Suffix Tree Application 1 - Substring Check - GeeksforGeeks

Source: https://www.geeksforgeeks.org/suffix-tree-application-1-substring-check/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Suffix Tree Application 1 - Substring Check Last Updated : 23 Jul, 2025 Given a text string and a pattern string, check if a pattern exists in text or not. Few pattern searching algorithms (KMP , Rabin-Karp , Naive Algorithm , Finite Automata) are already discussed, which can be used for this check. Here we will discuss suffix tree based algorithm. As a prerequisite, we must know how to build a suffix tree in one or the other way. Once we have a suffix tree built for given text, we need to traverse the tree from root to leaf against the characters in pattern. If we do not fall off the tree (i.e. there is a path from root to leaf or somewhere in middle) while traversal, then pattern exists in text as a substring. Here we will build suffix tree using Ukkonen’s Algorithm, discussed already as below: Ukkonen’s Suffix Tree Construction – Part 1 Ukkonen’s Suffix Tree Construction – Part 2 Ukkonen’s Suffix Tree Construction – Part 3 Ukkonen’s Suffix Tree Construction – Part 4 Ukkonen’s Suffix Tree Construction – Part 5 Ukkonen’s Suffix Tree Construction – Part 6 The core traversal implementation for substring check, can be modified accordingly for suffix trees built by other algorithms. C++ // A C program for substring check using Ukkonen's Suffix // Tree Construction #include <bits/stdc++.h> using namespace std ; #define MAX_CHAR 256 struct SuffixTreeNode { struct SuffixTreeNode * children [MAX_CHAR]; // pointer to other node via suffix link struct SuffixTreeNode * suffixLink ; /*(start, end) interval specifies the edge, by which the node is connected to its parent node. Each edge will connect two nodes, one parent and one child, and (start, end) interval of a given edge will be stored in the child node. Let's say there are two nods A and B connected by an edge with indices (5, 8) then this indices (5, 8) will be stored in node B. */ int start ; int * end ; /*for leaf nodes, it stores the index of suffix for the path from root to leaf*/ int suffixIndex ; }; typedef struct SuffixTreeNode Node ; char text [100]; // Input string Node * root = NULL ; // Pointer to root node /*lastNewNode will point to newly created internal node, waiting for it's suffix link to be set, which might get a new suffix link (other than root) in next extension of same phase. lastNewNode will be set to NULL when last newly created internal node (if there is any) got it's suffix link reset to new internal node created in next extension of same phase. */ Node * lastNewNode = NULL ; Node * activeNode = NULL ; /*activeEdge is represented as an input string character index (not the character itself)*/ int activeEdge = -1 ; int activeLength = 0 ; // remainingSuffixCount tells how many suffixes yet to // be added in tree int remainingSuffixCount = 0 ; int leafEnd = -1 ; int * rootEnd = NULL ; int * splitEnd = NULL ; int size = -1 ; // Length of input string Node * newNode (int start , int * end) { Node * node = (Node *) malloc (sizeof (Node)); int i ; for (i = 0 ; i < MAX_CHAR ; i ++) node -> children [i] = NULL ; /*For root node, suffixLink will be set to NULL For internal nodes, suffixLink will be set to root by default in current extension and may change in next extension*/ node -> suffixLink = root ; node -> start = start ; node -> end = end ; /*suffixIndex will be set to -1 by default and actual suffix index will be set later for leaves at the end of all phases*/ node -> suffixIndex = -1 ; return node ; } int edgeLength (Node * n) { if (n == root) return 0 ; return * (n -> end) - (n -> start) + 1 ; } int walkDown (Node * currNode) { /*activePoint change for walk down (APCFWD) using Skip/Count Trick (Trick 1). If activeLength is greater than current edge length, set next internal node as activeNode and adjust activeEdge and activeLength accordingly to represent same activePoint*/ if (activeLength >= edgeLength (currNode)) { activeEdge += edgeLength (currNode); activeLength -= edgeLength (currNode); activeNode = currNode ; return 1 ; } return 0 ; } void extendSuffixTree (int pos) {

