Progetto Algoritmi e laboratorio

SuffixTree

Rosario Scuderi

1. Caratteristiche

1.1 Introduzione

Per suffix-tree (o albero dei suffissi) si intende una struttura dati, che consente di trovare tutti i suffissi di una stringa, che si presenta come una valida soluzione, in alcuni casi alternativa ad altre già esistenti, a diversi problemi di tipo computazionale: ad esempio la ricerca di una sottostringa in tempo **O(n);** In particolare, in un contesto in cui **n** rappresenta la stringa su cui effettuare la ricerca e **m** la stringa da trovare, la costruzione dell'albero ha complessità **O(n)** mentre la ricerca delle occorrenze **O(m)**.

1.2 Storia

Nel corso degli anni sono stati ideate diverse versioni ed implementazioni del suffix-tree. La prima versione fu ideata da Weiner nel 1973; Successivamente McCreight progettò un'implementazione più efficiente dal punto di vista dello spazio utilizzato che però, così come la versione di Weiner risultava essere particolarmente complessa e non sempre garantiva la complessità lineare. Infine, tra il 1992 e il 1995, Ukkonen creò una versione in grado di mantenere la complessità lineare ed i vantaggi dell'algoritmo di McCreight.
Al giorno d'oggi, l'algoritmo di Ukkonen risulta essere il migliore per quanto riguarda l'implementazione di un Suffix-Tree.

1.3 Definizione formale e proprietà

Un albero dei suffissi A per una stringa S è una struttura dati tale che:

- È un albero ordinato, direzionato, e radicato, con esattamente n foglie (numerate da 0 a n− 1).
- Ogni nodo interno ha almeno due figli e ogni arco è etichettato con una sottostringa non vuota di S.
- Le etichette su due archi uscenti da un nodo devono avere caratteri iniziali diversi.
- Per ogni foglia i, la concatenazione delle etichette sugli archi del cammino dalla radice alla foglia i corrisponde al suffisso S(i, n − 1) di S.

ATTENZIONE

La definizione formale del suffix-tree non è in grado di garantire l'esistenza di un albero per ogni stringa.

Se un suffisso di S è uguale al prefisso di un altro suffisso di S, tale suffisso non potrà terminare in una foglia.

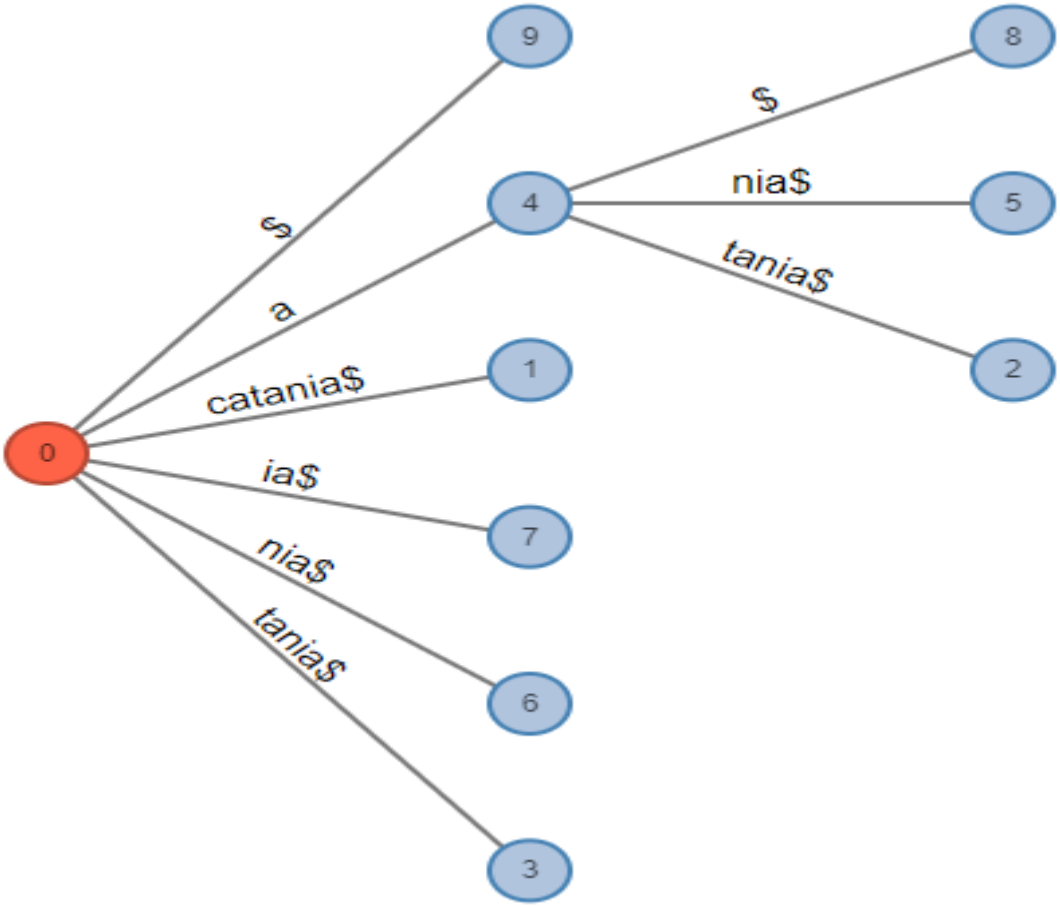
Per risolvere il problema basta aggiungere un carattere di blank alla fine della stringa S.

