In the case of an odd length palindrome, the middle character will be a character of the original string, surrounded by "#". In the case of an even length palindrome, the middle character will be a "#" character.

Methodology of Manacher's Algorithm:

1. Create an array or a list *(sChars)* of length *strLen* which is 2 * n + 3 (n being the length of the given string), to modify the given string.
2. Assign the first and last element of *sChars* to be "@" and "$", respectively.
3. Fill the blank spaces in *sChars* by characters of the given string and "#" alternatively.
4. Declare variables
    - Implicating maximum detected length of palindrome substring *maxLen* = 0
    - Position from where to start searching start = 0
    - Highest position of the extreme right character of all detected palindromes *maxRight* = 0
    - Center of the detected palindrome center = 0.
5. Create an array or a list p to record the width of each palindrome about their center, center being the corresponding characters in *sChars*.
6. Create a for loop iterating from 1 to *strLen* - 1, with i incrementing in each iteration.
7. In each iteration, check if i < *maxRight*, if yes, then assign minimum of *maxRight - i* and *p[2 * center - i]* to *p[i].*
8. Nest a while loop inside the for loop, to count with width along the center, condition being, *sChars[i + p[i] + 1]* is equal to *sChars[i - p[i] - 1]* if yes, increment *p[i]* by 1.
9. To update center, check if *i + p[i]* is greater than *maxRight,* if yes, assign center to be 1, and *maxRight* to be *i + p[i]*
10. For updating the Maximum length detected, check if p[i] is greater than *maxLen*, if yes, then assign start to be *(i - p[i] - 1) / 2*, and *maxLen* to be *p[i].*
11. Come out of the for loop, and print the substring in the given string, starting from start and ending at *start + maxLen - 1*.

Note:

- *strLen* is the length of the modified (after inserting extra characters) list/array.
- *sChars* is the modified (after inserting extra characters) list/array.
- *maxLen* is the length of the longest palindrome detected.
- start is the position from which we have to search for a palindrome.
- *maxRight* is the position of the rightmost character of a palindrome.
- center is the center of a palindrome
- p is the list/array of the width of palindromes about their center, center being the corresponding characters in *sChars.*

Complexity Analysis:
The Manacher's Algorithm has a time complexity of O(n) whereas it requires more auxiliary space since it stores string linearly and stored the # character between every character and before / after the string.

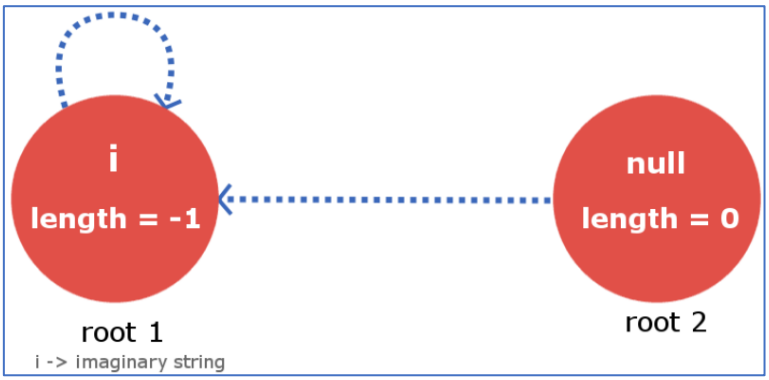
3. PALINDROMIC TREE

Palindromic Tree is a data structure which solves the longest palindromic substring problem in a much simpler way.

Structure:
Edges: The structure of a palindromic tree is similar to that of a directed graph. It is a merged structure of two trees which share common nodes. The nodes store palindromic substrings of a given string by storing their indices. There are two types of edges in the structure:

1. Insertion Edge – It is a weighted edge from node u to v with weight x indicating that node v is a palindromic substring formed by inserting character x before and after palindromic substring u. Thus, a node can have a maximum of 26 insertion edges.
2. Maximum Palindromic Suffix Edge – It is an unweighted edge which points to the maximum length palindromic suffix node of a string(node). It is also known as suffix edge.

