Sommario

1. TRACCIA SCELTA:

1

Costruire un vocabolario V utilizzando un Hash Table con il metodo della concatenazione, che abbia le seguenti funzioni:
- Inserimento del termine
- Cancellazione
- Ricerca del termine. In caso di fallimento deve restituire una lista delle parole più prossime, utilizzando un approccio basato sulla Distanza di editing.

La distanza di editing (o di Levenshtein) misura il numero di operazioni (inserimento, cancellazione e correzione) che devono essere eseguite sulla tastiera per trasformare una stringa in un' altra. La distanza di Levenshtein tra due stringhe di caratteri S = (S1; : : : ; SN) e T = (T1; : : : ; TM) è definita nel modo seguente. Siano Si e Tj i caratteri corrispondenti rispettivamente all' i-esimo carattere di S ed al j-esimo carattere di T allora la loro distanza di editing d[i; j] _e data da:

$$d[i,j] = min \begin{cases} d[i-1,j]+1 \\ d[i,j-1]+1 \\ d[i-1,j-1]+(S_i \neq T_j) \end{cases} \quad (1)$$

$$d[0,0] = 0 \quad (2)$$

2

Costruire un vocabolario V_Permutato, utilizzando un albero RED BLACK che abbia le stesse funzioni, dell'esercizio precedente. Nell' implementazione di entrambi i quesiti si faccia uso delle Classi Astratte.

Lo scopo del programma è quello di costruire un dizionario, usando una tabella hash per la prima traccia e gli alberi Red Black per la seconda. Una parola può comparire una sola volta. Ogni richiesta di inserimento(Insert) o cancellazione(Delete) di una parola richiede una ricerca per verificare la presenza della parola nel dizionario.

2. ANALISI DEL PROBLEMA

La traccia chiede di gestire un vocabolario *V* utilizzando un ***Hash Table*** che gestisca le collisioni col metodo della *concatenazione* , e di gestire un vocabolario *V_permutato* utilizzando una struttura dati basata su un ***RB-TREE*** .

Inoltre entrambi i quesiti devono essere implementati utilizzando una Classe Astratta :
In questo caso la Classe Astratta è stata chiamata *AbsVocabolario*
In questa verrà implementato il metodo della "Distanza di Levenshtein" , che in caso la ricercà abbia un esito negativo , fornisce una lista di paroli più prossime a quella cercata.
Le classi che erediteranno i metodi della Classe Astratta "*AbsVocabolario* " sono :
- *VocabolarioHash* che servirà a gestire il primo quesito della traccia .
- *VocabolarioRbTree* che servirà a gestire il secondo quesito della traccia .

3. DISTANZA DI LEVENSHTEIN

In questo capitolo verrà illustrato come funziona la distanza di Levenshtein definita anche distanza di Editing .
Esempi di applicazione nell'informatica sono ad esempio la funzione "forse ceravi?" di GOOGLE , oppure la correzione automatica negli editor di testo tipo WORD.
Entrambe si basano sull'algoritmo di Levenshtein, questo costituisce il numero minimo di mosse necessarie per trasformare la stringa s1 nella stringa s2; le mosse consentite sono:

- la cancellazione di un carattere
- la sostituzione di un carattere
- l'aggiunta di un carattere

In poche parole, la distanza di Levenshtein, o distanza di edit, è una misura per la differenza fra due stringhe. Introdotta dallo scienziato russo Vladimir Levenshtein nel 1965, serve a determinare quanto due stringhe siano simili. Per esempio, per trasformare "bar" in "biro" occorrono due modifiche:
"bar" -> "bir" (sostituzione di 'a' con 'i')
"bir" -> "biro" (inserimento di 'o')
Non è possibile trasformare la prima parola nella seconda con meno di due modifiche, quindi la distanza di Levenshtein fra "bar" e "biro" è 2.

Si utilizza la tecnica della Programmazione Dinamica :
• Riduzione del problema in sottoproblemi
• Risoluzione di tutti i sottoproblemi possibili
• Risoluzione del problema originale tramite l'utilizzo delle soluzioni dei suoi sottoproblemi

I casi base, per i quali la distanza di Levenshtein (D) è calcolabile immediatamente, sono:
D(0,0) = 0
D(i,0) = i (i cancellazioni)
D(0,j) = j (j cancellazioni)

Si considerano due stringhe: $x=\{x_1,x_2,\dots,x_n\}$ e $y=\{y_1,y_2,\dots,y_m\}$, costruiamo il vettore D(i,j)= distanza tra i prefissi $x_1,x_2,\dots,x_n$ e $y_1,y_2,\dots,y_m$, il risultato cercato sarà: D(n.m) = distanza tra $x_1,x_2,\dots,x_n$ e $y_1,y_2,\dots,y_m$.
Si hanno tre possibilità per calcolare D(i,j), noto D(k,l) per $k < i$ e $l < j$:

- il carattere a_i va sostituito con il carattere b_j e quindi:
  D(i,j) = D(i - 1,j - 1) + t(ai,bj)

- il carattere a_i va cancellato e quindi:
  D(i,j) = D(i - 1,j) + 1

- il carattere b_j va inserito e quindi:
  D(i,j) = D(i,j - 1) + 1

Dato che si vuole ottenere il valore minimo, otteniamo la ricorrenza :

$$d[i,j] = min \begin{cases} d[i-1,j]+1 \\ d[i,j-1]+1 \\ d[i-1,j-1]+(S_i \neq T_j) \end{cases} \quad (1)$$

$$d[0,0] = 0 \quad (2)$$

Considerate due stringhe *sorgente* e *destinazione,* la distanza di Levenshtein è il numero di operazioni elementari necessarie a trasformare la striga *sorgente* in quella *destinazione*. Due stringhe uguali hanno distanza di Levenshtein pari a 0.

Un algoritmo usato comunemente per calcolare la distanza di Levenshtein richiede l'uso di una matrice di $(n + 1) \times (m + 1)$, dove n e m rappresentano le lunghezze delle due stringhe.

Esempio :

calcolo della distanza di Levenshtein per S1="winter" (n=6) e S2="writers" (m=7)

		w	r	i	t	e	r	s
	0	1	2	3	4	5	6	7
w	1	**0**	1	2	3	4	5	6
i	2	1	**1**	1	2	3	4	5
n	3	2	2	**2**	2	3	4	5
t	4	3	3	3	**2**	3	4	5
e	5	4	4	4	3	**2**	3	4
r	6	5	4	5	4	3	**2**	**3**

Nella cella D(6,7) è memorizzata la distanza di Levenshtein
tra le due stringhe.

La distanza di Levenshtein ha alcuni semplici limiti superiori ed inferiori:
- è almeno la differenza fra le lunghezze delle due stringhe;
- è 0 se e solo se le due stringhe sono identiche;
- se le lunghezze delle due stringhe sono uguali, la distanza di Levenshtein la distanza di Hamming[1], cioè pari alla lunghezza delle stringhe;
- il limite superiore è pari alla lunghezza della stringa più lunga.

[1] la **distanza di Hamming** tra due stringhe di ugual lunghezza è il numero di posizioni nelle quali i simboli corrispondenti sono diversi. In altri termini, la distanza di Hamming misura il numero di *sostituzioni* necessarie per convertire una stringa nell'altra

4. LA CLASSE ASTRATTA

Una classe base, definita con funzioni virtuali, serve a stabilire cosa possono fare gli oggetti delle classi derivate. In altre parole la classe base fornisce, oltre alle funzioni, anche uno "schema di comportamento" per le classi derivate. Portando all'estremo questa pratica, possiamo creare una classe base con funzioni virtuali senza codice, dette **funzioni virtuali pure**. Non avendo codice, queste funzioni servono solo da "schema di comportamento" per le classi derivate.