Alcune proprietà dell'albero dei suffissi sono:

- L'etichetta di un percorso dalla radice a un nodo è la concatenazione nell'ordine delle sottostringhe che etichettano gli archi che costituiscono il percorso
- L'etichetta di percorso di un nodo è l'etichetta del percorso dalla radice a quel nodo
- La profondità di un nodo v è il numero di caratteri nell'etichetta di v
- Un percorso che termina nel mezzo di un arco (u,v) divide l'etichetta di (u,v) in un punto prestabilito. L'etichetta di questo percorso è l'etichetta di u concatenata con i caratteri sull'arco (u,v) fino al punto di divisione.

1.4 Esempio stilizzato albero dei suffissi

S= Catania *dove '$' è carattere di blank*



2. Algoritmo di Ukkonen

2.1 Caratteristiche

L’algoritmo di Ukkonen elabora la stringa S da sinistra verso destra e in maniera continua costruisce un albero che ad ogni iterazione è il suffix-tree per la stringa prefisso di S elaborata in quel momento.

Durante l’esecuzione dell’algoritmo alcuni dei collegamenti costruiti fino a quel momento, in genere la struttura dell’albero stesso, tende a cambiare, ad esempio: alcuni sottoalberi vengono scomposti in più ramificazioni o se ne creano di nuovi.

Inoltre, possono verificarsi i seguenti casi:

1. **Nell'albero corrente il cammino etichettato S [j, i] termina in una foglia numerata *j*.** In questo caso il nuovo carattere S [i+1] viene aggiunto all'etichetta dell'ultimo arco di tale cammino (quello che termina nella foglia).

2. **Nell'albero corrente il cammino etichettato S [j, i] termina in un nodo *u* interno (implicito o esplicito) ma nessun cammino che parte da *u* inizia con il carattere S[i+1].**
   In questo caso viene creata una nuova foglia etichettata *j* connessa al nodo *u* con un arco etichettato con il carattere S[i+1]. Nel caso in cui *u* sia un nodo implicito, prima di fare ciò, il nodo *u* viene sostituito con un nodo esplicito che suddivide in due parti l'arco a cui esso appartiene.

3. **Nell'albero corrente il cammino etichettato S [j, i] termina in un nodo *u* interno (implicito o esplicito) e almeno un cammino che parte da *u* inizia con il carattere S [i+1].** In questo caso il suffisso S [j, i+1] è già presente nell'albero e non occorre fare niente.

2.2 Pseudocodice

```
Construct tree T1
For i from 1 to m-1 do
begin {phase i+1}
        For j from 1 to i+1
                begin {extension j}
                Find the end of the path from the root labelled S[j..i] in the current tree.
                Extend that path by adding character S[i+l] if it is not there already
        end;
end;
```

3. Algoritmo Naïve

Per l'implementazione in codice si è deciso, in questa sede, di implementare una versione naïve dell'algoritmo per la costruzione di un SuffxTree.

3.1 Caratteristiche

Nonostante, dal punto di vista computazionale, un'implementazione di tipo naïve non sia efficiente, infatti presenta una complessità di **$O(n^2)$,** presenta il vantaggio di essere più semplice da descrivere rispetto alle altre opzioni.

Il funzionamento è abbastanza intuitivo:

Per prima cosa, viene inserito, all'interno della struttura dati, l'intera stringa input S[0…S.len]+$;

A questo punto, ad ogni iterazione da i=1 a S.len+$, verrà inserito nell'albero la sottostringa S[i to S.len]+$.

Durante l'inserimento delle stringhe può verificarsi che un nodo abbia due o più archi figli che hanno stringhe con prefissi in comune.

In questo caso, in fase di costruzione dell'albero, quando un suffisso A viene inserito ed incontra un altro suffisso B che comincia con gli stessi caratteri, l'arco contenete il suffisso B dovrà essere diviso a partire dalla lettera successiva ultimo carattere in comune, creando due nuovi sottoalberi:

Il primo conterrà il suffisso S[B.substr (LCP(A,B).LEN)];

Il secondo conterrà il suffisso S[A.substr (LCP(A,B).LEN)];

Nel caso in cui questo caso non si verificasse, basterà semplicemente aggiungere il suffisso all'ultimo sottoalbero Ni visitato

#Si noti la procedura LCP (longhest common prefix) che permette di trovare il prefisso di lunghezza massima tra due stringhe.