/*Extension Rule 1, this takes care of extending all leaves created so far in tree*/ leafEnd = pos ; /*Increment remainingSuffixCount indicating that a new suffix added to the list of suffixes yet to be added in tree*/ remainingSuffixCount ++ ; /*set lastNewNode to NULL while starting a new phase, indicating there is no internal node waiting for it's suffix link reset in current phase*/ lastNewNode = NULL ; // Add all suffixes (yet to be added) one by one in tree while (remainingSuffixCount > 0) { if (activeLength == 0) activeEdge = pos ; // APCFALZ // There is no outgoing edge starting with // activeEdge from activeNode if (activeNode -> children [text [activeEdge]] == NULL) { // Extension Rule 2 (A new leaf edge gets // created) activeNode -> children [text [activeEdge]] = newNode (pos , & leafEnd); /*A new leaf edge is created in above line starting from an existing node (the current activeNode), and if there is any internal node waiting for it's suffix link get reset, point the suffix link from that last internal node to current activeNode. Then set lastNewNode to NULL indicating no more node waiting for suffix link reset.*/ if (lastNewNode != NULL) { lastNewNode -> suffixLink = activeNode ; lastNewNode = NULL ; } } // There is an outgoing edge starting with // activeEdge from activeNode else { // Get the next node at the end of edge starting // with activeEdge Node * next = activeNode -> children [text [activeEdge]]; if (walkDown (next)) // Do walkdown { // Start from next node (the new activeNode) continue ; } /*Extension Rule 3 (current character being processed is already on the edge)*/ if (text [next -> start + activeLength] == text [pos]) { // If a newly created node waiting for it's // suffix link to be set, then set suffix // link of that waiting node to current // active node if (lastNewNode != NULL && activeNode != root) { lastNewNode -> suffixLink = activeNode ; lastNewNode = NULL ; } // APCFER3 activeLength ++ ; /*STOP all further processing in this phase and move on to next phase*/ break ; } /*We will be here when activePoint is in the middle of the edge being traversed and current character being processed is not on the edge (we fall off the tree). In this case, we add a new internal node and a new leaf edge going out of that new node. This is Extension Rule 2, where a new leaf edge and a new internal node get created*/ splitEnd = (int *) malloc (sizeof (int)); * splitEnd = next -> start + activeLength - 1 ; // New internal node Node * split = newNode (next -> start , splitEnd); activeNode -> children [text [activeEdge]] = split ; // New leaf coming out of new internal node split -> children [text [pos]] = newNode (pos , & leafEnd); next -> start += activeLength ; split -> children [text [next -> start]] = next ; /*We got a new internal node here. If there is any internal node created in last extensions of same phase which is still waiting for it's suffix link reset, do it now.*/ if (lastNewNode != NULL) { /*suffixLink of lastNewNode points to current newly created internal node*/ lastNewNode -> suffixLink = split ; } /*Make the current newly created internal node waiting for it's suffix link reset (which is pointing to root at present). If we come across any other internal node (existing or newly created) in next extension of same phase, when a new leaf edge gets added (i.e. when Extension Rule 2 applies is any of the next extension of same phase) at that point, suffixLink of this node will point to that internal node.*/ lastNewNode = split ; } /* One suffix got added in tree, decrement the count of suffixes yet to be added.*/ remainingSuffixCount -- ; if (activeNode == root && activeLength > 0) // APCFER2C1 { activeLength -- ; activeEdge = pos - remainingSuffixCount + 1 ; } else if (activeNode != root) // APCFER2C2 { activeNode = activeNode -> suffixLink ; } } } void print (int i , int j) { int k ; for (k = i ; k <= j ; k ++) printf ("%c" , text [k]); } // Print the suffix tree as well along with setting suffix // index So tree will be printed in DFS manner Each edge // along with it's suffix index will be printed void setSuffixIndexByDFS (Node * n , int labelHeight) { if (n == NULL) return ; if (n -> start != -1) // A non-root node { // Print the label on edge from parent to current // node Uncomment below line to print suffix tree // print(n->start, *(n->end)); } int leaf = 1 ; int i ; for (i = 0 ; i < MAX_CHAR ; i ++) { if (n -> children [i] != NULL) { // Uncomment below two lines to print suffix // index // if (leaf == 1 && n->start != -1) // printf(" [%d]\n", n->suffixIndex); // Current node is not a leaf as it has outgoing // edges from it. leaf = 0 ; setSuffixIndexByDFS (n -> children [i], labelHeight + edgeLength (n -> children [i])); } } if (leaf == 1) { n -> suffixIndex = size - labelHeight ; // Uncomment below line to print suffix index // printf(" [%d]\n", n->suffixIndex); } } void freeSuffixTreeByPostOrder (Node * n) { if (n == NULL) return ; int i ; for (i = 0 ; i < MAX_CHAR ; i ++) { if (n -> children [i] != NULL) { freeSuffixTreeByPostOrder (n -> children [i]); } } if (n -> suffixIndex == -1) free (n -> end); free (n); } /*Build the suffix tree and print the edge labels along with suffixIndex. suffixIndex