for retrieval/insertion.  The trie structure though in theory has same worst case space complexity as hashTable or a binary tree, memory needed to store pointers causes it to be less space efficient during implementations.  
  
You can see in the below picture as to how a trie data structure looks like for key, value pairs ("abc",1),("xy",2),("xyz",5),("abb",9)("xyzb",8), ("word",5).  
  
  
  
Each node including root has 'n' number of children, where 'n' is total number of possible alphabets out of which a key would be formed. In this example, if we assume that a key would not consist of any other alphabets than characters 'a' to 'z' then value of 'n' would be 26. Each node therefore would point to 26 other nodes. Each node is basically an alphabet in the path from root node to leaf node(which stores value for that key). For example, to search the record associated with key 'abc',  we start from root node then go to 0th child since first character in key 'abc' has index of 0 in alphabets. By taking 0th index, we reach node 'a', at this point we go to 1st child and reach node 'b', here we take 2nd path to child and go to node 'c'. Now to reach node 'c', in total we have visited node 'a' then node 'b' followed by node 'c', in other words key 'abc' is visited completely and hence we stop at node 'c' and return the value stored at this node.   
  
We will now go through the precise insertion and search algorithms using the same example.**

# Algorithm/Insights

### **Construction:  Constructing a trie is equivalent to constructing a root node with following 'TrieNode' definition. this.root = new TrieNode(false, NON\_VALUE); We pass isLeafNode = false, value = NON\_VALUE while constructing this root. class TrieNode {     boolean isLeafNode;     int value;          TrieNode[] children;                  TrieNode(boolean isLeafNode, int value)     {         this.value = value;         this.isLeafNode = isLeafNode;         children = new TrieNode[ALPHABET\_SIZE];     } } Insertion: Let's say we want to insert a key-value pair ("abc",1) into the trie. 1. We go to root node. 2. Get the index of the first character ('a') of "abc". That would be 0 in our alphabet system. 3. Go to 0th child of root. Because 0th child is null we first construct a TrieNode to which this 0th child would point. This newly constructed node would be node 'a'. We mark this node as current node. 4. Now we get the index of the second character of "abc". That would be 1 and therefore we go to 1st child of current node(from step #3).  5. Here again, 1st child is null. We create a new TrieNode which would be node 'b'. We mark this node as current node. 6. Now we get the index of the third character of "abc". That would be 2 and therefore we go to 2nd child current node(from step#5). 7. Here again, 2nd child is null. We create a new TrieNode which would be node 'c'. We mark this node as current node. 8. At this step, we are done reading all the characters of the given key. Hence we mark the current node that is node 'c' as leaf node and store value 1 at this leaf node. Now at this point let's say we want to insert a key-value pair ("abb",9) into the trie. 1. We go to root node. 2. Get the index of the first character of "abb". That would be 0 in our alphabet system. 3. Go to 0th child of root. Now as you can notice, 0th child won't be null since we have constructed node 'a' in the previous insertion sequence. We mark this node 'a' as the current node. 4. Now we get the index of the second character('b') of "abb". That would be 1, we go to 1st child of current node(from step #3). 5. 1st child of current node which is node 'b' is not null. We mark this node 'b' as current node. We now get the index of the last character ('b') of "abb". That index would be 1 and hence we go to 1st child of current node which is null. We create a new TrieNode which would be node 'b'. Current node now points to this newly created node. 6. We are done reading all characters if given key "abb". We mark current node as leaf node and store value 9 in it. Hopefully, these steps will help to understand the insertion algorithm better. Search algorithm steps: Example-1 : searching for non-existing key "ac" 1. Go to root node. 2. Pick the first character of key "ac" which would be 'a'. Find out its index using alphabet system in use. 3. Index returned would be 0, go to 0th child of root which is node 'a'. Mark this node as current node. 4. Pick the second character of key "ac" which would be 'c'. Its index would be 2 and therefore we go to In the previous post, we have seen how can we insert and retrieve keys for trie data structure. In this post, we will discuss how to delete keys from trie.**

### **Background: A trie is a data structure used for efficient retrieval of data associated with keys. If key is of length n, then using trie, worst case time complexity for searching the record associated with this key is O(n). Insertion and deletion of (key, record) pair also takes O(n) time in worst case.**

### **You can see in the below picture as to how a trie data structure looks like for key, value pairs ("abc",1),("xy",2),("xyz",5),("abb",9)("xyzb",8), ("word",5).**

### **You might want to visit previous post for more details about trie data structure,constructing a trie, insertion and search algorithms, its comparison with other data structures.**