Entrando nel dettaglio, questa procedura prende in input due stringhe e ne restituisce una terza che corrisponde al prefisso più lungo in comune.

Poiché l'algoritmo LCP rappresenta un punto cruciale nella costruzione del SuffixTree, è doveroso scriverne pseudocodice e complessità

```
LCP(str1,str2){                                          O(N)
        result;
len1 = str1.len;
        for(from i=0 to len1-1){

                if(str1[i]=/=str2[i])
                        break;
                result << (str1[i])

        }
        Return result;
}
```

3.2 Funzione Build-SuffixTree

Questa procedura rappresenta le fondamenta della costruzione del SuffixTree in cui confluiscono tutti metodi necessari.

Ad ogni iterazione, dopo aver eseguito il caso base (inserire l'intera stringa di input nell'albero), a partire dalla sottostringa di S che è S[from i=1 to a S.LEN] , l'algoritmo andrà a ricercare all'interno dell'albero, partendo dalla radice, l'arco contenente il possibile suffisso da dividere;

A questo punto potranno verificarsi due casi:

Caso 1: Non è stato trovato alcun arco, all'interno dell'intero albero, che contiene il suffisso.

In questo caso il suffisso verrà collegato direttamente alla radice.

Caso 2 : E' stato trovato un suffisso che può essere diviso

In questo caso viene chiamata una funzione ausiliaria che divide l'arco.

PSEUDOCODICE

```
void build(input){
    input+=’$’;
    add input to root;   //caso base.
    string tmp;
    for(form i=1 to input.length){
        tmp=input.substr(i);
            if(find(tmp)==NOT FOUND){
                add tmp to root;
            }
            else{
                split(arco trovato, tmp);
            }
}
```

Assumendo che l’insieme dei caratteri presenti nella stringa sia finito questo algoritmo avrà complessità di $O(n^2)$.

Dimostrazione:

Durante la i-esima iterazione, l’algoritmo, trova il prefisso comune più lungo tra S[i to input.Len], il quale corrisponde a tmp all’interno dello pseudocodice, e i precedenti i suffissi; Sia X=S[i to k]$ il prefisso corrente.

Quindi si avranno k+1 confronti per identificare X.

A questo punto il resto del suffisso, S[k+1 to input.Len] deve essere letto e assegnato al nuovo arco.

Pertanto, la quantità totale di lavoro durante l’iterazione i è $\Theta(n)$ (dove n è la lunghezza della stringa in input)

Quindi $\Theta(n)$ lavoro eseguito per input.Len volte è uguale a $O(n^2)$.

Si noti che all'interno dello pseudocodice precedente vengono richiamate due diverse funzioni: "**find**" e "**split**", rispettivamente, il primo serve a trovare l'arco che potrebbe contenere il prefisso di lunghezza massima, il secondo lo divide.

Find

La funzione find, dopo aver preso in input un nodo di partenza X e una stringa S , inizia a cercare, a partire da X (che durante la prima iterazione corrisponde alla root), una stringa all'interno dell'albero che abbia il più lungo prefisso in comune con S.

- Qualora non venga trovato sarà restituito il nodo root.
- Nel caso in cui venga trovato si potranno verificare due casi
  - Caso 1: Il nodo trovato non ha figli e quindi verrà restituito in output.
  - Caso 2: Il nodo ha figli; in questo caso verrà richiamata la funzione in maniera ricorsiva passando in input il nuovo nodo e S a partire da LCP.LEN.

Questa procedura funziona in maniera simile ad una visita pre-order; di conseguenza la complessità è uguale a O(n) nel caso migliore.