for leaf edges will be >= 0 and for non-leaf edges will be -1*/ void buildSuffixTree () { size = strlen (text); int i ; rootEnd = (int *) malloc (sizeof (int)); * rootEnd = -1 ; /*Root is a special node with start and end indices as -1, as it has no parent from where an edge comes to root*/ root = newNode (-1 , rootEnd); activeNode = root ; // First activeNode will be root for (i = 0 ; i < size ; i ++) extendSuffixTree (i); int labelHeight = 0 ; setSuffixIndexByDFS (root , labelHeight); } int traverseEdge (char * str , int idx , int start , int end) { int k = 0 ; // Traverse the edge with character by character // matching for (k = start ; k <= end && str [idx] != '\0' ; k ++ , idx ++) { if (

text [k] != str [idx]) return -1 ; // mo match } if (str [idx] == '\0') return 1 ; // match return 0 ; // more characters yet to match } int doTraversal (Node * n , char * str , int idx) { if (n == NULL) { return -1 ; // no match } int res = -1 ; // If node n is not root node, then traverse edge // from node n's parent to node n. if (n -> start != -1) { res = traverseEdge (str , idx , n -> start , * (n -> end)); if (res != 0) return res ; // match (res = 1) or no match (res // = -1) } // Get the character index to search idx = idx + edgeLength (n); // If there is an edge from node n going out // with current character str[idx], traverse that edge if (n -> children [str [idx]] != NULL) return doTraversal (n -> children [str [idx]], str , idx); else return -1 ; // no match } void checkForSubString (char * str) { int res = doTraversal (root , str , 0); if (res == 1) printf ("Pattern <%s> is a Substring \n " , str); else printf ("Pattern <%s> is NOT a Substring \n " , str); } // driver program to test above functions int main (int argc , char * argv []) { strcpy (text , "THIS IS A TEST TEXT$"); buildSuffixTree (); checkForSubString ("TEST"); checkForSubString ("A"); checkForSubString (" "); checkForSubString ("IS A"); checkForSubString (" IS A "); checkForSubString ("TEST1"); checkForSubString ("THIS IS GOOD"); checkForSubString ("TES"); checkForSubString ("TESA"); checkForSubString ("ISB"); // Free the dynamically allocated memory freeSuffixTreeByPostOrder (root); return 0 ; } C // A C program for substring check using Ukkonen's Suffix // Tree Construction #include <stdio.h> #include <stdlib.h> #include <string.h> #define MAX_CHAR 256 struct SuffixTreeNode { struct SuffixTreeNode * children [MAX_CHAR]; // pointer to other node via suffix link struct SuffixTreeNode * suffixLink ; /*(start, end) interval specifies the edge, by which the node is connected to its parent node. Each edge will connect two nodes, one parent and one child, and (start, end) interval of a given edge will be stored in the child node. Let's say there are two nods A and B connected by an edge with indices (5, 8) then this indices (5, 8) will be stored in node B. */ int start ; int * end ; /*for leaf nodes, it stores the index of suffix for the path from root to leaf*/ int suffixIndex ; }; typedef struct SuffixTreeNode Node ; char text [100]; // Input string Node * root = NULL ; // Pointer to root node /*lastNewNode will point to newly created internal node, waiting for it's suffix link to be set, which might get a new suffix link (other than root) in next extension of same phase. lastNewNode will be set to NULL when last newly created internal node (if there is any) got it's suffix link reset to new internal node created in next extension of same phase. */ Node * lastNewNode = NULL ; Node * activeNode = NULL ; /*activeEdge is represented as an input string character index (not the character itself)*/ int activeEdge = -1 ; int activeLength = 0 ; // remainingSuffixCount tells how many suffixes yet to // be added in tree int remainingSuffixCount = 0 ; int leafEnd = -1 ; int * rootEnd = NULL ; int * splitEnd = NULL ; int size = -1 ; // Length of input string Node * newNode (int start , int * end) { Node * node = (Node *) malloc (sizeof (Node)); int i ; for (i = 0 ; i < MAX_CHAR ; i ++) node -> children [i] = NULL ; /*For root node, suffixLink will be set to NULL For internal nodes, suffixLink will be set to root by default in current extension and may change in next extension*/ node -> suffixLink = root ; node -> start = start ; node -> end = end ; /*suffixIndex will be set to -1 by default and actual suffix index will be set later for leaves at the end of all phases*/ node -> suffixIndex = -1 ; return node ; } int edgeLength (Node * n) { if (n == root) return 0 ; return * (n -> end) - (n -> start) + 1 ; } int walkDown (Node * currNode) { /*activePoint change for walk down (APCFWD) using Skip/Count Trick (Trick 1). If activeLength is greater than current edge length, set next