Una classe base con almeno una funzione virtuale pura è detta **classe astratta**, perché definisce la struttura di una gerarchia di classi, ma non può essere istanziata direttamente. Possiamo dire che una classe astratta è una classe da cui non si possono creare oggetti ed ha solo una funzione di "scheletro" per definire sottoclassi.

Per **dichiarare una funzione come virtuale pura**, basta aggiungere "= 0" alla dichiarazione.

A differenza dalle normali funzioni virtuali, le **funzioni virtuali pure** devono essere ridefinite (implementate) tutte nelle classi derivate (anche con "corpo nullo", quando non servono). Se una classe derivata non implementa anche una sola funzione virtuale pura della classe base, rimane una classe astratta e non può ancora essere istanziata.

Le classi astratte sono di importanza fondamentale nella programmazione in C++. Esse presentano delle **interfacce** , senza il vincolo degli aspetti implementativi, che sono invece forniti dalle loro classi derivate.

Una gerarchia di classi, che deriva da una o più classi astratte, può essere costruita in modo "incrementale", ciò permette il "raffinamento" di un progetto, aggiungendo via via nuove classi senza la necessità di modificare le parti preesistenti.

Le classi astratte sono usate per dichiarare componenti comuni per le classi che ne derivano. Quando si eredita da una classe astratta in una classe, dopo aver definito un suo membro, si può usarlo in tutte le funzioni definite nelle classi figlie.

5. VOCABOLARIO GESTITO CON HASH TABLE

Questo capitolo sintetizza come è stata affrontata la risoluzione del primo quesito , attraverso l'uso di una classe : *VocabolarioHash* , ereditata dalla Classe Astratta AbsVocabolario
Da quest'ultima erediterà metodi vituali da implementare quali :
Insert , Delete , Search , SimilWrd ,Print

e metodi già implementati come :
DistEdit e GetMin (questi ultimi implementano la distanza di Levenshtein.

Inoltre avrà il metodo privato hashu che servirà a creare le apposite key , per la Tabella Hash.

5.1 HASH TABLE

Una hash table è una struttura dati utilizzata per mettere in corrispondenza una data chiave con un dato valore.
L'hash table è molto utilizzata nei metodi di ricerca nominati Hashing.
L'hashing è un'estensione della ricerca indicizzata da chiavi che gestisce problemi di ricerca nei quali le chiavi di ricerca non presentano queste proprietà. Una ricerca basata su hashing è completamente diversa da una basata su confronti: invece di muoversi nella struttura data in funzione dell'esito dei confronti tra chiavi, si cerca di accedere agli elementi nella tabella in modo diretto tramite operazioni aritmetiche che trasformano le chiavi in indirizzi della tabella. Esistono vari tipi di algoritmi di hashing. Per quanto affermato, in una tabella di hashing ben dimensionata il costo medio di ricerca di ogni elemento è indipendente dal numero di elementi.

Una funzione hash `e una funzione che data una chiave $k \in U$ restituisce la posizione della tabella in cui l'elemento con chiave k viene memorizzato :

$h : U \rightarrow [0, 1, \ldots m - 1]$

N.B: la dimensione m della tabella può non coincidere con la $|U|$, anzi in generale $m < |U|$

L'idea `e quella di definire una funzione d'accesso che permetta di ottenere la posizione di un elemento in data la sua chiave . Con l'hashing, un elemento con chiave k viene memorizzato nella cella *h(k)*

Pro:

- riduciamo lo spazio necessario per memorizzare la tabella

Contro:

- perdiamo la corrispondenza tra chiavi e posizioni in tabella
- le tabelle hash possono soffrire del fenomeno delle **collisioni**

Collisioni
Due chiavi k1 e k2 collidono quando corrispondono alla stessa posizione della tabella, ossia quando *h(k1) = h(k2)*

Soluzione ideale: eliminare del tutto le collisioni scegliendo un'opportuna (= perfetta) funzione hash.
Una funzione hash si dice perfetta se `e iniettiva, cioè se per ogni k1, k2 ∈ U
$k1 \neq k2 \Rightarrow h(k1) \neq h(k2)$
Deve essere $|U| \leq m$
Se $|U| > m$, evitare del tutto le collisioni `e impossibile (Ad es. nel caso di dover inserire un nuovo elemento in una tabella piena).

Una possibile alternativa: utilizzare una buona funzione hash (per minimizzare le collisioni) e prevedere nel contempo dei metodi di risoluzione delle collisioni.
Una buona funzione hash `e una funzione che distribuisce le chiavi in modo uniforme sulle posizione della tabella e quindi minimizza le collisioni quando possibile.

Una buona funzione hash deve:

1. essere facile da calcolare (costo costante)
2. soddisfare il requisito di uniformità semplice: ogni chiave deve avere la stessa probabilità di vedersi assegnata una qualsiasi posizione ammissibile, indipendentemente da altri valori hash già assegnati

Sia P(k) la probabilità che sia estratta una chiave k tale che h(k) = j, allora :

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \text{ per } j = 0, \ldots, m-1$$

Il requisito di uniformità semplice è difficile da verificare perchè raramente è nota la funzione di distribuzione di probabilità con cui vengono estratte le chiavi (la funzione Pr) Nella pratica però è possibile usare delle euristiche (metodo di approssimazione) per realizzare delle funzioni hash con buone prestazioni

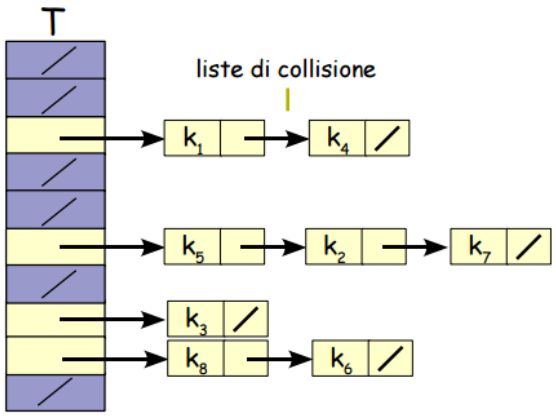
La funzione hash scelta nel progetto è quella universale :

```
int hashu(char *v, int M)
{
int h, a=31415, b=27183;
for(h=0;*v != '\0'; v++, a=a*b%(M - 1))
        h= (a*h + *v) %M;
return (h<0) ? (h+M) : h;
}
```

Un metodo più efficace è quello di utilizzare valori casuali per i coefficienti e un valore casuale diverso per ogni cifra. Una funzione di hash universale ideale fa sì che la probabilità di collisione fra chiavi distinte in una tabella di dimensione M sia precisamente 1/M.