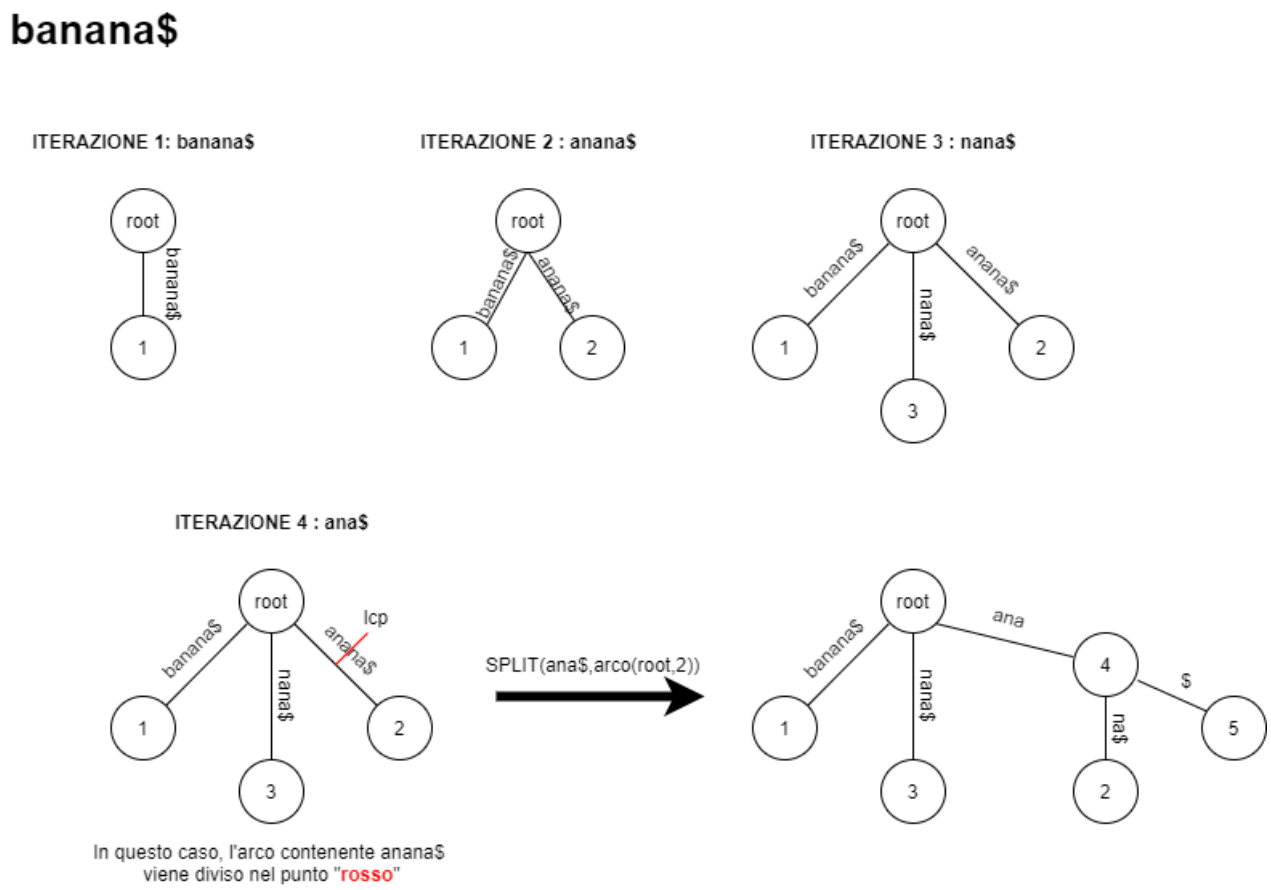
Split

La funzione split prende in input una stringa S e l'arco da dividere.

Il suo scopo è quello di creare due nuovi archi: il primo conterrà la stringa A[LCP.LEN to a.LEN] dove a è la stringa già presente nell'arco; il secondo conterrà la stringa B[LCP.LEN to S.len].

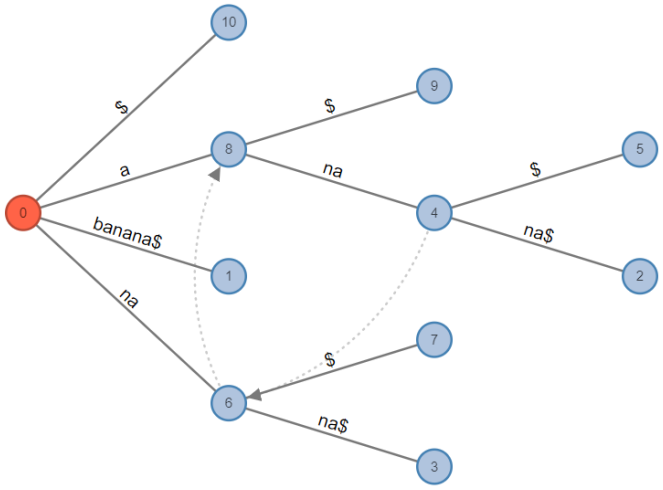
La complessità di questo algoritmo, poiché si effettuano solo operazioni di assegnamento, è costante O(1) .

3.3 Esempio



L'algoritmo procederà in maniera analoga, per tutte le iterazioni successive.