internal node as activeNode and adjust activeEdge and activeLength accordingly to represent same activePoint*/ if (activeLength >= edgeLength (currNode)) { activeEdge += edgeLength (currNode); activeLength -= edgeLength (currNode); activeNode = currNode ; return 1 ; } return 0 ; } void extendSuffixTree (int pos) { /*Extension Rule 1, this takes care of extending all leaves created so far in tree*/ leafEnd = pos ; /*Increment remainingSuffixCount indicating that a new suffix added to the list of suffixes yet to be added in tree*/ remainingSuffixCount ++ ; /*set lastNewNode to NULL while starting a new phase, indicating there is no internal node waiting for it's suffix link reset in current phase*/ lastNewNode = NULL ; // Add all suffixes (yet to be added) one by one in tree while (remainingSuffixCount > 0) { if (activeLength == 0) activeEdge = pos ; // APCFALZ // There is no outgoing edge starting with // activeEdge from activeNode if (activeNode -> children [text [activeEdge]] == NULL) { // Extension Rule 2 (A new leaf edge gets // created) activeNode -> children [text [activeEdge]] = newNode (pos , & leafEnd); /*A new leaf edge is created in above line starting from an existing node (the current activeNode), and if there is any internal node waiting for it's suffix link get reset, point the suffix link from that last internal node to current activeNode. Then set lastNewNode to NULL indicating no more node waiting for suffix link reset.*/ if (lastNewNode != NULL) { lastNewNode -> suffixLink = activeNode ; lastNewNode = NULL ; } } // There is an outgoing edge starting with // activeEdge from activeNode else { // Get the next node at the end of edge starting // with activeEdge Node * next = activeNode -> children [text [activeEdge]]; if (walkDown (next)) // Do walkdown { // Start from next node (the new

activeNode) continue ; } /*Extension Rule 3 (current character being processed is already on the edge)*/ if (text [next -> start + activeLength] == text [pos]) { // If a newly created node waiting for it's // suffix link to be set, then set suffix // link of that waiting node to current // active node if (lastNewNode != NULL && activeNode != root) { lastNewNode -> suffixLink = activeNode ; lastNewNode = NULL ; } // APCFER3 activeLength ++ ; /*STOP all further processing in this phase and move on to next phase*/ break ; } /*We will be here when activePoint is in the middle of the edge being traversed and current character being processed is not on the edge (we fall off the tree). In this case, we add a new internal node and a new leaf edge going out of that new node. This is Extension Rule 2, where a new leaf edge and a new internal node get created*/ splitEnd = (int *) malloc (sizeof (int)); * splitEnd = next -> start + activeLength - 1 ; // New internal node Node * split = newNode (next -> start , splitEnd); activeNode -> children [text [activeEdge]] = split ; // New leaf coming out of new internal node split -> children [text [pos]] = newNode (pos , & leafEnd); next -> start += activeLength ; split -> children [text [next -> start]] = next ; /*We got a new internal node here. If there is any internal node created in last extensions of same phase which is still waiting for it's suffix link reset, do it now.*/ if (lastNewNode != NULL) { /*suffixLink of lastNewNode points to current newly created internal node*/ lastNewNode -> suffixLink = split ; } /*Make the current newly created internal node waiting for it's suffix link reset (which is pointing to root at present). If we come across any other internal node (existing or newly created) in next extension of same phase, when a new leaf edge gets added (i.e. when Extension Rule 2 applies is any of the next extension of same phase) at that point, suffixLink of this node will point to that internal node.*/ lastNewNode = split ; } /* One suffix got added in tree, decrement the count of suffixes yet to be added.*/ remainingSuffixCount -- ; if (activeNode == root && activeLength > 0) // APCFER2C1 { activeLength -- ; activeEdge = pos - remainingSuffixCount + 1 ; } else if (activeNode != root) // APCFER2C2 { activeNode = activeNode -> suffixLink ; } } } void print (int i , int j) { int k ; for (k = i ; k <= j ; k ++) printf ("%c" , text [k]); } // Print the suffix tree as well along with setting suffix // index So tree will be printed in DFS manner Each edge // along with it's suffix index will be printed void setSuffixIndexByDFS (Node * n , int labelHeight) { if (n == NULL) return ; if (n -> start != -1) // A non-root node { // Print the label on edge from parent to current // node Uncomment below line to print suffix tree // print(n->start, *(n->end)); } int leaf = 1 ; int i ; for (i = 0 ; i < MAX_CHAR ; i ++) { if (n -> children [i] != NULL) { // Uncomment below two lines to print suffix // index // if (leaf == 1 && n->start != -1) // printf(" [%d]\n", n->suffixIndex); // Current node is not a leaf as it has outgoing // edges from it. leaf = 0 ; setSuffixIndexByDFS (n -> children [i], labelHeight + edgeLength (n -> children [i])); } } if (leaf == 1) { n -> suffixIndex = size - labelHeight ; // Uncomment below line to print suffix index // printf(" [%d]\n", n->suffixIndex); } } void freeSuffixTreeByPostOrder (Node * n) { if (n == NULL) return ; int i ; for (i = 0 ; i < MAX_CHAR ; i ++) { if (n -> children [i] != NULL) { freeSuffixTreeByPostOrder (n -> children [i]); } } if (n -> suffixIndex == -1) free (n -> end); free (n); } /*Build the suffix tree and print the edge labels along with suffixIndex. suffixIndex for leaf edges will be >= 0 and for non-leaf edges will be -1*/ void buildSuffixTree () { size = strlen (text); int i ; rootEnd = (int *) malloc (sizeof (int)); * rootEnd = -1 ; /*Root is a special node with start and end indices as -1, as it has no parent from where an edge comes to root*/ root = newNode (-1 , rootEnd); activeNode = root ; // First activeNode will be root for (i = 0 ; i < size ; i ++) extendSuffixTree (i); int labelHeight = 0 ; setSuffixIndexByDFS (root , labelHeight); } int traverseEdge (char * str , int idx , int start , int end) { int k = 0 ; // Traverse the edge with character by character // matching for (k = start ; k <= end && str [idx] != '\0' ; k ++ , idx ++) { if (text [k] != str [idx]) return -1 ; // mo match } if (str [idx] == '\0') return 1 ; // match return 0 ; // more characters yet to match } int doTraversal (Node * n , char * str , int idx) { if (n == NULL) { return -1 ; // no match } int res = -1 ; // If node n is not root node, then traverse edge // from node n's parent to node n. if (n -> start != -1) { res = traverseEdge (str , idx , n -> start , * (n -> end)); if (res != 0) return res ; // match (res = 1) or no match (res // = -1) } // Get the character index to search idx = idx + edgeLength (n); // If there is an edge from node n going out // with current character str[idx], traverse that edge if (n -> children [str [idx]] != NULL) return doTraversal (n -> children [str [idx]], str , idx); else return -1 ; // no match } void checkForSubString (char * str) { int res = doTraversal (root , str , 0); if (res == 1) printf ("Pattern <%s> is a Substring \n " , str); else printf ("Pattern <%s> is NOT a Substring \n " , str); } // driver program to test above functions int main (int argc , char * argv []) { strcpy (text , "THIS IS A TEST TEXT$"); buildSuffixTree (); checkForSubString ("TEST"); checkForSubString ("A"); checkForSubString (" "); checkForSubString ("IS A"); checkForSubString (" IS A "); checkForSubString ("TEST1"); checkForSubString ("THIS IS GOOD"); checkForSubString ("TES"); checkForSubString ("TESA"); checkForSubString ("ISB"); // Free the dynamically allocated memory freeSuffixTreeByPostOrder (root); return 0 ; } Java class SuffixTreeNode { SuffixTreeNode [] children ;

SuffixTreeNode suffixLink ; int start ; int [] end ; int suffixIndex ; public SuffixTreeNode () { this . children = new SuffixTreeNode [256] ; this . suffixLink = null ; this . start = 0 ; this . end = null ; this . suffixIndex = - 1 ; } } public class SuffixTree { static final int MAX_CHAR = 256 ; static String text = "" ; static SuffixTreeNode root = null ; static SuffixTreeNode lastNewNode = null ; static SuffixTreeNode activeNode = null ; static int activeEdge = - 1 ; static int activeLength = 0 ; static int remainingSuffixCount = 0 ; static int leafEnd = - 1 ; static int [] rootEnd = null ; static int [] splitEnd = null ; static int size = - 1 ; public static SuffixTreeNode newNode (int start , int [] end) { SuffixTreeNode node = new SuffixTreeNode (); node . start = start ; node . end = end ; node . suffixLink = root ; node . suffixIndex = - 1 ; return node ; } public static int edgeLength (SuffixTreeNode n) { return (n == root) ? 