**Trie/Prefix Tree**

A prefix tree is used to store strings based on prefixes. A prefix is a part of a string, starting from the beginning. For example, for the string 'fabacde', prefixes include 'f', 'fa', 'fab' and so on.

We make the tree using nodes. Say we are dealing with the English alphabets, from a to z, the small letters. Thus, every node must have a total of 26 pointers, one for each alphabet.

Say we have this list of strings:

algo, algea, also

Using this list, we get the following tree:

Note that we are not actually storing the alphabets in the nodes. The child number will be used to identify which alphabet is being pointed to. This will become clearer with the code.

Starting from the root, we go down to each of the leaves to form a word. This is pretty effective, since normally, we would need to store the words separately, but here we can store them in a combined manner. With very large lists, this will save a lot of memory space.

Now add a few more words to the list:

bat, bad, bata

The corresponding tree becomes:

We have found a problem. The words bat and bata are in the same line, i.e. bat is a prefix of the word bata. Starting from the root, we have no way to know that there was a word that ended at t. To keep track of this, we must use an extra flag, to denote the end of a word. Thus, the alphabets are being stored implicitly, using the edges, while the nodes themselves are being used to store the flag.

struct node  
{  
 bool endMark; *//word flag* node\* next[26]; *//array of pointers for alphabets* node () *//constructor* {  
 endMark = false;  
 for (int i=0; i<26; i++) next[i] = **NULL**;  
 }  
};

C++

From the main function, we can use the structure like this:

int main()  
{  
 node\* root = new node(); *//creating root* root->next[2]; *//accessing ‘c’ from the root*}

C++

Notice that the root itself does not provide any information, just a starting point.

Now for the insertion function:

void insert (char\* str, node\* x)  
{  
 int len = strlen(str);  
 for (int i=0; i<len; i++)  
 {  
 int id = str[i] - 'a'; *//finding which of the 26 nodes to use* if (x->next[id] == **NULL**) x->next[id] = new node();  
 *//if node did not exist, create it* x = x->next[id]; *//go to next node* }  
 x->endMark = true; *//set endMark at end of word*}

C++

The way this function has been implemented, which pointer to use is being decided depending on which character is in a particular position of the string. However, this makes use of ASCII codes and may not work if different characters from different places are being used. In the current implementation, even capital letters will cause the program to fail.

Now the searching function. This is simple enough. It just keeps following the input string down the tree. If a character cannot be found, then the string does not exist.

bool searchTree (char\* str, node\* x)  
{  
 int len = strlen(str);  
 for (int i=0; i<len; i++)  
 {  
 int id = str[i] – ‘a’;  
 if (x->next[id] == **NULL**) return false; *//word not found* else x = x->next[id];  
 }  
 return x->endMark; *//check if prefix or word*}

C++

The search function is more efficient than normal string searching. With a normal search, we would need to compare the input with every available string, which can take a very long time with a large database. Here however, we only need to perform the number of operations equal to the length of the input string. The function can also be used to detect invalid searches.

We could have another implementation of the search function that does not perform the check for endMark at the end. There, we would get a true result if the input was a word or a prefix for some word.

In order to delete a word, we need to first ensure that the word exists. This means following the tree using the input, like we did with the search function, until we reach the end of the input, and then confirming that we have an endMark there. Next, we delete the endMark, and from that node we start travelling up, checking each parent. If a parent has an endMark, it means some other word ends at that node and we cannot delete that node or any of its parents as they will also be a part of the word. If a parent has other pointers in the next array, that means it is parent of other words for which it is part of a prefix, so we cannot delete it in this case either. Otherwise, we can delete that node. We must be careful here to not delete the root. The deletion function is not really all that important to prefix trees. Note that we need to add a parent pointer and edit our insertion code to keep track of parents.

void remove (char\* str, node\* x)  
{  
 int len = strlen(str);  
 for (int i=0; i<len; i++)  
 {  
 int id = str[i] – ‘a’;  
 if (x->next[id] == **NULL**) return; *//word does not exist* else x = x->next[id];  
 }  
 x->endMark = false; *//removing marker for word* int j = len – 1;  
 *//will be used to go from end to beginning of input* while (x != **NULL**)  
 {  
 if (x->endMark == true) return;*//another word ends here* for (int i=0; i<26; i++) if (x->next[i] != **NULL**) return;  
 *//node is part of other words* x = x->parent;  
 if (x == **NULL**) return; *//parent of root reached* int id = str[j] – ‘a’; *//locating alphabet of word* x->next[id] = **NULL**; *//removing alphabet* j--;  
 }  
}