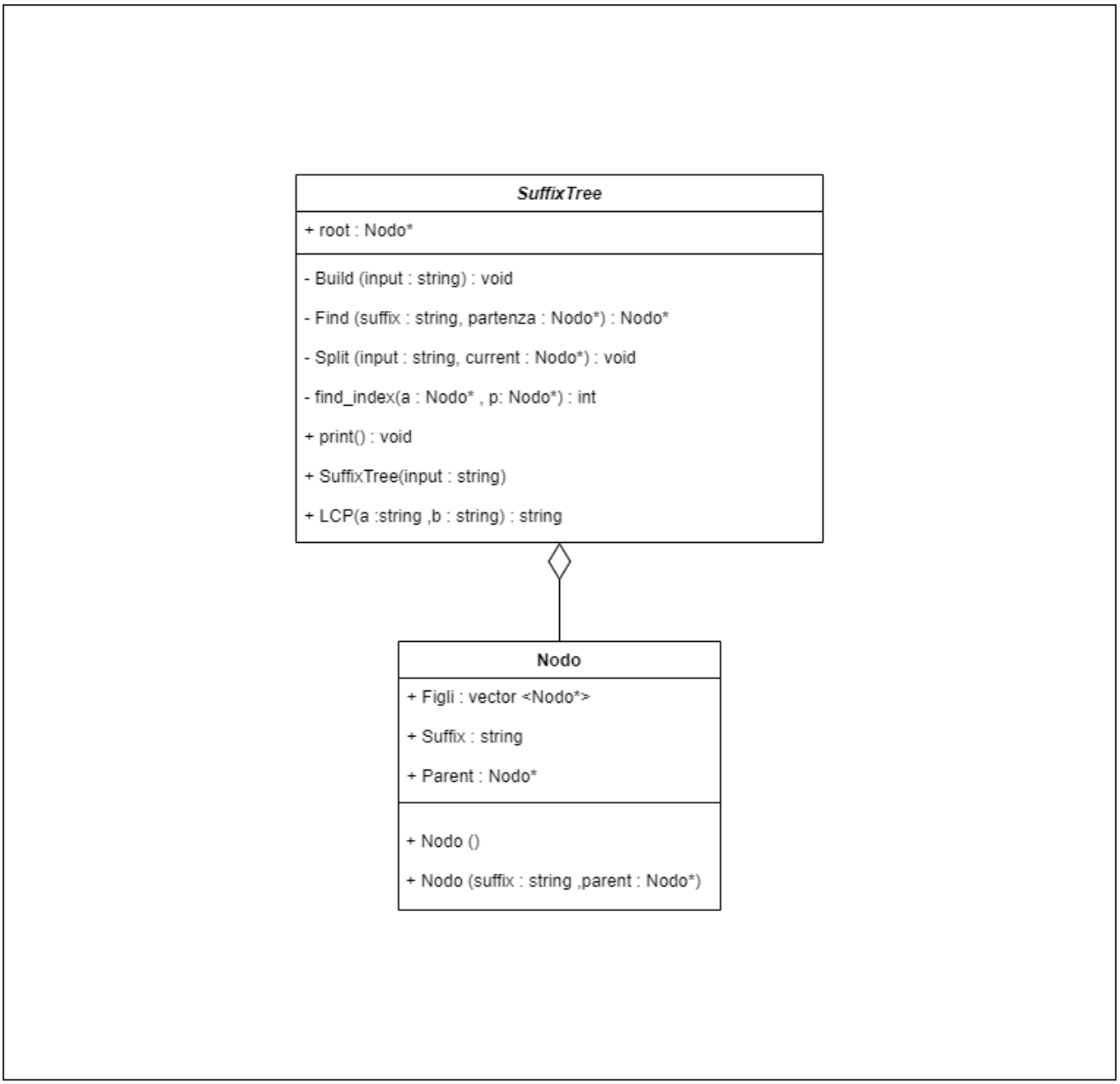
Il risultato finale sarà il seguente:



4. Implementazione

Come già detto precedentemente, per l'implementazione in codice si è pensato di adottare l'algoritmo Naïve; la scelta è motivata dalla semplicità concettuale del suo funzionamento.

4.1 UML



In fase di progettazione dell'algoritmo e del codice in C++, si è deciso di effettuare un'operazione di "aggregazione" in cui Nodi, Archi e Suffissi sono stati riuniti in un' unica entità nominata Nodo; Questa operazione è stata effettuata per semplificare il codice e ridurne le dimensioni; Di conseguenza, nel momento in cui bisognerà effettuare la scissione dell'arco, anziché passare in input un arco, verrà inserito un nodo contenente il suffisso che, dunque, verrà suddiviso in altri due nodi.

4.2 Classe Nodo

L'obbiettivo di questa classe è quello di raccogliere al suo interno tutte le informazioni che riguardano nodi, archi e suffissi e ha l'obbiettivo di rappresentare le foglie, dunque anche i vari sotto-alberi, del Suffix-Tree.

La classe "Nodo" presenta al suo interno tre attributi e due costruttori.

Gli attributi sono:

- Una struttura dati di tipo "vector" che contiene i puntatori ai nodi figli.
- Una stringa denominata "suffix" che contiene il suffisso del sottoalbero corrente.
- Un puntatore al padre del nodo corrente.

Per quanto riguarda i costruttori: il primo serve solo a creare il nodo radice dell'albero e non prende in input alcun parametro, mentre il secondo permette la creazione di tutti gli altri nodi; Quest'ultimo prende in input due parametri: una stringa, che setta l'attributo suffix del nodo, ed un puntatore a Nodo che, invece, definisce il nodo padre.

La classe Nodo si presenta nella forma seguente:

```
class Nodo{
    public:
        vector<Nodo*> figli;
        string suffix;
        Nodo *parent;

        Nodo(){}

        Nodo(string suffix,Nodo *parent){
            this->suffix=suffix;
            this->parent=parent;
        }
};
```

4.3 Classe SuffixTree

La classe SuffixTree rappresenta la parte più importane del codice: infatti, al suo interno sono contenuti tuti i metodi necessari alla corretta costruzione dell'albero. La struttura presenta un solo attributo, descritto da un puntatore al nodo radice dell'albero, e sette metodi.