0 : n . end [0] - n . start + 1 ; } public static boolean walkDown (SuffixTreeNode currNode) { if (activeLength >= edgeLength (currNode)) { activeEdge += edgeLength (currNode); activeLength -= edgeLength (currNode); activeNode = currNode ; return true ; } return false ; } public static void extendSuffixTree (int pos) { leafEnd = pos ; remainingSuffixCount ++ ; lastNewNode = null ; while (remainingSuffixCount > 0) { if (activeLength == 0) { activeEdge = pos ; } if (activeNode . children [text . charAt (activeEdge)] == null) { activeNode . children [text . charAt (activeEdge)] = newNode (pos , new int [] { leafEnd }); if (lastNewNode != null) { lastNewNode . suffixLink = activeNode ; lastNewNode = null ; } } else { SuffixTreeNode next = activeNode . children [text . charAt (activeEdge)] ; if (walkDown (next)) { continue ; } int start = next . start ; int end = next . end [0] ; if (text . charAt (start + activeLength) == text . charAt (pos)) { if (lastNewNode != null && activeNode != root) { lastNewNode . suffixLink = activeNode ; lastNewNode = null ; } activeLength ++ ; break ; } splitEnd = new int [] { start + activeLength - 1 }; SuffixTreeNode split = newNode (next . start , splitEnd); activeNode . children [text . charAt (activeEdge)] = split ; split . children [text . charAt (pos)] = newNode (pos , new int [] { leafEnd }); next . start += activeLength ; split . children [text . charAt (next . start)] = next ; if (lastNewNode != null) { lastNewNode . suffixLink = split ; } lastNewNode = split ; } remainingSuffixCount -- ; if (activeNode == root && activeLength > 0) { activeLength -- ; activeEdge = pos - remainingSuffixCount + 1 ; } else if (activeNode != root) { activeNode = activeNode . suffixLink ; } } } public static void setSuffixIndexByDFS (SuffixTreeNode n , int labelHeight) { if (n == null) return ; if (n . start != - 1) { // Uncomment the following line to print suffix // tree print(n.start, n.end[0]); } int leaf = 1 ; for (int i = 0 ; i < MAX_CHAR ; i ++) { if (n . children [i] != null) { // Uncomment the following two lines to // print suffix index if (leaf == 1 && // n.start != -1) // System.out.print(" [" + n.suffixIndex // + "]"); leaf = 0 ; setSuffixIndexByDFS (n . children [i] , labelHeight + edgeLength (n . children [i])); } } if (leaf == 1) { n . suffixIndex = size - labelHeight ; // Uncomment the following line to print suffix // index System.out.print(" [" + n.suffixIndex + // "]"); } } public static void freeSuffixTreeByPostOrder (SuffixTreeNode n) { if (n == null) return ; for (int i = 0 ; i < MAX_CHAR ; i ++) { if (n . children [i] != null) { freeSuffixTreeByPostOrder (n . children [i]); } } if (n . suffixIndex == - 1) { n . end = null ; } } public static void buildSuffixTree () { size = text . length (); rootEnd = new int [] { - 1 }; root = newNode (- 1 , rootEnd); activeNode = root ; for (int i = 0 ; i < size ; i ++) { extendSuffixTree (i); } int labelHeight = 0 ; setSuffixIndexByDFS (root , labelHeight); } public static int traverseEdge (String str , int idx , int start , int end) { int k = 0 ; for (k = start ; k <= end && idx < str . length (); k ++ , idx ++) { if (text . charAt (k) != str . charAt (idx)) { return - 1 ; } } return (idx == str . length ()) ? 1 : 0 ; } public static int doTraversal (SuffixTreeNode n , String str , int idx) { if (n == null) { return - 1 ; } int res = - 1 ; if (n . start != - 1) { res = traverseEdge (str , idx , n . start , n . end [0]); if (res != 0) { return res ; } } idx += edgeLength (n); if (n . children [str . charAt (idx)] != null) { return doTraversal (n . children [str . charAt (idx)] , str , idx); } else { return - 1 ; } } public static void checkForSubstring (String str) { int res = doTraversal (root , str , 0); if (res == 1) { System . out . println ("Pattern <" + str + "> is a Substring"); } else { System . out . println ("Pattern <" + str + "> is NOT a Substring"); } } // Driver program to test above functions public static void main (String [] args) { text = "THIS IS A TEST TEXT$" ; buildSuffixTree (); checkForSubstring ("TEST"); checkForSubstring ("A"); checkForSubstring (" "); checkForSubstring ("IS A"); checkForSubstring (" IS A "); checkForSubstring ("TEST1"); checkForSubstring ("THIS IS GOOD"); checkForSubstring ("TES"); checkForSubstring ("TESA"); checkForSubstring ("ISB"); // Free the dynamically allocated memory freeSuffixTreeByPostOrder (root); } } Python class SuffixTreeNode : def __init__ (self , start , end , root): # Dictionary to hold children (edges out of this node) self . children = {} # Suffix link which points to another internal node (initially None) self . suffix_link = root # Start and end indices of the edge label from parent node to this node self . start = start self . end = end # Suffix index for leaf nodes (initially set to -1) self . suffix_index = - 1 def edge_length (self , current_pos): # Calculate edge length; use current_pos if it's a leaf (end is a reference to leafEnd) return self . end if isinstance (self . end , int) else current_pos - self . start + 1