Root Nodes and Convention: The tree contains 2 root nodes considering them roots of two separate trees. The first root refers a string of length -1 whereas the second root refers a null string of length 0. The first root has a self-loop suffix edge pointing to itself making its maximum palindromic substring imaginary which is thus justified. Whereas, the second root has a suffix edge connected to the first root indicating that there is no palindromic substring which has length less than 0.



Methodology of Building the Palindromic Tree:
A string is added to a palindromic tree character by character. On reaching the end of the string the tree will contain all distinct palindromes available in the string. While building the tree we must ensure that the rules of the edges are not violated.

If we are given a string s of length l and have inserted up to k characters into the tree, then insertion of the $(k+1)^{th}$ character indicates insertion of the node that is the longest palindrome ending at index (k+1). Now the longest palindromic substring will be of the form ('s[k+1]' + "X" + 's[k+1]') and X will itself be a palindrome. Thus a new node ('s[k+1]' + "X" + 's[k+1]') will be formed using a =n insertion edge with weight s[k+1] from node X.

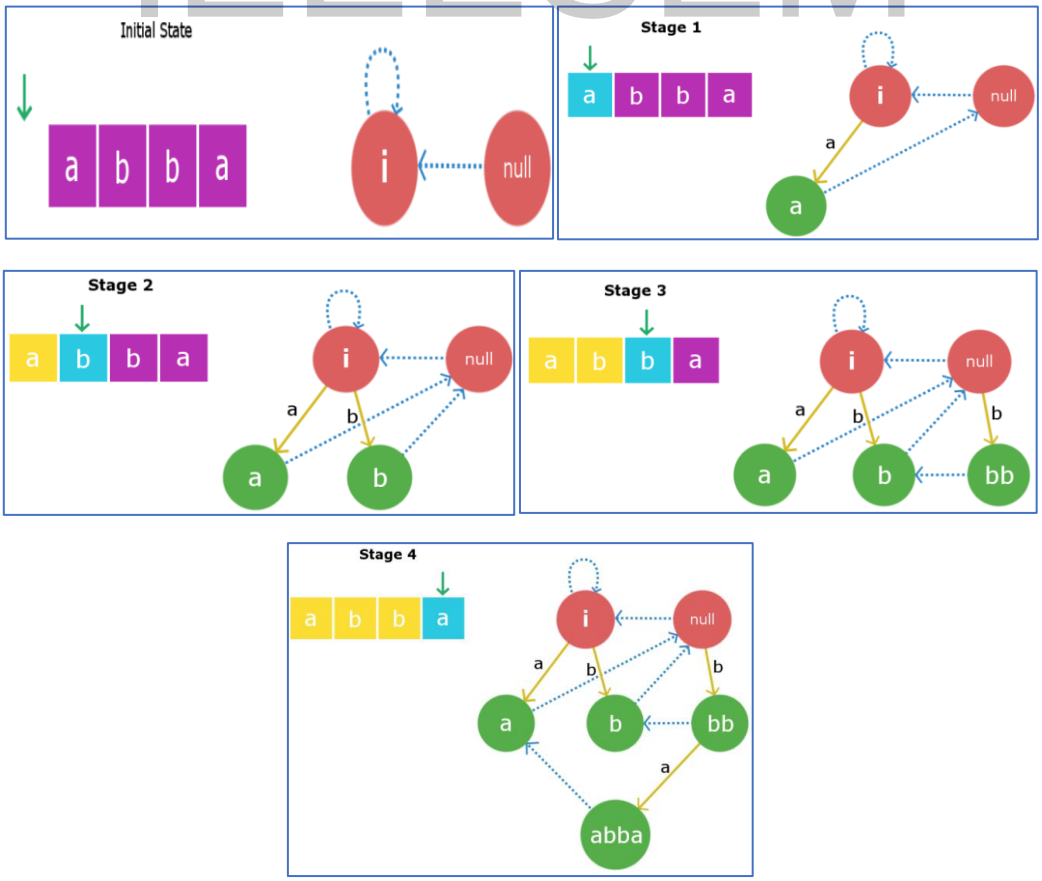
The main task to find string X in efficient time. This is done by moving down the suffix edges until we arrive at X. To find the required node that contains the string X the $(k+1)^{th}$ character is placed at end of every node that lies within suffix link chain and we check if the starting character is equal to the $(k+1)^{th}$ character.