I metodi implementati sono elencati di seguito:

- Costruttore
- Build
- Find
- Split
- LCP
- Find_Index
- Print

Costruttore

Il costruttore accetta in input la stringa "input" fornita all'interno del main, crea la radice dell'albero e chiama la procedura **build()** che, dopo aver ricevuto "input" dal costruttore, costruisce l'intero albero servendosi di altre procedure ausiliarie.

```
SuffixTree(string input){
    root=new Nodo();
    build(input);
}
```

Build

Il metodo build() svolge un ruolo essenziale all'interno del codice e rappresenta il motore della creazione del SuffixTree.

Il suo compito è quello di gestire le chiamate ai metodi ausiliari attraverso vari controlli e scorrere la stringa ad ogni iterazione fino a S[S.len-1] dove S è la stringa in input.

In particolare, al suo interno, si invocano i metodi **find** e **split**: il primo cerca all'interno dell'albero un prefisso in comune con la stringa S[i to len-1] fornitagli da build(), il secondo 'divide' il nodo ritornato dal metodo find().

All'avvio, il metodo build(), crea il primo nodo contenente l'intera stringa S e lo collega alla radice dell'albero; successivamente avvia un ciclo for che, ad ogni iterazione, inserisce all'interno di una variabile temporanea T la stringa S[i to len-1]; a questo punto la funzione find() cerca, all'interno dell'albero, partendo dal nodo radice, un nodo che contiene il suffisso con il più lungo prefisso in comune con T e, in base al nodo restituito da find() , l'algoritmo deciderà se creare un nuovo nodo da collegare alla radice o dividerne uno già esistente.

```
void build(string input){
    Nodo *first=new Nodo(input,root);
    root->figli.push_back(first);
    string tmp;
    for(int i=1;i<input.length();i++){
        tmp=input.substr(i);
        Nodo *nodo=find(tmp,root);
        if(nodo==root){
            Nodo *nuovo=new Nodo (tmp,nodo);
            root->figli.push_back(nuovo);
        }
        else{
            split(tmp,nodo);
        }
    }
}
```

Find

La funzione **find()** si occupa di cercare , all'interno dell'albero, un nodo che contiene il suffisso con il più lungo prefisso in comune con la stringa in input.

L'algoritmo, dopo aver ricevuto come parametri la stringa suffix e un puntatore a Nodo P che rappresenta il nodo di partenza, inizia a cercare il nodo interessato tra i figli del nodo di partenza; a questo punto si possono verificare due casi:

1. Non è stato trovato nessun nodo:
   In questo caso, l'algoritmo restituisce P.

2. L'algoritmo ha trovato un nodo X che contiene un suffisso che ha n caratteri inziali in comune con la stringa suffix;
   In questo caso l'algoritmo, dopo determinati controlli, può intraprendere due diverse strade:

   1. Se il nodo trovato X ha almeno un figlio, si invoca la funzione find() ricorsivamente passando in input X e la stringa suffix[lcp(suffix,partenza->figli[i].suffix).len to S.len-1].

   2. Se il nodo X trovato non ha figli, l'algoritmo restituisce X e termina.

```
Nodo *find(string suffix,Nodo* partenza){
    int max,dim;        max=dim=0;
    Nodo *daRitornare=partenza; string sdim;
    if(partenza->figli.size()>0){
        for(int i=0;i<partenza->figli.size();i++){
            sdim=LCP(suffix,partenza->figli[i]->suffix);
            dim=sdim.length();
            if(dim>max){
                max=dim;
                daRitornare=partenza->figli[i];
            }
        }
    }
    if(max==0){
        return daRitornare;
    }
    else{
        if(daRitornare->figli.size()>0){
            return find(suffix.substr(dim),daRitornare);
        }
        else{
            return daRitornare;
        }
    }
    return daRitornare;
}
```

Si noti che durante l'esecuzione del ciclo for si effettua il controllo if(dim>max) che serve a confrontare i suffissi dei vari nodi figli visitati; nel caso di esito positivo, viene memorizzato il nodo contenente il prefisso comune più grande al parametro suffix all'interno della variabile "daRitornare".

In particolare: **max** contiene la dimensione dell'ultimo prefisso più lungo, mentre **dim** contiene la dimensione del suffisso del nodo corrente.

Split

La funzione **split()** ha l'obbiettivo di 'dividere' il nodo che riceve in input.

Il metodo si presenta come una lunga successione di assegnazioni a variabili, intervallate da alcune istruzioni di controllo e accetta, come parametri, una stringa "input" e un puntatore a Nodo "current".