### **Algorithm requirements for deleting key 'k':**

### **1. If key 'k' is not present in trie, then we should not modify trie in any way.**

### **2. If key 'k' is not a prefix nor a suffix of any other key and nodes of key 'k' are not part of any other key then all the nodes starting from root node(excluding root node) to leaf node of key 'k' should be deleted. For example, in the above trie if we were asked to delete key - "word", then nodes 'w','o','r','d' should be deleted.**

### **3. If key 'k' is a prefix of some other key, then leaf node corresponding to key 'k' should be marked as 'not a leaf node'. No node should be deleted in this case. For example, in the above trie if we have to delete key - "xyz", then without deleting any node we have to simply mark node 'z' as 'not a leaf node' and change its value to "NON\_VALUE"**

### **4. If key 'k' is a suffix of some other key 'k1', then all nodes of key 'k' which are not part of key 'k1' should be deleted.**

### **For example, in the above trie if we were to delete key - "xyzb", then we should only delete node "b" of key "xyzb" since other nodes of this key are also part of key "xyz".**

### **5. If key 'k' is not a prefix nor a suffix of any other key but some nodes of key 'k' are shared with some other key 'k1', then nodes of key 'k' which are not common to any other key should be deleted and shared nodes should be kept intact. For example, in the above trie if we have to delete key "abc" which shares node 'a', node 'b' with key "abb", then the algorithm should delete only node 'c' of key "abc" and should not delete node 'a' and node 'b'. 2nd child of current node.  5. Now at this point, we find out that 2nd child of current node is null. If you notice, from insertion algorithm of a given key, no node in the key-path from the root could be null. If it is null then that implies that this key was never inserted in the trie. And therefore in such cases, we return 'KEY\_NOT\_FOUND'.  Example-2 : searching for an existing key "abb" The steps are very similar to example-1. We keep on reading characters of given key and according to indices of characters, travel from root node to node 'b' which is at level-3 (if root is at level-0). At this point we would have read all the characters of the key "abb" and hence we return the value stored at this node.         Please checkout code snippet and algorithm visualization section for more details of the algorithm.**

Trie Data Structure | Delete

In the previous post, we have seen how can we insert and retrieve keys for trie data structure. In this post, we will discuss how to delete keys from trie.

Background: A trie is a data structure used for efficient retrieval of data associated with keys. If key is of length n, then using trie, worst case time complexity for searching the record associated with this key is O(n). Insertion and deletion of (key, record) pair also takes O(n) time in worst case.

You can see in the below picture as to how a trie data structure looks like for key, value pairs ("abc",1),("xy",2),("xyz",5),("abb",9)("xyzb",8), ("word",5).

You might want to visit previous post for more details about trie data structure,constructing a trie, insertion and search algorithms, its comparison with other data structures.

Algorithm requirements for deleting key 'k':

1. If key 'k' is not present in trie, then we should not modify trie in any way.

2. If key 'k' is not a prefix nor a suffix of any other key and nodes of key 'k' are not part of any other key then all the nodes starting from root node(excluding root node) to leaf node of key 'k' should be delete In the previous post, we have seen how can we insert and retrieve keys for trie data structure. In this post, we will discuss how to delete keys from trie.

Background: A trie is a data structure used for efficient retrieval of data associated with keys. If key is of length n, then using trie, worst case time complexity for searching the record associated with this key is O(n). Insertion and deletion of (key, record) pair also takes O(n) time in worst case.

You can see in the below picture as to how a trie data structure looks like for key, value pairs ("abc",1),("xy",2),("xyz",5),("abb",9)("xyzb",8), ("word",5).

You might want to visit previous post for more details about trie data structure,constructing a trie, insertion and search algorithms, its comparison with other data structures.

Algorithm requirements for deleting key 'k':

1. If key 'k' is not present in trie, then we should not modify trie in any way.

2. If key 'k' is not a prefix nor a suffix of any other key and nodes of key 'k' are not part of any other key then all the nodes starting from root node(excluding root node) to leaf node of key 'k' should be deleted. For example, in the above trie if we were asked to delete key - "word", then nodes 'w','o','r','d' should be deleted.

3. If key 'k' is a prefix of some other key, then leaf node corresponding to key 'k' should be marked as 'not a leaf node'. No node should be deleted in this case. For example, in the above trie if we have to delete key - "xyz", then without deleting any node we have to simply mark node 'z' as 'not a leaf node' and change its value to "NON\_VALUE"

4. If key 'k' is a suffix of some other key 'k1', then all nodes of key 'k' which are not part of key 'k1' should be deleted.

For example, in the above trie if we were to delete key - "xyzb", then we should only delete node "b" of key "xyzb" since other nodes of this key are also part of key "xyz".

