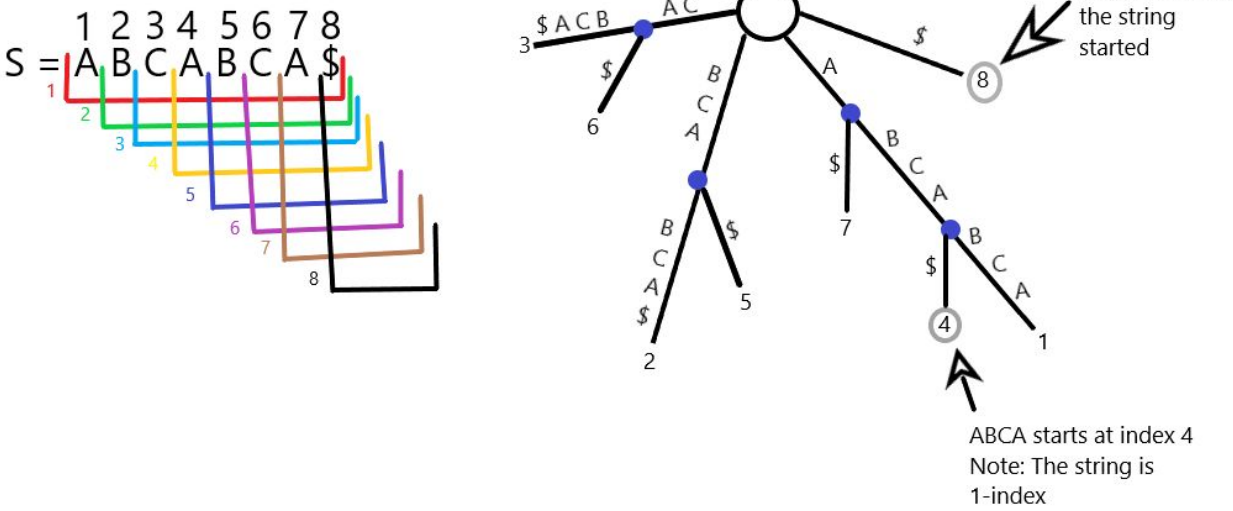
Suffix Tree

Basic - How to Build:



Note: $ is used as a delimiter

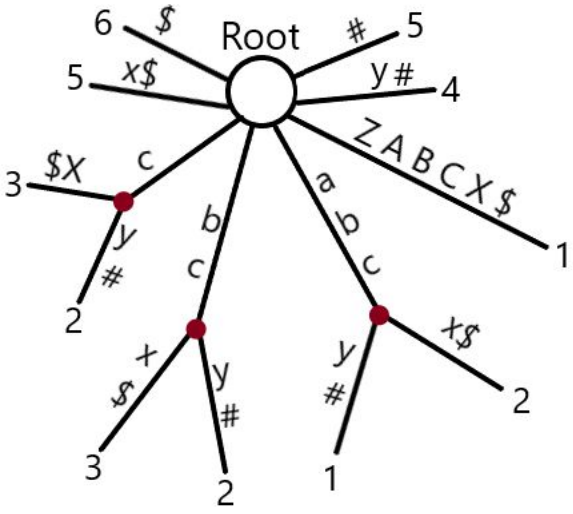
1) You build the tree from left to right with these suffixes:
    a) ABCABCA$
    b) BCABCA$
    c) CABCA$
    d) ABCA$
2) Create new branch whenever you see a new letter: A vs B vs C
3) Create a new leaf node upon seeing a suffix that starts with an existing branch
    a) ABCABCA$ vs ABCA$ vs A$

Example Applications:

Longest Common Substring (LCS):

Str1 = zabcx Str2 = abcy Answer= abc

S = zabc$abcy#
1 2 3 4 5 1 2 3 4 5



How to find **Longest Common Substring:**

Steps:

1) Concatenate the two strings with individual delimiter such that S = zabcx$abcy#
2) Using depth-first search, look for the deepest node that has children leaf nodes with both delimiters. The answer is the one with the longest character length.
3) In the picture above, we have three nodes with deepest length: ‘c’, ‘bc’, ‘abc’. The answer will be ‘abc’ since it has the longest character length.

Longest Palindrome:

Using S = xabay$ as an example, we are looking to find the longest palindrome substring.

Steps:

1) While using two different delimiters, concatenate the string with the reverse of the string such that S = xabay$yabax#. Where $ and # are the delimiters.
2) Make a suffix tree similar to the LCS example above.
3) Look for the deepest node that has children leaf nodes with both delimiters. However, we also need to compare the **indices** of both the children.
4) If the **indices** are the same then we have the longest palindrome. If not then we go back one parent node and check if their children’s index numbers are the same.

Circular String Linearization:

What is it?

Circular String Linearization - Choose a place to cut **S** such that the resulting linear string is lexically smallest of all the n possible strings.

EX: S = gcttc
Possible strings from S are ‘gcttc’, ‘cttcg’, ‘ttcgc’, ‘tcgct’, and ‘cgctt’
Answer = cgctt ← lexically smallest out of all the possible strings above ↑

1) Cut the string arbitrarily to a linear string **L,** then concat **L** with **L** to build a suffix tree with **LL**. **EX:** S = gcttc so **LL** = gcttcgcttc
2) Traverse the tree with the rule that at every node, traverse the edge whose first character is lexically smallest.
3) Traverse until you reach the depth that equals the string length of **L**. (In this case the depth to stop at would be 5).
4) Any leaf in the subtree at that point can be used to cut S

Random Fact: This can be useful for chemistry with an O(n) look up time for circular molecules.

How We Test The Implementation:

1) Create a naive linearization where we randomly create a string and append it to itself.
2) Do all rotation of it and put each possible string into an array and sort it.
3) naiveArray[0] should be the smallest lexical string
4) Compare Array[0] with our implementation of circular string linearization.
    a) Via ASSERT_EQ(suffixTreeAnswer, naiveAnswer);

Suffix Array:

What is it?

1) Purpose: Reduce space complexity for some applications. Can be used to solve exact matching problem or substring problem with almost similar efficiency as suffix tree but more space efficient
2) Suffix Array - is an array that holds all starting index of the string’s suffix in lexicographical order
3) **NOTE**: The suffix array should only hold **integers (ints)**

S = mississippi

suffix array = [11,8,5,2,1,10,9,7,4,6,3]

Ex: Find all suffixes that starts with issi
P = issi Answer = it appears at pos2&5
S = mississippi

NOTE: Any pattern in S will be adjacent in Array
Convert suffix tree to suffix array can be done in linear time

11: i
8: ippi
5: issippi
2: ississippi
1: mississippi
10: pi
9: ppi
7: sippi
4: sissippi
6: ssippi
3: ssissippi

How We Test The Implementation:

1) We ran a naive algorithm that computes all suffixes in a string array and then sort it.
    a) naiveSuffix[0] = i, naiveSuffix[1] = ippi, naiveSuffix[2] = issippi and so on.
2) We then created a naiveSuffixArray and computed the starting index of each suffix within the naiveSuffix array.
    a) naiveSuffixArray[0] = 11, naiveSuffixArray[1] = 8 and so on.
3) Compare all elements of naiveSuffixArray with our implementation of circular string linearization.

```
for(int j = 0; j< RANDOM_STRING_SIZE; j++)
   EXPECT_EQ(naiveArray[j], suffixArrayOutput[j]);
```