class SuffixTree : def __init__ (self , text): self . text = text self . root = SuffixTreeNode (None , None , None) self . root . suffix_link = self . root self . last_new_node = None self . active_node = self . root self . active_edge = 0 self . active_length = 0 self . remaining_suffix_count = 0 self . leaf_end = - 1 self . size = len (text) self . build_suffix_tree () def build_suffix_tree (self): # Main function to build the suffix tree for the given text for i in range (self . size): self . extend_suffix_tree (i) def extend_suffix_tree (self , pos): # Rule 1 extension: Every suffix extension extends the leaf edges self . leaf_end = pos # Increment the count of suffixes we need to add self . remaining_suffix_count += 1 # Active node handling self . last_new_node = None while self . remaining_suffix_count > 0 : if self . active_length == 0 : self . active_edge = pos # APCFALZ next_char = self . text [self . active_edge] if self . active_edge < len (self . text) else None # Check if there is an outgoing edge starting with the active edge character if next_char not in self . active_node . children : # Rule 2 extension: Create a new leaf node self . active_node . children [next_char] = SuffixTreeNode (pos , self . leaf_end , self . root) if self . last_new_node is not None : self . last_new_node . suffix_link = self . active_node self . last_new_node = None else : next_node = self . active_node . children [next_char] # Walk down the tree if active length is longer than current edge length if self . active_length >= next_node . edge_length (pos): self . active_edge += next_node . edge_length (pos) self . active_length -= next_node . edge_length (pos) self . active_node = next_node continue # Rule 3 extension: Current character is already on the edge if self . text [next_node . start + self . active_length] == self . text [pos]: if self . last_new_node is not None and self . active_node != self . root : self . last_new_node . suffix_link = self . active_node self . active_length += 1 break # Rule 2 extension again: Creating a new internal node split_end = next_node . start + self . active_length - 1 split = SuffixTreeNode (next_node . start , split_end , self . root) self . active_node . children [next_char] = split split . children [self . text [pos]] = SuffixTreeNode (pos , self . leaf_end , self . root) next_node . start += self . active_length split . children [self . text [next_node . start]] = next_node if self . last_new_node is not None : self . last_new_node . suffix_link = split self . last_new_node = split # One suffix less to add self . remaining_suffix_count -= 1 if self . active_node == self . root and self . active_length > 0 : self . active_length -= 1 self . active_edge = pos - self . remaining_suffix_count + 1 elif self . active_node != self . root : self . active_node = self . active_node . suffix_link def _do_traversal (self , node , string , idx): if node . start is not None : # Traverse edge character by character edge_len = node . edge_length (len (self . text) - 1) if self . text [node . start : node . start + edge_len] != string [idx : idx + edge_len]: return False idx += edge_len if idx == len (string): return True next_char = string [idx] if next_char in node . children : return self . _do_traversal (node . children [next_char], string , idx) return False def check_substring (self , string): # Function to check if the string is a substring if self . _do_traversal (self . root , string , 0): print (f "Pattern < { string } > is a Substring") else : print (f "Pattern < { string } > is NOT a Substring") # Driver code def main (): st = SuffixTree ("THIS IS A TEST TEXT$") queries = ["TEST" , "A" , " " , "IS A" , " IS A " , "TEST1" , "THIS IS GOOD" , "TES" , "TESA" , "ISB"] for query in queries : st . check_substring (query) if __name__ == "__main__" : main () JavaScript class SuffixTreeNode { constructor () { this . children = new Array (256). fill (null); this . suffixLink = null ; this . start = 0 ; this . end = null ; this . suffixIndex = - 1 ; } } const MAX_CHAR = 256 ; let text = "" ; let root = null ; let lastNewNode = null ; let activeNode = null ; let activeEdge = - 1 ; let activeLength = 0 ; let remainingSuffixCount = 0 ; let leafEnd = - 1 ; let rootEnd = null ; let splitEnd = null ; let size = - 1 ; function newNode (start , end) { const node = new SuffixTreeNode (); node . start = start ; node . end = end ; node . suffixLink = root ; node . suffixIndex = - 1 ; return node ; } function edgeLength (n) { return (n === root) ? 0 : n . end [0] - n . start + 1 ; } function walkDown (currNode) { if (activeLength >= edgeLength (currNode)) { activeEdge += edgeLength (currNode); activeLength -= edgeLength (currNode); activeNode = currNode ; return true ; } return false ; } function extendSuffixTree (pos) { leafEnd = pos ; remainingSuffixCount ++ ; lastNewNode = null ; while (remainingSuffixCount > 0) { if (activeLength === 0) { activeEdge = pos ; } if (! activeNode . children [text [activeEdge]. charCodeAt (0)]) { activeNode . children [text [activeEdge]. charCodeAt (0)] = newNode (pos , [leafEnd]); if (lastNewNode) { lastNewNode . suffixLink = activeNode ; lastNewNode = null ; } } else { const next = activeNode . children [text [activeEdge]. charCodeAt (0)]; if (walkDown (next)) { continue ; } const start = next . start ; const end = next . end [0]; if (text [start + activeLength] === text [pos]) { if (lastNewNode && activeNode !