5. If key 'k' is not a prefix nor a suffix of any other key but some nodes of key 'k' are shared with some other key 'k1', then nodes of key 'k' which are not common to any other key should be deleted and shared nodes should be kept intact. For example, in the above trie if we have to delete key "abc" which shares node 'a', node 'b' with key "abb", then the algorithm should delete only node 'c' of key "abc" and should not delete node 'a' and node 'b'. d. For example, in the above trie if we were asked to delete key - "word", then nodes 'w','o','r','d' should be deleted.

3. If key 'k' is a prefix of some other key, then leaf node corresponding to key 'k' should be marked as 'not a leaf node'. No node should be deleted in this case. For example, in the above trie if we have to delete key - "xyz", then without deleting any node we have to simply mark node 'z' as 'not a leaf node' and change its value to "NON\_VALUE"

4. If key 'k' is a suffix of some other key 'k1', then all nodes of key 'k' which are not part of key 'k1' should be deleted.

For example, in the above trie if we were to delete key - "xyzb", then we should only delete node "b" of key "xyzb" since other nodes of this key are also part of key "xyz".

5. If key 'k' is not a prefix nor a suffix of any other key but some nodes of key 'k' are shared with some other key 'k1', then nodes of key 'k' which are not common to any other key should be deleted and shared nodes should be kept intact. For example, in the above trie if we have to delete key "abc" which shares node 'a', node 'b' with key "abb", then the algorithm should delete only node 'c' of key "abc" and should not delete node 'a' and node 'b'. Tricky part:

When we make currentNode = null for deleting the current node, current node won't be deleted since we are making only the local reference to the node as null. Note that this node still has got another reference in terms of parent node's child reference. To delete the node completely, what we do is once a child returns deletedSelf as true, we make child reference = null as well. Code for this is code snippet is - if (childDeleted) currentNode.children[getIndex(key.charAt(level))] = null;

Example run through:

Now let's try to understand this algorithm using an example where we delete key "abc" in the trie shown in diagram above.

1. We make a call delete("abc") call which in turn calls deleteHelper(key = "abc", currentNode = root, length = 3, level = 0).

2. In deleteHelper(), we make a recursive call deleteHelper(key = "abc", currentNode = root.children[(getIndex('a'))], length = 3, level = level + 1) to reach node 'a' at level 1.

3. At this step, we again make a recursive call deleteHelper(key = "abc", currentNode = root.children[(getIndex('b'))], length = 3, level = level + 1) to reach node 'b' at level 2.

4. At this step, we again make a recursive call deleteHelper(key = "abc", currentNode = root.children[(getIndex('c'))], length = 3, level = level + 1) to reach node 'c' at level 3.

5. Now at this point level is 3 which is also equal to length. This implies we have reached last node for given key "abc". This is the base case for this algorithm. We check if node 'c' at level 3 has any children. It does not have any children and hence we mark currentNode = null(which refers to node 'c') and return deletedSelf = true.

6. Step #5 returns to call from step #4 where currentNode is 'b'. Since child was deleted we mark currentNode.children[(getIndex('c'))] = null to delete all references to node 'c' at level 3. Then we check if this currentNode 'b' has any more children. It does have one more child 'b' at level 3 (part of "abb") which implies that this node is part of other key and hence we do not delete currentNode and return deletedSelf = false.

7. Step #6 returns to call from step #3. Here currentNode is 'a'. Since child is not deleted, this currentNode also cannot be deleted and we simply return with deletedSelf = false.

8. Same return sequence is repeated until the recursion ends by a return call to function which had curreneNode as root.

At the end of this call sequence, node 'c' at level-3 would have been deleted.

Hopefully, this example will help to understand the algorithm more clearly.

Please checkout code snippet and algorithm visualization section for more details of the algorithm.

Pattern matching using Trie

Given a pattern and text, find all the occurrences of pattern in given text. For example, in given text "banana" pattern "ana" occurs twice - starting at index 1 and starting at index 2.

Algorithm/Insights

This algorithm uses trie data structure for time efficient pattern matching for a given pattern in given text. You might want to visit previous posts for more details about trie data structure,constructing a trie, insertion and search algorithms, its comparison with other data structures. In the previous posts, we have also discussed KMP algorithm for pattern matching.

The first step of this algorithm is to construct a trie from given text. To construct this trie, all the suffixes of text are inserted into the trie. For example, if given text is "banana" then all suffixes that is "banana","anana","nana","ana","na","a" are inserted. The trie looks like below image after inserting these suffixes.

