Palindromic tree

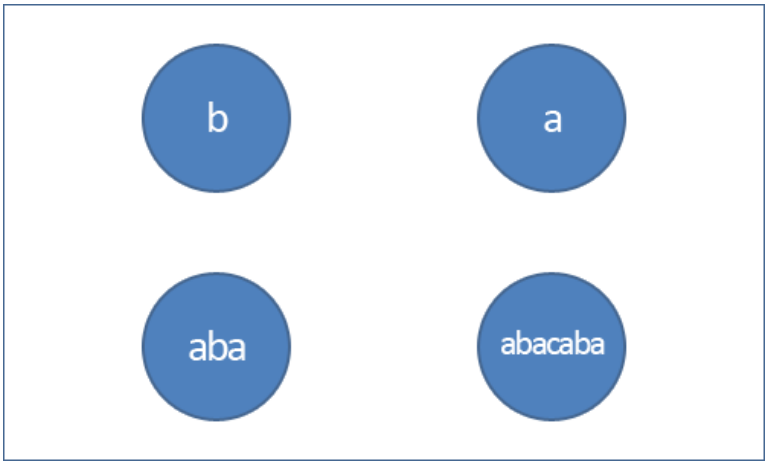
25 September 2014

This blog post will describe palindromic tree — a nice data structure allowing to solve some interesting problems involving palindromes. This data structure is quite new, and even Google still doesn't provide any information about it. As far as I know the name "palindromic tree" is also not an official name yet. Would you come up with a better name after reading this article?

Thanks to Mikhail Rubinchik, inventor of this data structure, for presenting it during Petrozavodsk Summer Camp 2014 and thus making creation of this article possible.

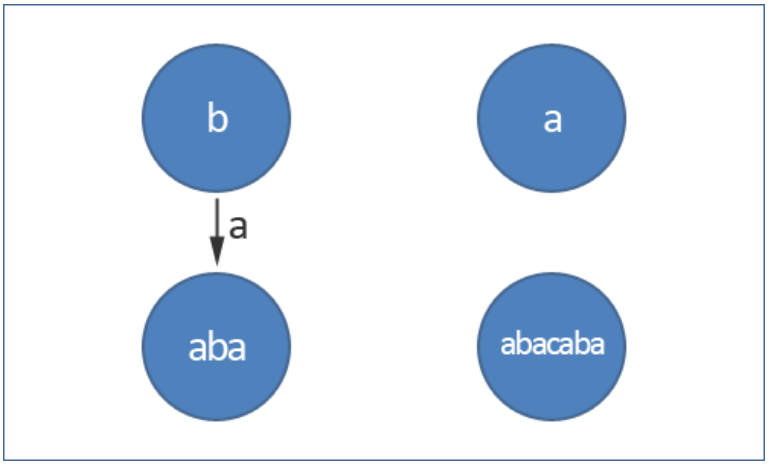
Structure of palindromic tree

As in any tree-like data structure in palindromic tree we have nodes. Here each node represents a palindrome.



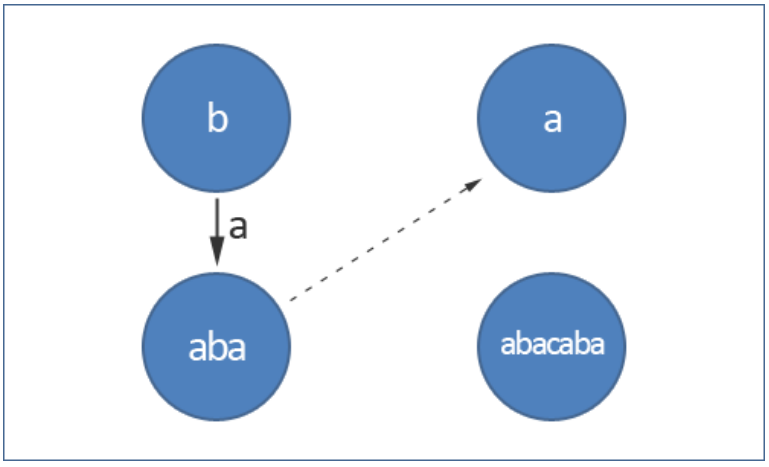
Example of 4 nodes of palindromic tree.