Per quanto riguarda invece le **collisioni** esse sono state gestite con il metodo della concatenazione :
Gli elementi collidenti vengono inseriti nella stessa posizione della tabella in una lista concatenata

Concatenamento (chaining)



- Gli elementi con lo stesso valore hash h vengono memorizzati in una lista concatenata
- Si memorizza un puntatore alla testa della lista nello slot A[h] della tabella hash

Operazioni:

- Insert: inserimento in testa
- Search, Delete: richiedono di scandire la lista alla ricerca della chiave

Concatenamento

Costo computazionale

Notazione
– n: numero di elementi nella tabella
– m: numero di slot nella tabella
Fattore di carico
– α: numero medio di elementi nelle liste (α = n/m)
Caso pessimo: tutte le chiavi sono in una unica lista
– Insert: Θ(1)
– Search, Delete: Θ(n)
Caso medio: dipende da come le chiavi vengono distribuite
– Assumiamo hashing uniforme , da cui ogni slot della tabella avrà mediamente α chiavi

Complessità

Teorema1 :
– In tabella hash con concatenamento, una ricerca senza successo richiede un tempo atteso di Θ(1 + α)

Dimostrazione:
– Una chiave non presente nella tabella può essere collocata in uno qualsiasi degli m slot
– Una ricerca senza successo tocca tutte le chiavi nella lista corrispondente
– Tempo di hashing: 1 + lunghezza attesa lista: α → Θ(1+α)

Teorema 2:

– In una tabella hash con concatenamento, una ricerca con successo richiede un tempo atteso di Θ(1 + α)
– Più precisamente: Θ(2 + α/2 + α/2n)
(dove n è il numero di elementi)

Dimostrazione:
– se n = O(m), α = O(1)
– quindi tutte le operazioni sono Θ(1)

6. VOCABOLARIO GESTITO CON ALBERI RED&BLACK

Per il secondo quesito , nel progetto ho creato la classe : *VocabolarioRbTree* , ereditata dalla Classe Astratta AbsVocabolario , dalla quale eredita gli stessi metodi della classe precedente (*VocabolarioHash).*
Inoltre avrà tutta una serie di altri metodi atti a gestire il funzionamento della struttura dati RbTree.

6.1 ALBERI RED & BLACK

Un RB-albero (Red-black tree) è un albero binario di ricerca dove ogni nodo ha in aggiunta un campo per memorizzare il suo colore: RED (rosso) o BLACK (nero).
La colorazione avviene mediante regole precise, che assicurano che nessun cammino dalla radice ad una foglia risulti lungo più del doppio di qualsiasi altro. Si ottiene così che l'albero è abbastanza bilanciato.
Ciascun nodo dell'albero contiene i campi color, key, left, right, e parent.

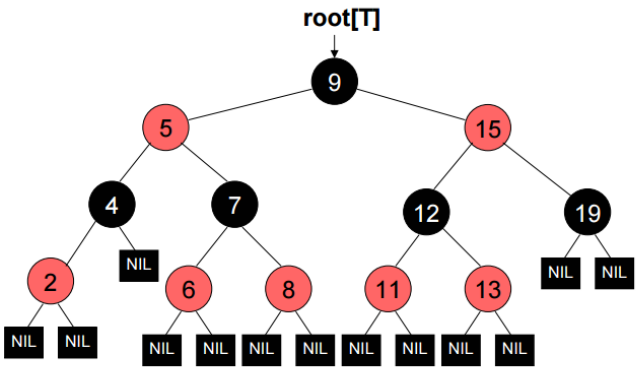
Proprietà

Un RB-albero (red-black tree) soddisfa le seguenti proprietà:

1. Ciascun nodo è rosso o nero.
2. Ciascuna foglia (NIL) è nera.
3. Se un nodo è rosso allora entrambi i suoi figli sono neri.
4. Ogni cammino da un nodo ad una foglia sua discendente contiene lo stesso numero di nodi neri.

Dalla proprietà 4 si definisce la b-altezza di un nodo x.
bh(x) = numero di nodi neri su un cammino da un nodo x, non incluso, ad una foglia sua discendente.

Nota: Un nodo nero può avere figli rossi o neri. Tutti i nodi interni hanno due figli.



ALTEZZA NERA
L'altezza nera di un nodo x è indicata con bh(x) e viene definita come il numero di nodi neri lungo un percorso che inizia dal nodo x(escluso) e finisce in una foglia.

Teoremi

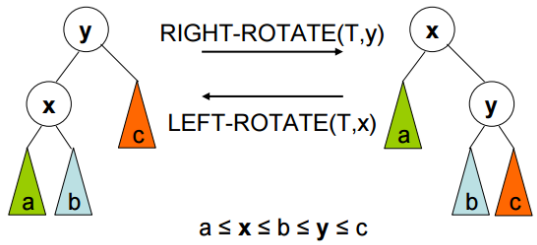
1. Ogni albero RB con radice v, ha almeno 2bh(v)-1 nodi interni, dove bh(x) è l'altezza nera del nodo x.
2. In un albero RB, almeno la metà dei nodi dalla radice ad una foglia deve essere nero.
3. In un albero RB, nessun percorso da un nodo v ad una foglia è lungo più del doppio del percorso da v a un'altra foglia.
4. Un albero RB con n nodi interni ha altezza massima pari a 2log(n+1).

Operazioni su un RB-albero

Le operazioni di SEARCH, PREDECESSOR, MINIMUM e MAXIMUM sono quelle viste per un albero binario di ricerca. Possono essere eseguite in un RB-albero con un tempo pari a O(h) = O(lg(n)).
Le operazioni di INSERT e DELETE si complicano, poiché devono tener conto delle proprietà aggiuntive del RB-albero.
Più precisamente si devono effettuare dei cambiamenti in modo da ripristinare le regole di colorazione dei nodi.
Tuttavia, RB-INSERT() e RB-DELETE() possono essere eseguite in tempo O(lg(n))

Rotazioni

Le rotazioni sono delle operazioni che cambiano la struttura dei puntatori di due nodi (padre e figlio).
Sono operazioni locali dell'albero di ricerca che non modificano l'ordinamento delle chiavi.
Queste operazioni sono utilizzate da INSERT e DELETE per ripristinare le proprietà violate degli RB-alberi mantenendo l'albero sempre un albero binario di ricerca.



I due nodi interni x e y potrebbero essere ovunque nell'albero. Le lettere a, b e c rappresentano sottoalberi arbitrari.
Un'operazione di rotazione mantiene l'ordinamento delle chiavi secondo la visita inorder.

Inserimento

L'inserimento potrebbe modificare la struttura dell'albero andando a violare le proprietà precedentemente indicate degli alberi RB.
In particolare può violare due Proprietà:

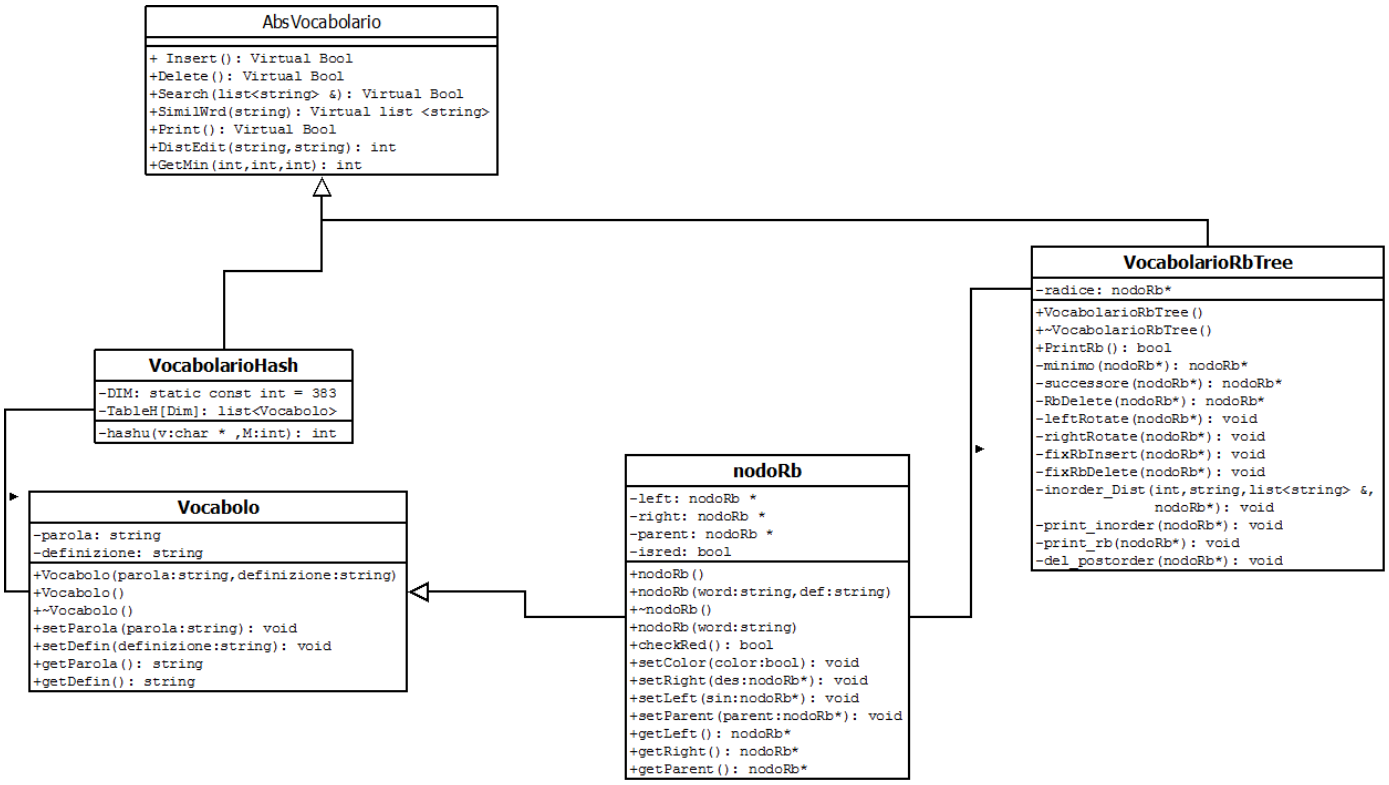
- Proprietà 2 : Può essere violata se il nodo (inserito) rosso è la radice, che invece deve essere di colore nero
- Proprietà 4 : Il nodo rosso deve avere entrambi figli neri.

Cancellazione

L'algoritmo di cancellazione dei RB è costruito sull'algoritmo di cancellazione per alberi binari di ricerca. Le operazioni di ripristino del bilanciamento sono necessarie solo quando il nodo cancellato è nero. Questo perché se il nodo cancellato fosse rosso, non cambierebbe l'altezza bh(x)
Si potrebbero creare nodi rossi consecutivi e la radice dunque resterebbe nera.
In particolare può violare due proprietà:
- Radice Nera
- I figli di uno nodo rosso devono essere neri

7. CLASSI E CODICE

Di Seguito verranno illustrate le varie classi e il codice relativo di ognuna di esse , per le quali è composto il progetto.
Come prima cosa sarà mostrato un diagramma Uml delle classi , e poi saranno descritte una per una :



7.1 AbsVocabolario

La classe AbsVocabolario è una classe Astratta in quanto ha dei metodi virtuali

Ha i seguenti metodi :

int DistEdit(string , string) : il quale restituisce la distanza di editing tra due stringhe

int GetMin (int , int , int) : il quale resituisce il minimo fra tre valori

I metodi virtuali da implementare sono :
virtual bool Insert()
virtual bool Delete ()
virtual bool Search (list<string> &)
virtual list<string> SimilWrd (string)
virtual bool Print()

Codice :

"AbsVocabolario.h"

```
class AbsVocabolario
{
        public:

                virtual bool Insert()= 0;
                virtual bool Delete () = 0;
                virtual bool Search ( list<string> & )=0;
                virtual list<string> SimilWrd (string )=0;
                virtual bool Print()=0;
            int DistEdit(string , string);
            int GetMin (int , int , int );

};
```

"AbsVocabolario.cpp"

```
#include "absvocabolario.h" // class's header file

// class constructor

int AbsVocabolario::GetMin (int a, int b, int c)
{
   int min =a;
   if (b < min)
      min=b;
   if (c < min)
      min=c;
   return min;
}
```

```
int AbsVocabolario::DistEdit(string word1 , string word2)
{
   int n = word1.length();
   int m = word2.length();
   int i,j;
   char w_1 , w_2;
   int sizem = m+1;
   int sizen = n+1;
   int costo;

         // se la stringa sorgente è vuota
   if (n == 0) // la distanza è il num di char della dest.
   {
      return m;
   }
      // se la stringa destinazione è vuota
   if (m == 0) // la distanza è il num di chr della sorg.
   {
      return n;
   }
   // creo la matrice (n+1)*(m+1)
   int d[sizen][sizem];

      // la prima riga della mat. conterrà le distanze da 0 a n
      // la distanza 1 è associata al primo chr della stringa, 0 al vuoto
      //  esempio:
      // 0 1 2 3 4 5 6 7 8
      // - f a g g i a n o
   for (i = 0; i <= n; i++)
   {
         d[i][0] = i;
   }
    // la prima colonna della mat. conterrà le distanze da 0 a m
```

```
    // la distanza 1 è associato al primo chr della stringa, 0 al vuoto
    // Nel caso del nostro esempio:
    // - 0
    // f 1
    // a 2
    // b 3
    // b 4
    // r 5
    // i 6
    // a 7
    // n 8
    // o 9
  for (j = 0; j <= m; j++)
  {
        d[0][j] = j;
  }

  for (i=1;i <= n;i++)
  {
     w_1 = word1[i-1];
     for (j = 1 ;j <= m ; j++)
     {
        w_2 = word2[j-1];
        if (w_1==w_2)
           costo=0;
        else
           costo=1;
     // imposto la cella d[i][j] scegliendo il valore minimo tra:
     //   - la cella immediatamente superiore + 1
     //   - la cella immediatamente a sinistra + 1
     //   - la cella diagonalmente in alto a sinistra più il costo
        d[i][j] = GetMin(d[i - 1][j] + 1, d[i][j - 1] + 1, d[i - 1][j - 1] + costo);
     }
```

```
    }
    return d[n][m];
}
```

7.2 VocabolarioHash

La classe VocabolarioHash è una classe che eredita alcuni metodi virtuali da implementare , dalla classe AbsVocabolario.

Come Attributi ha :

static const int DIM : che è un variabile statica che serve definire la dimensione massima della Tabell Hash

list<Vocabolo> TableH[DIM] : che è una lista gestita dalla stl (Standard Template List) di un array di oggetti costituiti dalla classe Vocabolo , e che forma la TabellaHash

Ha i seguenti metodi pubblici :
I primi sono ereditati dalla classe AbsVocabolario e implementati secondo le caratteristiche delle tabelle Hash