L'algoritmo in questione, in realtà, non divide nessun nodo ma si serve di varie istruzioni per creare un effetto simile. L'idea è quella di creare un nuovo nodo X che sostituirà current all'interno del vettore dei figli del nodo padre di current, quest'ultimo, invece, diventerà figlio di X che, di conseguenza, verrà inserito all'interno del vettore dei figli di X insieme ad un altro nodo nuovo, che conterrà la sottostringa input[lcp(input,current.suffix).len to input.len-1], e il suo suffisso verrà sostituito con la sottostringa current.suffix[lcp(input,current.suffix).len to current.suffix.len-1].

```
void split(string input,Nodo *current){

    string nev=" ";
    if(current->parent->suffix.length()==1){
        input=input.substr(1);
        current=find(input,current->parent);
        //cout<<"Nodo attuale (current post reFind) == "<<
    }
    if(current->suffix.length()==1){
        string suff=input.substr(1);
        Nodo *nuovo=new Nodo(suff,current);
        current->figli.push_back(nuovo);
        return;
    }
    //cout<<"Nodo attuale (current) == "<<current->suffix<<
    nev=LCP(input,current->suffix);
    int len=nev.length();
    Nodo * nuovo=new Nodo(nev,current->parent);
    int index=find_index(current,current->parent);
    current->parent->figli[index]=nuovo;
    nuovo->figli.push_back(current);
    current->parent=nuovo;
    string suffix1=current->suffix.substr(len);
    current->suffix=suffix1;
    string suffix2=input.substr(len);
    //cout<<"nev "<<nev<<" "<<"suffix1 "<<suffix1<<" "<<"su
    Nodo *right=new Nodo(suffix2,nuovo);
    nuovo->figli.push_back(right);
    return;

}
```

*All'interno del codice sono presenti due controlli **if**; questi sono stati implementati per risolvere alcuni bug che andavano ad assegnare suffissi vuoti a nodi contenenti suffissi di lunghezza uguale a 1.

Inoltre, risolvono errori di "Segmentation fault"

**Per trovare l'indice di current all'interno del vettore dei figli del suo nodo padre si utilizza la funzione find_index che verrà approfondita inseguito.

LCP

La funzione **LCP()** restituisce il prefisso comune più lungo tra due stringhe.

```
static string LCP(string str1, string str2)
{
    string result="";
    int n1 = str1.length(), n2 = str2.length();

    // Compare str1 and str2
    for (int i=0, j=0; i<=n1-1 && j<=n2-1; i++,j++)
    {
        if (str1[i] != str2[j])
            break;
        result.push_back(str1[i]);
    }

    return result;
}
```

Find_Index

La funzione **find_index()** restituisce la posizione, sotto forma di indice, del nodo a all'interno del vettore dei figli di p.

```
int find_index(Nodo *a,Nodo *p){
    for(int i=0;i<p->figli.size();i++){
        if(p->figli[i]==a){
            return i;
        }
    }
    return 0;
}
```

Print

La funzione **print()** stampa a schermo, ricorsivamente, i nodi dell'albero.

```
void print(Nodo *x) {
    if(x->figli.size()==0){return;}
    for(int i=0;i<x->figli.size();i++){
        cout<<x->figli[i]->suffix<<"{ ";
        print(x->figli[i]);
        cout<<"} ";
    }
}
```

*All'interno del codice è possibile trovare un'implementazione iterativa della procedura di stampa che visualizza a schermo un numero limitato di nodi.