In addition to nodes we also have edges between them. Directed edge between nodes *u* and *v* marked with some letter - let's say X - means that we can obtain the palindrome of node *v* by adding letter X to the both sides of the palindrome of node *u*. Please refer to the picture below for better understanding.



We get palindrome aba by adding letter a to the both sides of palindrome b.

And finally we also have suffix links. A suffix link from node *u* goes to such node *w*, that the palindrome of node *w* is the largest palindrome which is also a proper suffix of the palindrome of node *u* (suffix is a substring of the string which contains its last character. Proper suffix is such suffix, which is not equal to the whole string itself). Hereafter we will call the largest palindrome which is also a proper suffix of a string just largest suffix-palindrome of a string.



We added suffix link (dashed line) from aba to a because a is the largest suffix-palindrome of string aba.

Name palindromic tree is a bit confusing, since this data structure is not an ordinary tree, because here we have two roots. First root will contain a fictive palindrome string of length -1. This strange root is created for convenience — we will agree that adding some letter to both sides of this "string" will produce a new string of length one and consisting of exactly that letter. Second root will just represent an empty string, which is also a palindrome, and will be of length 0. We also set suffix links of both roots going to the root with length -1.

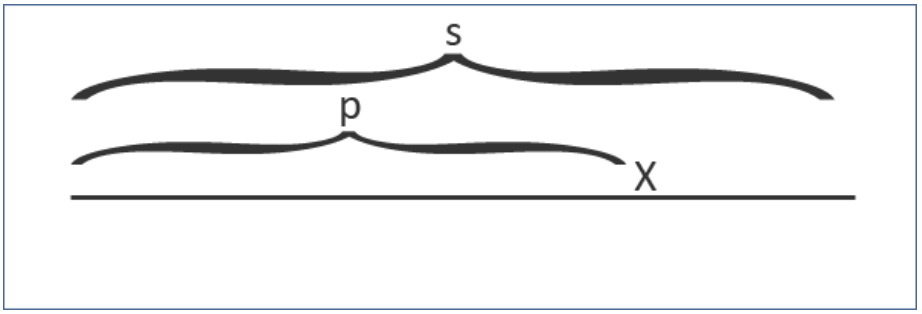
Note that we don't actually store palindrome strings in the nodes, because it would take too much memory. Implementation of node structure will consist only of its palindrome length, edges on all letters of the given alphabet (possibly with some null edges) and suffix link.

Construction of palindromic tree

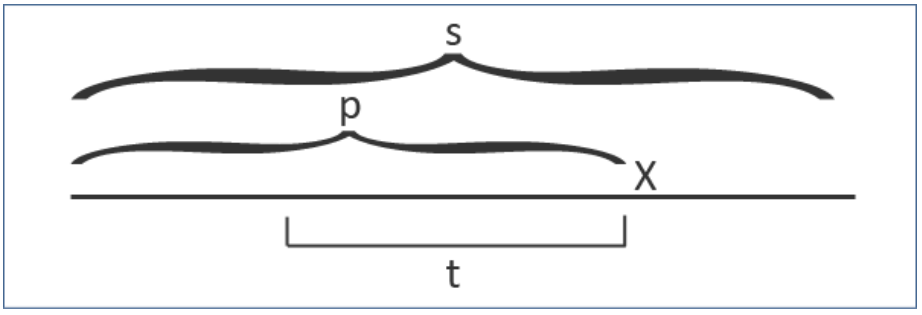
Let's learn how to build the palindromic tree for some given string *s*, that is, a palindromic tree which will consist of all different palindrome substrings of *s*. Because of the fact that any string of length *n* has not more than *n* different

palindrome substrings (try to prove this yourself if you didn't know it before, it is not hard) our palindromic tree will also have not more than length of the given string plus two nodes, where two extra nodes are roots.