Ha i seguenti metodi privati :

int hashu (char *v , int M) : che ritorna una chiave hash universale

Codice :

```
"VocabolarioHash.h"

class VocabolarioHash : public AbsVocabolario
{
        public:
                // class constructor
                VocabolarioHash(){};
                // class destructor
```

```
                ~VocabolarioHash();

        bool Insert();
                bool Delete ();
                bool Search (list<string> &SameW);
                list<string> SimilWrd (string word);
                bool Print();

  private:
    static const int DIM = 383;
    list<Vocabolo> TableH[DIM];
    int hashu (char *v , int M);
};

#include "vocabolariohash.h"

// class destructor
VocabolarioHash::~VocabolarioHash()
{
}

bool VocabolarioHash::Insert()
{
  string word;
  cout << endl << endl << endl;
  cout << "\tVocabolo da inserire : ";
  cout<<endl<<endl<<"\t";
  getline(cin,word);
  cout<<"\t_____"<<endl;
  int k = hashu((char*)word.c_str(), DIM); // calcola l'hash della parola da inserire

  for(auto it = TableH[k].begin(); it!= TableH[k].end(); it++)
    if ( word==it->getParola())
    {
      cout << endl <<  "\tVocabolo gia presente";
      cout<<endl<<"\t_____"<<endl;
      return false;
```

```
    }
  string defin;
  cout << endl << "\tInserisci la definizione : " ;
  cout<<endl<<endl<<"\t";
  getline (cin,defin);
  cout<<"\t_____"<<endl;
  TableH[k].push_front(Vocabolo(word,defin));
  return true;

}



bool VocabolarioHash::Delete ()
{
  string word;
  cout << endl <<  "\tVocabolo da eliminare : ";
  cout<<endl<<endl<<"\t";
  getline(cin,word);
  cout<<"\t_____"<<endl;
  int k = hashu((char*) word.c_str(), DIM); // calcola l'hash della parola da eliminare

  for(auto it = TableH[k].begin(); it!= TableH[k].end(); it++)
    if ( word==it->getParola())
    {
      TableH[k].erase(it);
      return true;
    }

  cout << endl <<"\tVocabolo Inesistente";
  cout<<endl<<"\t_____"<<endl;
  return false;
}

bool VocabolarioHash::Search (list<string> &SameW)
{
  string word;
  cout << endl <<  "\tVocabolo da cercare : ";
  cout<<endl<<endl<<"\t";
  getline(cin,word);
```

```
   cout<<"\t_____"<<endl;
   int k = hashu((char*)word.c_str() , DIM);

   string def;
   for (auto it = TableH[k].begin();it!= TableH[k].end();it++)
      if(word == it->getParola())
      {
         def = it->getDefin();
         cout << endl << "\tLa definizione della parola cercata e' : ";
         cout<<endl<<endl<<"\t"<< def;
         cout<<endl<<"\t_____"<<endl;
         return true;
      }
   cout << endl <<"\tVocabolo non presente "<< endl;
   cout<<"\t_____"<<endl;
   SameW = SimilWrd(word);
   return false;
}

list<string> VocabolarioHash::SimilWrd (string word)
{
   list <string> SameW;
   int x,y=(word.length())/2;
   string temp;

   for (int i = 0; i < DIM ; i++)
      for(auto it = TableH[i].begin(); it != TableH[i].end() ; it++)
      {
         temp= it->getParola();
         x = DistEdit(word,temp);
         if (x <= y)
            SameW.push_back(temp);
      }
   return SameW;
}


bool  VocabolarioHash::Print()
{
```

```
   bool check = false;
   int cont = 0;

   for (int i = 0 ; i < DIM ; i++)
      for(auto it = TableH[i].begin(); it!= TableH[i].end() ; it++)
      {
         cont++;
         if (cont == 1)
         {
            check=true;
            cout <<endl<<endl<<endl;
            cout  << "\tPAROLE CONTENUTE NEL VOCABOLARIO HASH :  ";
            cout <<endl<<endl<<endl;
         }
         cout << endl << "\tVOCABOLO :   " << it->getParola();
         cout <<endl << "\tDEFINIZIONE :   " << it->getDefin();
         cout << endl << "\tRIGA TABELLA HASH : " << i ;
         cout<<endl<<"\t_____"<<endl;


      }
   return check;
}


int VocabolarioHash::hashu (char *v , int m)
{
   int h=0, a = 31415 , b = 27183;
   for ( ;*v!=0 ;v++, a= a*b%(m-1))
      h= (a * h + *v) % m;
   return (h<0)? (h+m) : h;
}
```

7.3 Vocabolo.h

La classe Vocabolo ha i seguenti attributi :

string parola
string definizione

ed i seguenti metodi :

void setParola (string) modifica l'attributo parola cambiandolo con una stringa fornita come parametro

void setDefin (string) modifica l'attributo definizione cambiandolo con una stringa fornita come parametro

string getParola () Ritorna l'attributo parol
string getDefin() Ritorna l'attributo definizione

Codice :

```
"Vocabolo.h"

class Vocabolo
{
  public:

    Vocabolo(string parola , string definizione);  // Costruttore con inizializzazione
                    Vocabolo(){};  // Costruttore  di default
                    ~Vocabolo(){};   // distruttore

                    void setParola (string parola); //modifica  la parola
                    void setDefin (string definizione); //modifica la definizione
                    string getParola (); // Ritorna La parola
                    string getDefin(); // Ritorna La definiziione

          private:
     string parola;
     string definizione;

};
"Vocabolo.cpp"
```

```
#include "vocabolo.h"

// costruttore classe con inizializzazione
Vocabolo::Vocabolo(string parola,string definizione)
{
   this->parola = parola;
   this->definizione = definizione;
}

//modifica  la parola
void Vocabolo :: setParola (string parola)
{
   this->parola = parola;
}

//modifica la definizione
void Vocabolo :: setDefin (string definizione)
{
   this->definizione = definizione;
}

// Ritorna La parola
string Vocabolo :: getParola()
{
   return parola;
}

// Ritorna La definizione
string Vocabolo :: getDefin()
{
   return definizione;
}
```

7.4 VocabolarioRbTree

La classe VocabolarioRbTree è una classe che eredita alcuni metodi virtuali da implementare , dalla classe AbsVocabolario.

Come Attributo ha :

nodoRb *radice che è un puntatore a un oggetto della classe nodoRb

Ha i seguenti metodi pubblici :
I primi sono ereditati dalla classe AbsVocabolario e implementati secondo le caratteristiche della classe VocabolarioRbTree

bool Insert() Inserisce i nodi nell'albero
bool Delete () Elimina i nodi dall'albero
bool Search (list<string> &) Cerca un nodo altrimenti restituisce una lista di stringhe simili a quella contenuta nel nodo
list<string> SimilWrd (string) Nel caso fallisce la ricerca restituisce una lista di stringhe simili a quella cercata , calcolata con la distanza di editing

bool Print() Stampa tutti le informazioni dei nodi presenti

inoltre in aggiunta ha :

bool PrintRB() Stampa i nodi in formato albero

Inoltre ha altri metodi privati in quanto non utilizzati esternamente

alla classe , questi sono :

nodoRb* minimo (nodoRb*)
nodoRb* successore (nodoRb*)
nodoRb* RbDelete (nodoRb*)
void leftRotate (nodoRb*)
void rightRotate (nodoRb*)
void fixRbInsert (nodoRb*)
void fixRbDelete (nodoRb*)
void inorder_Dist (int , string , list<string> & , nodoRb*)
void print_inorder (nodoRb*)

void print_rb(nodoRb* nodo)
void del_postorder (nodoRb*)