At the time of insertion, each node also stores the indices where the character corresponding to that node occurs in the text and that character is the last character of the sub-string starting from root node. Let's call these indices magic-indices. For example, node 'b' stores index 0 since it occurs at index 0 in text "banana" for sub-string "b". Similarly, node 'a' at level-3 (in the path "root->a->n->a") stores indices 3,5 because letter 'a' is at index 3 and at index 5 in text "banana" and is the last character for sub-string "ana". Similarly, node 'n' at level-3 (path "root->n->a->n") stores index 4 because for substring "nan" it is the last character and it occurs at index 4 in text "banana". We will go through the modified insertion algorithm to understand how this can be easily done. But before that, we will go through the algorithm that uses this modified trie.

Once we have this modified trie in place, all we need to is to traverse the trie by taking out characters of the given pattern one by one and at the end of traversal, return the indices( with appropriate modifications) stored at node reached at the end of the traversal. If during this traversal, we reach the null node then we print out the message that this pattern does not exist in the text and return.

For example, to search for pattern "ana", we will start traversing the trie from root node and we will reach node 'a' at level-3 at the end of the traversal by taking path "root->a->n->a". The indices stored at this node 'a' are 3 and 5. We return these indices by subtracting (pattern length - 1) from each of these. Returned indices would then indicate the start index in the text for the pattern that we are searching for("ana"). For this example, returned indices would be 1 and 4 and as you can verify pattern "ana" is there in text "banana" starting at index 1 and at index 4.

Now let's look at the modified trie insertion algorithm which inserts suffixes of text in tree. This insertion algorithm takes an extra argument along with the key itself. The extra argument is the index in the text where this suffix starts from. Let's call this extra argument an offset for suffix. For example, suffix "anana" in "banana" would have an offset of 1 because it starts at index 1, which is passed to insert function along with suffix "anana". Now to add correct magic index at each node, all we need to do is to add this offset of suffix to the index of the character(corresponding to the node) in the suffix and store the result as a magic index at the node. In this example, while inserting suffix "anana", first node 'a' at level-1 stores the index 1 (offset=1, index=0). Then while inserting suffix "ana", the same first node 'a' at level-1 stores the index 3 (offset=3, index=0) and lastly while inserting suffix "a", it stores index 5(offset=3, index=0).

The time taken by this algorithm is O(n^2) where 'n' is the length of the text. This time is essentially taken to build the trie. Note that this is one time activity and subsequent searches of another pattern in this text would take O(m) time where m is the length of the pattern.

The worst case space complexity of this algorithm is O(n^2) where 'n' is the length of the text. This worst case occurs for text like "aaaaa".

Please checkout code snippet for more details of the algorithm

Longest Prefix Matching using Trie

Given a dictionary of words and an input string, find out the longest prefix of the input string which is also present in the given dictionary.

For example, if dictionary consists of following words -

word, cat, cam, name

And if the input string is 'camera', then output should be 'cam'. If the input string is 'cataract', output should be cat.

Algorithm/Insights

This algorithm uses trie data structure for efficient longest prefix matching. You might want to visit previous posts for more details about trie data structure,constructing a trie, insertion and search algorithms, its comparison with other data structures.

An example trie data structure for following (key, value) pairs

("cat",9),("cam",1),("word",5),("name",8)("na",2), ("nam",5).

First step of this algorithm is to construct a trie by inserting words of the given dictionary in it. Once this trie is constructed, we just need to traverse along the trie by taking out characters of the input string one by one. During this traversal, we keep track of the characters of the string that have been matched and update the longest prefix matched when a leaf node of trie is visited. Algorithm terminates if it comes across a null TrieNode during traversal or if the complete input string is matched. Null TrieNode visit indicates that there are no more nodes to visit in this path.

Let's run through an example for more clear understanding. Say we are given input string as "camera" and a dictionary with words - cat, cam, word, name, na, nam. We have chosen to use words which are keys in above shown trie. The step by step execution of the algorithm goes as follows for input string "camera"

1. Initialize charSequenceMatched and longestPrefixMatched to empty strings. Go to root node of the trie.

2. Make currentNode = root.children[getIndex('c')]. getIndex('c') would return 2 in the alphabet system of 'a'-'z'.

3. currentNode is now pointing to node 'c' at level-1. Update charSequenceMatched to string "c". Read the next character of the input string that is 'a' and update currentNode = currentNode.children[getIndex('a')].

4. currentNode is now pointing to node 'a' at level-2. Update charSequenceMatched to string "ca". Read the next character of the input string which is 'm' and update currentNode = currentNode.children[getIndex('m')].

5. currentNode is now pointing to node 'm' at level-3. Update charSequenceMatched to string "cam". This node is leafNode, longestPrefixMatched is updated to charSequenceMatched that is "cam". Read the next character of the input string which is 'e' and update currentNode = currentNode.children[getIndex('e')].

6. At this point, currentNode is null and hence the algorithm terminates and returns longestPrefixMatched which is "cam".

Please checkout code snippet and algorithm visualization section for more details of the algorithm.