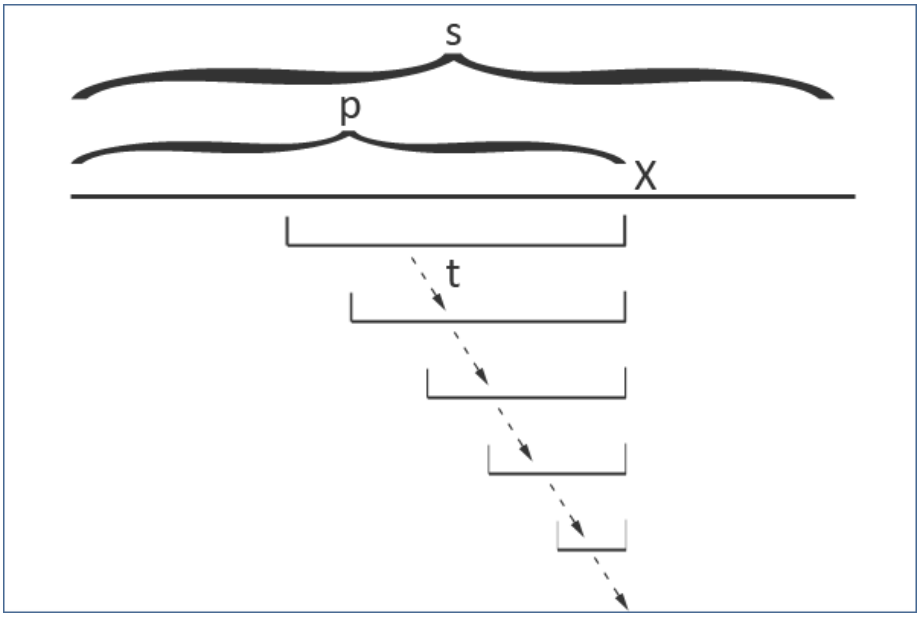
We will process the string letter by letter. Suppose we already processed some prefix (beginning of the string) *p*, and now we want to add the next letter of the string, say x.



We will also maintain the largest suffix-palindrome of the processed prefix *p*, let's call it *t*.