== root) { lastNewNode . suffixLink = activeNode ; lastNewNode = null ; } activeLength ++ ; break ; } splitEnd = [start + activeLength - 1]; const split = newNode (next . start , splitEnd); activeNode . children [text [activeEdge]. charCodeAt (0)] = split ; split . children [text [pos]. charCodeAt (0)] = newNode (pos , [leafEnd]); next . start += activeLength ; split . children [text [next . start]. charCodeAt (0)] = next ; if (lastNewNode) { lastNewNode . suffixLink = split ; } lastNewNode = split ; }

remainingSuffixCount -- ; if (activeNode === root && activeLength > 0) { activeLength -- ; activeEdge = pos - remainingSuffixCount + 1 ; } else if (activeNode !== root) { activeNode = activeNode . suffixLink ; } } } function setSuffixIndexByDFS (n , labelHeight) { if (! n) return ; if (n . start !== - 1) { // Uncomment the following line to print suffix tree // print(n.start, n.end[0]); } let leaf = 1 ; for (let i = 0 ; i < MAX_CHAR ; i ++) { if (n . children [i]) { // Uncomment the following two lines to print suffix index // if (leaf === 1 && n.start !== -1) // console.log(" [" + n.suffixIndex + "]"); leaf = 0 ; setSuffixIndexByDFS (n . children [i], labelHeight + edgeLength (n . children [i])); } } if (leaf === 1) { n . suffixIndex = size - labelHeight ; // Uncomment the following line to print suffix index // console.log(" [" + n.suffixIndex + "]"); } } function freeSuffixTreeByPostOrder (n) { if (! n) return ; for (let i = 0 ; i < MAX_CHAR ; i ++) { if (n . children [i]) { freeSuffixTreeByPostOrder (n . children [i]); } } if (n . suffixIndex === - 1) { n . end = null ; } } function buildSuffixTree () { size = text . length ; rootEnd = [- 1]; root = newNode (- 1 , rootEnd); activeNode = root ; for (let i = 0 ; i < size ; i ++) { extendSuffixTree (i); } let labelHeight = 0 ; setSuffixIndexByDFS (root , labelHeight); } function traverseEdge (str , idx , start , end) { let k = 0 ; for (k = start ; k <= end && str [idx] !== undefined ; k ++ , idx ++) { if (text [k] !== str [idx]) { return - 1 ; } } return (str [idx] === undefined) ? 1 : 0 ; } function doTraversal (n , str , idx) { if (! n) { return - 1 ; } let res = - 1 ; if (n . start !== - 1) { res = traverseEdge (str , idx , n . start , n . end [0]); if (res !== 0) { return res ; } } idx += edgeLength (n); if (n . children [text [idx]. charCodeAt (0)]) { return doTraversal (n . children [text [idx]. charCodeAt (0)], str , idx); } else { return - 1 ; } } function checkForSubString (str) { const res = doTraversal (root , str , 0); if (res === 1) { console . log ("Pattern <" + str + "> is a Substring"); } else { console . log ("Pattern <" + str + "> is NOT a Substring"); } } // Driver program to test above functions function main () { text = "THIS IS A TEST TEXT$" ; buildSuffixTree (); checkForSubString ("TEST"); checkForSubString ("A"); checkForSubString (" "); checkForSubString ("IS A"); checkForSubString (" IS A "); checkForSubString ("TEST1"); checkForSubString ("THIS IS GOOD"); checkForSubString ("TES"); checkForSubString ("TESA"); checkForSubString ("ISB"); // Free the dynamically allocated memory freeSuffixTreeByPostOrder (root); } main (); Output: Pattern <TEST> is a Substring Pattern <A> is a Substring Pattern < > is a Substring Pattern <IS A> is a Substring Pattern < IS A > is a Substring Pattern <TEST1> is NOT a Substring Pattern <THIS IS GOOD> is NOT a Substring Pattern <TES> is a Substring Pattern <TESA> is NOT a Substring Pattern <ISB> is NOT a Substring Ukkonen’s Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that, traversal for substring check takes O(M) for a pattern of length M. With a slight modification in the traversal algorithm discussed here, we can answer the following: Find all occurrences of a given pattern P present in text T. How to check if a pattern is prefix of a text? How to check if a pattern is suffix of a text? We have published following more articles on suffix tree applications: Suffix Tree Application 2 – Searching All Patterns Suffix Tree Application 3 – Longest Repeated Substring Suffix Tree Application 4 – Build Linear Time Suffix Array Generalized Suffix Tree 1 Suffix Tree Application 5 – Longest Common Substring Suffix Tree Application 6 – Longest Palindromic Substring Comment Article Tags: Article Tags: Pattern Searching Advanced Data Structure DSA Suffix-Tree