C++

We can find the longest common prefix by starting at a particular node and going downwards until we find a node with an endMark or one for which there is more than one next node.

char\* longestPrefix(node\* x, int j = 0, char\* oldarr = new char)  
*//allocate memory to array by default when no array present*{  
 char\* arr = oldarr;  
 *//copy array so that allocated memory can be deleted* delete oldarr;  
 arr[j] = ‘\0’; *//arr is an array, end is marked* if (x->endMark) return arr;  
 *//return the array if a word ends* int id, flag = 0; *//flag for multiple nodes* for (int i=0; i<26; i++)  
 {  
 if (x->next[i] != **NULL** && flag == 0)  
 {  
 flag = 1;  
 id = i; *//id tracks next child position for single nodes* }  
 else if(x->next[i] != **NULL** && flag == 1) return arr;  
 *//if multiple nodes, return arr* }  
 arr[j] = id + ‘a’; *//add next child to array* return longestPrefix(x->next[id], j+1, arr);  
 *//recursive call on next child*}

C++

We can display all the words in the tree by recursively adding nodes to a queue and printing the queue if an endMark is found.

void display (node\* x, int j = -1, queue <char> arr = queue <char>())  
*//new queue by default when no queue is present*{  
 if (j != -1) arr.push(j+'a');  
 *//push character into queue if not at root* if (x->endMark == true)  
 {  
 queue <char> print = arr; *//make a copy* while (!print.empty()) *//print the copy* {  
 cout<<print.front();  
 print.pop();  
 }  
 cout<<endl;  
 }  
 for (int i=0; i<26; i++)  
 if (x->next[i] != **NULL**) display(x->next[i], i, arr);  
 *//call recursively on all children*}

C++

We can create an autocomplete function by finding the last node of the input, adding the nodes before it to a queue and calling the display function on the last node.

void autoComplete (char\* arr, node\* x)  
{  
 int len = strlen(arr);  
 int id;  
 for (int i=0; i<len; i++)  
 {  
 id = arr[i] - 'a'; *//next child index* if (x->next[id] == **NULL**) return; *//input not present* else x = x->next[id];  
 }  
 queue <char> displayarr; *//queue for display function* for (int i=0; i<len-1; i++) displayarr.push(arr[i]);  
 *//adding prior alphabets to queue* display(x, id, displayarr);  
 *//print from node cur, with queue containing prior alphabets*}

C++

We can find how many words start with some prefix (how many results the autocomplete function will give) by keeping an extra integer at every node that keeps track of the number of words that include that node. Note that we need to edit the insert and remove functions. In the insert function, we increment the child count of each node as we go down the tree. In the remove function, we decrement the count going up the tree.

int countChildren (char\* arr, node\* x) {  
 int id, len = strlen(arr);  
 for (int i=0; i<len; i++)  
 {  
 id = arr[i] - 'a';  
 if (x->next[id] == **NULL**) return 0;  
 else x = x->next[id];  
 }  
 return x->words;  
}

C++

We can also have binary tries, which are just binary trees. They store numbers in binary format. The processes for insertion, removal and search are the same, and even easier to visualize since there are only two children per node. They must be edited however, to accommodate numbers instead of alphabets.

We can have a slightly odd function with a binary trie. We can find the maximum XOR for an input. Say we have an input 110, and the tree only contains the number 101. The result of this function will then be 011, i.e. 3.

For every digit of the input, we go to the opposite node, i.e. for the first digit, 1, we go to 0, for the second digit, 1, we go to 0 and so on. If the digit we tried to go to, i.e. the opposite digit, is not **NULL** and we can go to it, then we add 1 to a result array. If the digit is not available in the tree, then we will add 0 to the result array and will be forced to go to the only available node. Finally, we can display the result array.

char\* maxXOR (char\* str, node\* x)  
{  
 char\* res;  
 int i = 0, len = strlen(str);  
 for (i=0; i<len; i++)  
 {  
 int id = str[i] – '0';  
 if (x->next[!id] != **NULL**) *//if opposite digit is available* {  
 x = x->next[!id];  
 res[i] = '1'; *//result for digit is 1* }  
 else  
 {  
 x = x->next[id];  
 res[i] = '0'; *//result for digit is 0* }  
 }  
 res[i] = '\0'; *//ending the string* return res;  
}

C++

Note that in order for this to work, we need to have a fixed length for the inputs, and every binary number needs to include the preceding 0s.