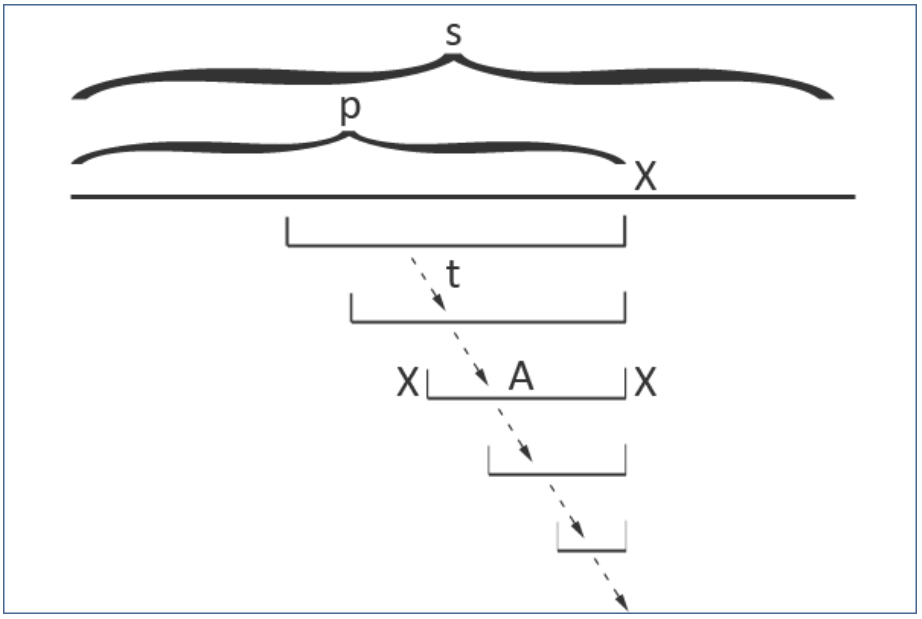


Since *t* is in already processed prefix, it corresponds to some node of the palindromic tree. This node has a suffix link to some other existing node, which also has a suffix link and so on.



Chain of suffix links from *t*.

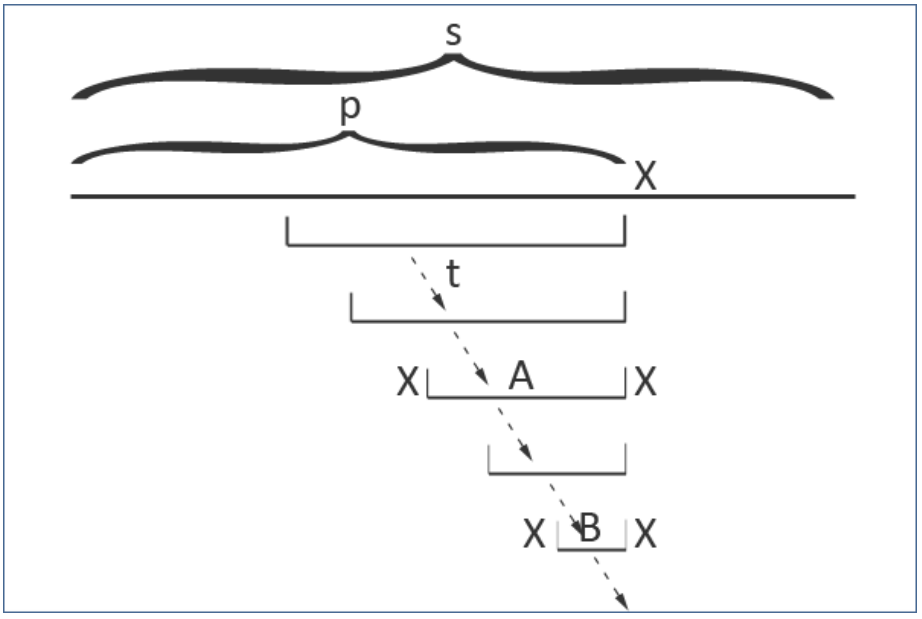
Now let's find the suffix-palindrome of new prefix consisting of *p* plus x. This suffix-palindrome will look like xAx, where A is some string, possibly empty (or it is our fictive root string of length -1, when suffix-palindrome is just x). Since xAx is a palindrome, A is also a palindrome, and moreover it is some suffix of *p*, therefore it can be reached from *t* by suffix links.



The string xAx is the only palindrome substring of *p* + x, that we possibly never saw in *p*. To understand why this is so, notice that all new palindrome substrings which we didn't meet in *p* must end on that last letter x, and therefore be the suffix-palindromes of *p* + x. Because xAx is the largest suffix-palindrome of *p* + x, all other smaller suffix-palindromes of it can be found in some prefix of xAx (since for any suffix of the palindrome there is a corresponding similar prefix), and therefore we already saw them in *p*.

So, to process new letter x we just go along suffix links of *t* until we find an appropriate A (and we always find some, possibly of length -1 if we trace suffix links back to the roots). Then we check if there is an edge on letter x from the node corresponding to A, and if not, set this edge going to the new node corresponding to xAx.

What about suffix link of the node xAx? If this node already existed before, the suffix link was also already set and we don't need to do anything. If not, then we need to find the largest suffix-palindrome of xAx, which will be in a form of xBx, where B is some string, possibly empty. By the same logic that we used before, B is a suffix-palindrome of *p* and can be reached from *t* by suffix links.