Codice :

```
"VocabolarioRbTree.h"
class VocabolarioRbTree : public AbsVocabolario
{
        public:
                VocabolarioRbTree();  // class constructor
                ~VocabolarioRbTree() ; // class destructor

                bool Insert();
                bool Delete ();
                bool Search ( list<string> &);
                list<string> SimilWrd (string );
                bool Print();
                bool PrintRB();

        private:
    nodoRb* radice;
    nodoRb* minimo (nodoRb* );
    nodoRb* successore (nodoRb* );
    nodoRb* RbDelete (nodoRb* );
    void leftRotate (nodoRb* );
    void rightRotate (nodoRb* );
    void fixRbInsert (nodoRb* );
    void fixRbDelete (nodoRb* );
    void inorder_Dist (int , string , list<string> & , nodoRb* );
    void print_inorder (nodoRb*);
    void print_rb(nodoRb* nodo);
    void del_postorder (nodoRb* );
};
```

```
#include "vocabolariorbtree.h"


//COSTRUTTORE DELLA CLASSE
VocabolarioRbTree::VocabolarioRbTree()
{
   radice = nullptr;
}

//DISTRUTTORE
VocabolarioRbTree::~VocabolarioRbTree()
{
              del_postorder(radice);
}



///METODI

nodoRb* VocabolarioRbTree::successore (nodoRb* x)
{
   //CASO 1  : ESISTE FIGLIO DESTRO
   if ( x->getRight() != nullptr)
   {
      x = x->getRight();
      // il successore è il minimo
      // del sottoalbero destro di x
      while(x->getLeft() != nullptr)
         x = x->getLeft();
      return x;
   }

   nodoRb* y = x->getParent();
   //CASO 2-3  : NON ESISTE FIGLIO DESTRO DI X ,
   //         X FIGLIO SINISTRO
   // il successore è il padre di X

   //          NON ESISTE FIGLIO DESTRO DI X,
   //          X FIGLIO DESTRO
   // il successore è l'ultimo antenato per il
   // quale X si trova nel suo sottoalbero sinistro
```

```
   while (y != nullptr && x == y->getLeft())
      {
         x=y;
         y = y->getParent() ;
      }
   return y;
}

void VocabolarioRbTree::leftRotate (nodoRb* x)
{
   if (x->getRight() == nullptr)
      return;
   nodoRb* y;  // nodo di appoggio
   y = x->getRight();   // y diventa figlio destro di x
   x->setRight(y->getLeft()); // Imoosto il figlio sinistro di y  come figlio destro di x
   y->setParent(x->getParent());
   if(x->getParent() == nullptr)
      radice = y;
   else
   {
      if (x->getParent()->getLeft()== x)
         x->getParent()->setLeft(y);
      else
         x->getParent()->setRight(y);
   }
    y->setLeft(x);
    x=y;
}


void VocabolarioRbTree::rightRotate (nodoRb* x)
{
   if (x->getLeft() == nullptr)
      return;
   nodoRb* y;  // nodo di appoggio
   y = x->getLeft();   // y diventa figlio destro di x
   x->setLeft(y->getRight()); // Imoosto il figlio sinistro di y  come figlio destro di x
   //y->setLeft(x);
   y->setParent(x->getParent());
```

```
    if(x->getParent() == nullptr)
        radice = y;
    else
    {
        if (x->getParent()->getRight()== x)
            x->getParent()->setRight(y);
        else
            x->getParent()->setLeft(y);
    }
     y->setRight(x);
     x=y;
}



bool VocabolarioRbTree::Insert()
{
    string word ;
    bool check = false;
    cout << endl << "\tVocabolo da inserire : ";
    cout<<endl<<endl<<"\t";
    getline(cin,word);
    cout<<"\t_____"<<endl;
    nodoRb* x;  // nodo di appoggio per la ricerca nell'albero
    nodoRb* y; // padre di x
    x = radice;
    y = nullptr;
    //ricerca posizione da inserire
    while ( x!= nullptr)
    {
        if( word == x->getParola() )
        {
            cout << endl <<  "\tVocabolo gia presente";
            cout<<endl<<"\t_____"<<endl;
            return check;
        }
        // altrimenti si muove nel figlio sinistro se la parola è piu piccola
        // o viceversa nel figlio destro
        y = x;
```

```
      x = (x->getParola() > word)?  x->getLeft()  : x->getRight();
   }
   //altrimenti inserisce la parola
   check  = true; // AVVISO CHE LA PAROLA SARA INSERITA
   string def;
   cout << endl << "\tInserisci la definizione : ";
   cout<<endl<<endl<<"\t";
   getline (cin,def);
   cout<<"\t_____"<<endl;
   x = new nodoRb(word,def); //inizializzo il nuovo nodo come  RED

   if (y == nullptr)  // SE NON CE NESSUN NODO NELL'ALBERO
      radice = x; // la radice punterà al nuovo nodo creato

   else  // ALTRIMENTI SI POSIZIONA NELLA FOGLIA GIUSTA
      {
         if ( word < y->getParola())
         y->setLeft(x);
         else
         y->setRight(x);
      }

    if (check)  // SE IL NODO E' STATO INSERITO
      fixRbInsert(x); //SI RIBILANCIA l'ALBERO
   return check;

}


void VocabolarioRbTree::fixRbInsert(nodoRb* x)
{
   nodoRb* p;// PADRE DI X
   nodoRb* n; // NONNO DI X
   nodoRb* z; // ZIO DI X
   while (x!= nullptr)
   {
      // RIMPOSTO AD OGNI ITERAZIONE
      // PADRE
      p  = x->getParent();
```

```
//NONNO
n = ( p != nullptr)? p->getParent() : nullptr;

//ZIO
if (n == nullptr)
   z = nullptr;
else
z =  (p == n->getLeft()) ? n->getRight() : n->getLeft();

   // CASO 1 PRIMO NODO A ESSER INSERITO
if ( p == nullptr)
{
   x->setColor(false) ;       // X DIVENTA NERO
   x = nullptr; // si esce dal while
}

// CASO 2 PADRE DI X NERO  - NESSUN VINCOLO VIOLATO
else if(p->checkRed()== false)
   x = nullptr;

else if (z!= nullptr)
{
   if(z->checkRed())
    {
      //CASO 3  :  PADRE ROSSO , ZIO ROSSO ,x ROSSO
       p->setColor(false); //COLORO IL PADRE NERO
       z->setColor(false);// COLORO LO ZIO NERO
      if (n != nullptr)
         n->setColor(true); // COLORO IL NONNO ROSSO
      x = n;// SPOSTIAMO IL PROBLEMA + IN ALTO
    }
   // CASO 4
   // p FIGLIO SINISTRO DI N
   // p NERO , z ESISTE ED é NERO , x ROSSO
   // 2 SOTTO CASI , x FIGLIO DESTRO OR x FIGLIO SINISTRO
   else
   {
      // 4.a X FIGLIO DESTRO
```

```
        if (x==p->getRight() &&  p == n->getLeft())
        {
            leftRotate(p);
            x = p;
        }
        // 4.b X FIGLIO SINISTRO
        else if (x==p->getLeft() && p == n->getRight())
        {
            rightRotate(p);
            x = p;
        }
        // CASO 5
        // p ROSSO , z NERO  , x ROSSO
        // 2 SOTTO CASI ,
        else
        {  //5.a x figlio sinistro di p , p FIGLIO SINISTRO DI N
            if (x==p->getLeft() && p == n->getLeft())
                rightRotate(n);
            else if ( x==p->getRight() && p == n->getRight())
                leftRotate(n);
        p->setColor(false); // coloro p di nero
        n->setColor(true); // coloro n di rosso
        x = nullptr;
        }
    }
}

// CASO 4 e 5 CON ZIO NULLPTR QUINDI NERO
// UGUALI AI PRECEDENTI MA SI EVITA CONTROLLO
else
{
    if (x==p->getRight() &&  p == n->getLeft())
    {
        leftRotate(p);
        x = p;
    }
    else if (x==p->getLeft() && p == n->getRight())
    {
        rightRotate(p);
```

```
            x = p;
        }
        else
        {
            if (x==p->getLeft() && p == n->getLeft())
                rightRotate(n);
            else if ( x==p->getRight() && p == n->getRight())
                    leftRotate(n);

            p->setColor(false); // coloro p di nero
            n->setColor(true); // coloro n di rosso
            x = nullptr;
        }

      }
   }
}
```

```
bool VocabolarioRbTree::Delete ()
{
   bool check = false;
   string word ;
   nodoRb* z = radice; // per la ricerca
   nodoRb* y; //nodo da eliminare

   // Inseriamo vocabolo da eliminare
   cout << endl <<  "\tVocabolo da eliminare : ";
   cout<<endl<<endl<<"\t";
   getline(cin,word);
   cout<<"\t_____"<<endl;

   // FASE 1 : RICERCA DELLA PAROLA NELL' ALBERO
   while( z != nullptr && z->getParola() != word)
      z = (z->getParola() > word)?  z->getLeft()  : z->getRight();

   if ( z == nullptr) // nel caso non viene trovata
```

```
    {
        cout << endl <<"\tVocabolo Inesistente";
        cout<<endl<<"\t_____"<<endl;
        return check;
    }
    check=true;  // parola trovata  = z
    //se la parola trovata è nella  radice e la radice non ha figli
    if (z == radice && (z->getRight()==nullptr && z->getLeft()== nullptr))
    {
        delete z;
        radice = nullptr;
        return check;

    }
    //altrimenti verifichiamo quale nodo effettivamente va eliminato
    y = RbDelete(z);
    delete y; //deallochiamo il nodo da eliminare
    return check;
}

nodoRb* VocabolarioRbTree::RbDelete (nodoRb* z)
{
    nodoRb* y;
    nodoRb* x;
    nodoRb* temp =  nullptr;