As we have created a new node at this s[k+1] insertion, therefore we will also have to connect it with its suffix link child. Once, again to do so we will use the above down the suffix link iteration from node X to find some new string Y such that s[k+1] + Y + s[k+1] is a largest palindromic suffix for the newly created node. Once, we find it we will then connect the suffix link of our newly created node with the node Y.

If a node ('s[k+1]' + "X" + 's[k+1]') is already present in tree it is not added again only an insertion edge from X is added.

Odd Palindromic strings have insertion edges starting from First root. Since its length is -1 adding an insertion edge to it will yield a node with a string of length 1. Even Palindromic strings have insertion edges starting from Second root. Since its length is 0 adding an insertion edge to it will yield a node with a string of length 2. Then suffix edges are added, suffix edges of strings of length 1 as the second root and that of string of length 2 could be a string of length 1.

Example:



Functionality:
Node Structure: Start and end indexes of string in current node, length of string in current node, insertedge[26] (26 possible insert edges) and suffix edge.

Insertion:

1. Search for Node X such that s[k+1] X S[k+1] is maximum palindrome ending at position k+1 iterate down the suffix link of current Node to find X.
2. Now we have found X, check if s[k+1] X s[k+1] already exists or not. If yes, return.
3. Else, create new node. Make new Node as child of X with weight as s[k+1].
4. Calculate length, end and start points of new node.
5. Set the suffix edge for the newly created Node. Finding some String Y such that s[k+1] + Y + s[k+1] is longest possible palindromic suffix for newly created Node.
6. If new palindrome's length is 1 make its suffix link to be null string.
7. Else find string Y and link current Node's suffix link with s[k+1]+Y+s[k+1].

Search for Palindromic substring:

1. If length of palindrome is odd start search from first root else from second root.
2. Traverse through insertion edges until length of palindrome you are searching for is equal to length of string of the node you have reached.
3. Compare for equality.

Complexity Analysis:
The time complexity for the building process will be O(k*n), here n is the length of the string and k is the extra iterations required to find the string X and string Y in the suffix links every time we insert a character. The approximation of constant k is very less making the time complexity equal to O(n) where n is the length of the string.

Time complexity of searching a palindrome is O(n/2) since every insertion edge adds a character to the start and end of the previous palindromic string. Here n is the size of string to be searched.

The Auxiliary Space required will increase with the increase in number of palindromes in the string.

4. REFERENCES

[1] Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai (2021), Computing longest palindromic substring after single-character or block-wise edits, Theoretical Computer Science 859(8–10).

[2] Sumaiya Iqbal, Mohammad Sohel Rahman, Hasan Mahbubul, Shihabur Rahman Chowdhury (2012), Computing a Longest Common Palindromic Subsequence, Conference: 23rd International Workshop, IWOCA.

[3] Shoupu Wan, An Efficient Implementation of Manacher's Algorithm, arXiv:2003.08211v2 [cs.DS] 19 Mar 2020.

[4] Elizabeth (2021), Manacher's Algorithm: Longest Palindromic Substring, www.medium.com.

[5] Mikhail Rubinchik, Arseny M. Shur (2015), Eertree: An Efficient Data Structure for Processing Palindromes in Strings, arXiv:1506.04862v1 [cs.DS] 16 Jun 2015.