We ended up with the following algorithm of construction of the palindromic tree. Process the string letter by letter, always maintaining the largest suffix-palindrome *t* of the processed prefix (initially *t* is an empty string). When adding new letter *x*, we need to go along suffix link chain from *t*, until we find some palindrome A, which can be extended with x on both sides. The string xAx will be our new largest suffix-palindrome and the only candidate for the new node of the tree. To obtain its suffix link we need to continue going along suffix link chain, until we find some another palindrome B, which also can be extended with x on both sides, and the suffix link from xAx will go to xBx.

For better understanding of how it is all implemented, take a look on my code of the palindromic tree (ignore variable num in the node structure, it is specific to the problem which this particular code solves). As you can see, the code is not very long and not very hard to write.

Time complexity

To understand how long this algorithm works let's take a look at what happens as we build the palindromic tree. We may notice that as we process string letter by letter, *left* bound of the largest suffix-palindrome of the processed prefix goes only to the right as we iterate through suffix link chain. This bound can move to the right only n times, where n is the length of the given string. The same logic applies to the left bound of the string to which the suffix link of the largest suffix-palindrome points. So, overall time complexity of this algorithm is *O(length of the string)* or *O(n)*.

Applications

Let's take a look at some problems that can be solved using the palindromic tree.

How many new palindromes are added?

We have a problem that asks how many new palindrome substrings are added to the string if we append a letter to its end. For example appending letter *a* to the string *aba* (which already has subpalindromes a, b and aba) adds new palindrome aa.

We know that the number of new subpalindromes can be either 0 or 1. The solution of this problem is straightforward — we need to build the palindromic tree letter by letter, and for any new letter we can answer how many new palindromes were added just by inspecting the number of newly created nodes of the tree.

The number of palindrome substrings

In this problem we have to answer how many palindrome substrings the given string has. For example the string *aba* has four — two times a, one time b and one time aba.

We will solve this problem along with the construction of the palindromic tree for the given string. When we process a new letter we add to the answer all palindromes that contain this new letter. But these palindromes are the new longest suffix-palindrome *t* (which contains new letter) and all palindromes that can be reached from it by suffix links. To quickly find the number of them let's store in each node the length of the suffix link chain from it (including this node), then we need just to add this value for *t* to the answer as we process new letter. This value itself is calculated very simple: for roots it is zero, for other nodes it is just this value in the suffix link node (which was created and handled earlier) plus one.

You may see the code solving this problem here.

This problem can also be solved by Manacher's algorithm that also runs in *O(n)* (code of Manacher's algorithm solving this problem is available here). However, in my opinion, the palindromic tree is easier to write and it can be extended for many other types of problems.

The number of occurrences of the palindromes

Another problem asks for the number of occurrences of each subpalindrome in the string, and it also can be solved using the palindromic tree.

Note that when we add a new letter, it increases the number of occurrences of the new longest suffix-palindrome *t* (which contains new letter) and all palindromes that can be reached from it by suffix links.

To quickly update these values we will increase some lazy flag in *t* node, which means that we later increase the number of occurrences of it and all nodes reachable from *t* node by suffix links. After we are done with the construction of the palindromic tree, we iterate through the list of nodes from the end to the beginning (from the lately created nodes to the early ones) and propagate this lazy flag from the node to its suffix link node — update the number of occurrences of the current node by the value of the flag and then add this value to the flag of the current node's suffix link.

In the end, we get the palindromic tree with each node containing the number of its occurrences and the construction runtime remains the same *O(n)*.

Conclusion

The code of palindromic tree is available here. You can practice palindromic tree in online judges on this and this problems.

All comments, questions and notices about any errors in the text are more than welcome.

P.S. Комментарии и вопросы на русском языке также приветствуются :)