    // COME PRIMO PASSAGGIO CONTROLLIAMO SE
    if (z->getLeft()== nullptr || z->getRight() == nullptr) // CASO 1 AL + UN FIGLIO
          y=z; // Il nodo da eliminare è esattamente z
    else
        y = successore(z); // CASO2   -   2 FIGLI
        // Il nodo da eliminare è il suo successore


    // Adesso Impostiamo il nodo X che serve a ribilanciare successivamente
    // l'albero :

    if (y->getLeft() != nullptr)
        x=y->getLeft(); //
```

```
else if (y->getRight()!= nullptr) // VICEVERSA , DESTRO
   x=y->getRight();

else    // Y NON HA FIGLI
        x = new nodoRb("NIL");

if (y->getParent() == nullptr) // SE IL PADRE è NULLPTR
  {
     radice = x; // X è La radice
     x->setParent(nullptr);
  }
    //ALTRIMENTI EFFETTUIAMO LO SHORTCUT
else
{
   if (y == y->getParent()->getLeft())
      y->getParent()->setLeft(x);
   else
      y->getParent()->setRight(x);
}

      // sostituiamo le chiavi
if (y!= z)
{
   z->setParola(y->getParola());
   z->setDefin(y->getDefin());
}
if (x->getParola()=="NIL")  // se il nodo di appoggio é stato creato
  temp = x; // ce lo copiamo in un nodo temp

if (y->checkRed()== false)  // se il nodo da eliminare é nero
   fixRbDelete (x);       // Ribilanciamo l'albero RB

if (temp != nullptr) // elimino il nodo di appoggio creato in precedenza
{               // effettuando uno shortcut dal padre
   if (temp == temp->getParent()->getLeft())
      temp->getParent()->setLeft(nullptr);
```

```
      else
         temp->getParent()->setRight(nullptr);

      delete temp; // il nodo temp viene deallocato
   }


   return y;
}

void VocabolarioRbTree::fixRbDelete (nodoRb* x)
{
   nodoRb* brother;
   while (x != radice && x->checkRed()== false)
   {

      if (x == x->getParent()->getLeft()) // CASI (1-3)A x Figlio sinistro
      {
         brother = x->getParent()->getRight();  // fratello figlio destro
         if(brother->checkRed())
         {    //  CASO 1.a  --> il fratello di x è rosso
            x->getParent()->setColor(true);  // il padre di x è rosso
            brother->setColor(false); //  il fratello di x diventa nero
            leftRotate(x->getParent()); //padre(x) effettua una rotazione a sin
            brother = x->getParent()->getRight(); // reimposto w come fratello di x
         }


         if ( brother->getLeft()->checkRed()== false && brother->getRight()->checkRed() == false)
         {  //CASO 2.a  --> il fratello di x è NERO e ha FIGLI NERI
            brother->setColor(true);  // fratello di x diventa rosso
            x = x->getParent(); //spostiamo il problema più in alto
         }

         else
         {
            if (brother->getRight()->checkRed() == false) //
            { //CASO 3.a  --> fratello di x è nero e ha il figlio sinistro rosso
               brother->getLeft()->setColor(false); //il figlio sinistro diventa nero
```

```
            brother->setColor(true); // e il fratello diventa rosso
            rightRotate(brother); // effettuiamo una rotazione a destra
            brother = x->getParent()->getRight(); // reimpostiamo il fratello
            // e ci portiamo nel caso 4
          }// CASO 4.a
          brother->setColor(x->getParent()->checkRed()); // settiamo lo stesso colore del padre di x
          brother->getParent()->setColor(false);  // il padre del fratello diventa nero
          brother->getRight()->setColor(false); // il figlio destro diventa nero
          leftRotate(x->getParent()); // effettuiamo una rotazione a sinistra
          x = radice; // e settiamo x = alla radice e si esce dal while
        }

    }
    else  // CASI (1-3)B x Figlio destro
       // simmetrici ai precedenti
    {
       brother = x->getParent()->getLeft();
       if(brother->checkRed())
       {  //  CASO 1.b
          x->getParent()->setColor(true);
          brother->setColor(false);
          rightRotate(x->getParent());
          brother= x->getParent()->getLeft();
       }
       if ( brother->getLeft()->checkRed()== false && brother->getRight()->checkRed()== false)
       { // CASO 2.b
          brother->setColor(true);
          x = x->getParent();
       }
       else
       {

          if (brother->getLeft()->checkRed() == false)
          { //CASO 3.b
             brother->getRight()->setColor(false);
             brother->setColor(true);
             leftRotate(brother);
             brother = x->getParent()->getLeft();
          }// CASO 4.b
```

```
            brother->setColor(x->getParent()->checkRed());
            x->getParent()->setColor(false);
            brother->getLeft()->setColor(false);
            rightRotate(x->getParent());
            x = radice;
          }

        }

  }
  x->setColor(false); // x viene colorato di nero
}


bool VocabolarioRbTree::Search ( list<string> &SameW)
{
  string word ;
  cout << endl <<  "\t\tVocabolo da cercare : ";
  cout<<endl<<endl<<"\t";
  getline(cin,word);
  cout<<"\t_____"<<endl;
  nodoRb* z = radice; // per la ricerca

  // FASE 1 : RICERCA DELLA PAROLA NELL' ALBERO
  while( z != nullptr && z->getParola() != word)
     z = (z->getParola() > word)?  z->getLeft()  : z->getRight();

  if ( z == nullptr) // nel caso non viene trovata
  {
     cout << endl <<  "\t\tVocabolo non presente " << endl ;
     cout<<"\t_____"<<endl;
     SameW = SimilWrd(word);
     return false;
  }
  string def = z->getDefin();
  cout << endl <<"\tLa definizione della parola cercata e' : "<< endl;
  cout<<"\t" << def << endl;
  cout<<"\t_____"<<endl;
  return true;
```

```
}



list<string> VocabolarioRbTree::SimilWrd (string word)
{
   list <string> SameW;
   int y=(word.length())/2;
   inorder_Dist(y,word,SameW,radice);
   return SameW;
}

void VocabolarioRbTree::inorder_Dist (int y , string word,list<string>  &SameW , nodoRb* nodo)
{
   if(nodo == nullptr)
      return;
   inorder_Dist(y , word ,SameW , nodo->getLeft());
   string temp = nodo->getParola();
   int x = DistEdit(word,temp);
   if (x<=y)
      SameW.push_back(temp);
   inorder_Dist(y , word , SameW,nodo->getRight());
   return;
}

bool VocabolarioRbTree::PrintRB()
{
   if (radice == nullptr)
      return false;
   else
   {
      cout <<endl<<endl<<endl;
      cout  << "\t\tVISTA ALBERO RB";
      cout <<endl<<endl<<endl;
      print_rb(radice);
   }
   return true;
}

void VocabolarioRbTree::print_inorder(nodoRb* nodo)
```

```
{
   if(nodo == nullptr)
      return;
   print_inorder(nodo->getLeft());
   cout <<endl << "\tVOCABOLO :   " << nodo->getParola();
   cout <<endl << "\tDEFINIZIONE:   " << nodo->getDefin();
   cout<<endl<<"\t_____"<<endl;
   print_inorder(nodo->getRight());
}

bool VocabolarioRbTree::Print()
{
   if (radice == nullptr)
      return false;
   else
    {
      cout <<endl<<endl<<endl;
      cout  << "\t\tPAROLE CONTENUTE NEL VOCABOLARIO RED&BLACK ";
      cout <<endl<<endl<<endl;
      print_inorder(radice);
    }
   return true;
}
void VocabolarioRbTree::print_rb(nodoRb* nodo)
{
   if ( nodo == nullptr)
      return;

   string s1,s2,s3;
   s1 = "\t";
   s2 = "          ";
   nodoRb* temp = nullptr;
   temp = nodo->getParent();
   while (temp != nullptr)
   {
        temp = temp->getParent();
        s1 += s2;
   }
   s3 = s1 + s2;
```

```
    print_rb(nodo->getRight());

    if(nodo->getRight() == nullptr)
    {
        cout <<endl<<endl;
        cout <<endl<<s3 + "NIL";
        cout<<endl<<s3 + "PADRE : ";
        cout << nodo->getParola();
        cout <<endl<<s3 + "Colore : ";
        cout<< "NERO";
        cout<<endl;
    }

    cout <<endl<<endl;
    cout <<endl<<s1 + nodo->getParola();
    if (nodo->getParent()!= nullptr)
    {
        cout<<endl<<s1 + "PADRE : ";
        cout << nodo->getParent()->getParola();
    }
    else
        cout<<endl<<s1 + "RADICE";
    cout <<endl<<s1 + "Colore : ";
    if (nodo->checkRed())
        cout <<"ROSSO"<<endl;
    else
        cout <<"NERO"<<endl;

    if(nodo->getLeft() == nullptr)
    {
        cout <<endl<<endl;
        cout <<endl<<s3 + "NIL";
        cout<<endl<<s3 + "PADRE : ";
        cout << nodo->getParola();
        cout <<endl<<s3 + "Colore : ";
        cout<< "NERO"<<endl;
    }
    print_rb(nodo->getLeft());
}
```

```
void VocabolarioRbTree::del_postorder (nodoRb* nodo)
{

   if(nodo == nullptr)
      return;
   del_postorder(nodo->getLeft());
   del_postorder(nodo->getRight());
   delete nodo;
}
```