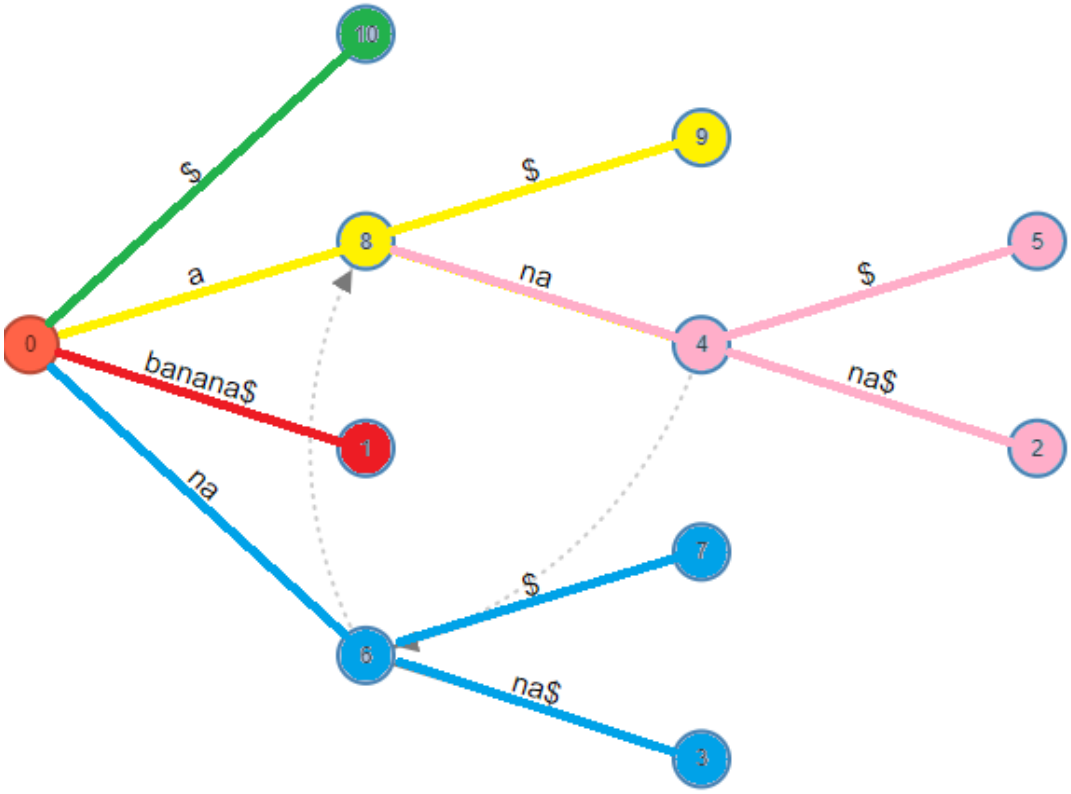
4.4 Esempi

I seguenti esempi sono stati compilati tramite terminale Windows e Linux (Ubuntu).

BANANA

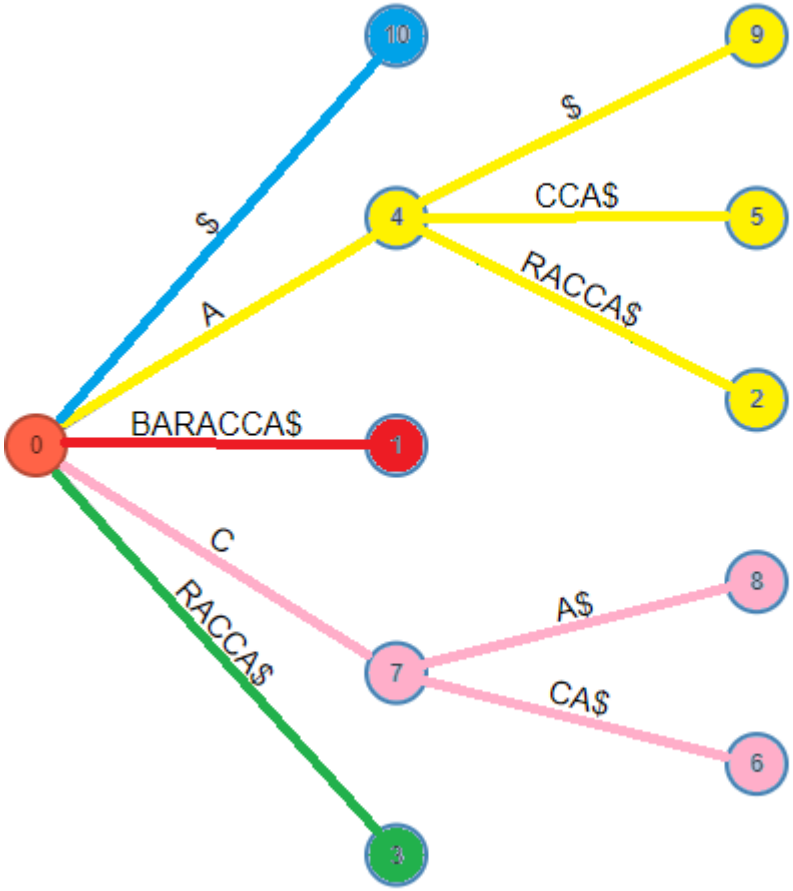
```
PROBLEMS    OUTPUT    TERMINAL    DEBUG CONSOLE

PS C:\Users\rosar\Desktop> g++ project.cpp
PS C:\Users\rosar\Desktop> ./a.exe
BANANA${ } A{ NA{ NA${ } ${ } } ${ } } NA{ NA${ } ${ } } ${ }
PS C:\Users\rosar\Desktop>
```



BARACCA (caso con più caratteri uguali consecutivi)

```
PS C:\Users\rosar\Desktop> g++ project.cpp
PS C:\Users\rosar\Desktop> ./a.exe
BARACCA${ } A{ RACCA${ } CCA${ } ${ } } RACCA${ } C{ CA${ } A${ } } ${ }
PS C:\Users\rosar\Desktop>
```



MISSISSIPPI (caso con più caratteri uguali consecutivi e con una stringa di input lunga)

```
PS C:\Users\rosar\Desktop> g++ project.cpp
PS C:\Users\rosar\Desktop> ./a.exe
MISSISSIPPI${ } I[ SSI{ SSIPPI${ } PPI${ } ] PPI${ } ${ } ) S{ SI{ SSIPPI${ } PPI${ } } I{ SSIPPI${ } PPI${ } } ] P{ PI${ } I${ } } ${ }
PS C:\Users\rosar\Desktop>
```