7.5 nodoRb

La classe NodoRb è una classe che eredita attributi dalla classe Vocabolo , quindi ogni nodo contiene due attributi stringa : parola e definizione

Ha in aggiunta i seguenti attributi :
nodoRb *left, *right, *parent
puntatori ad altri nodi

bool isred; : variabile booleana per verificare il colore

Ha i seguenti metodi :

bool checkRed(); : ritorna il colore del nodo

void setColor (bool) : setta il colore del nodo

void setRight(nodoRb*) : setta il nodo destro

void setLeft(nodoRb*) : setta il nodo sinistro

void setParent(nodoRb*) : setta il nodo padre

nodoRb* getLeft() : ritorna il puntatore al nodo sinistro

nodoRb* getRight() : ritorna il puntatore al nodo destro

nodoRb* getParent() : ritorna il puntatore al nodo padre

Codice :

"nodoRb.h"

```
class nodoRb : public Vocabolo
{
        public:
                // class constructor
                nodoRb();
                nodoRb(string word,string def);
                nodoRb(string word);

                // class destructor
                ~nodoRb();
                bool checkRed();          // ritorna il colore del nodo
    void setColor (bool color);    // setta il colore del nodo
                void setRight(nodoRb* des);   // setta il nodo destro
                void setLeft(nodoRb* sin);   // setta il nodo sinistro
                void setParent(nodoRb* parent);   // setta il nodo padre
                nodoRb* getLeft();          // ritorna il puntatore al nodo sinistro
                nodoRb* getRight();         // ritorna il puntatore al nodo destro
                nodoRb* getParent();        // ritorna il puntatore al nodo padre


        private:
    nodoRb *left, *right, *parent;  // puntatori al padre, e figli
    bool isred;               // var.booleana per il colore

};
```

"nodoRb.cpp"

```
#include "nodorb.h"

// Costruttore Default
nodoRb::nodoRb()
{
   this->isred = false;
   left = nullptr;
   right = nullptr;
```

```
    parent = nullptr;
}

//Costruttore NIL

nodoRb::nodoRb(string word)
{
    this->isred = false;
    this->setParola (word);
    left = nullptr;
    right = nullptr;
    parent = nullptr;
}

//COSTRUTTORE CON PARAMETRI
nodoRb::nodoRb(string word,string def)
{
    this->isred = true;
    this->setParola (word);
    this->setDefin (def);
    left = nullptr;
    right = nullptr;
    parent = nullptr;
}

// DISTRUTTORE NODO
nodoRb::~nodoRb()
{
}

//METODI



void nodoRb::setColor (bool color)
{
      if (this == nullptr)
          return;
      else
          this->isred = color;
```

```
}



void nodoRb::setRight(nodoRb* des)
{
   this->right = des;
   if(des != nullptr)
      des->parent = this;
}

void nodoRb::setLeft(nodoRb* sin)
{
   this->left = sin;
   if(sin != nullptr)
      sin->parent = this;
}
void nodoRb::setParent(nodoRb* parent)
{
      this->parent = parent;
}

bool nodoRb::checkRed()
{
   if (this==nullptr)
      return false;
   else
      return this->isred;
}




nodoRb* nodoRb::getLeft()
{
   return this->left;
}

nodoRb* nodoRb::getRight()
{
   return this->right;
```

```
}

nodoRb* nodoRb::getParent()
{
   return this->parent;
}
```

7.6 Librerie utilizzate

IN QUESTO SEGUENTE HEADER SONO STATI RACCHIUSI I VARI MENU STAMPATI DEL PROGRAMMA , E TUTTE LE LIBRERIE UTILIZZATE

"projectheader.h"

```
#ifndef PROJECTHEADER_H_INCLUDED
#define PROJECTHEADER_H_INCLUDED
#include <iostream>
#include <conio.h>
#include <string>
#include <list>
#include <cstdlib>

using namespace std;
void sceltaVocabolario();
void opzioniH();
void opzioniRB();
void firma ();
void buttContinue();
```

Codice :

"projectheader.cpp"

```
#include "projectheader.h"
void firma ()
{
   cout << "\n\n\n\n\n";
   cout <<"\t************************************************************\n";
   cout <<"\t*                                          *\n";
   cout <<"\t*              GESTIONE VOCABOLARIO                *\n";
   cout <<"\t*          ESAME ALGORITMI E STRUTTURE DATI          *\n";
   cout <<"\t*           MATRICOLA  :  0124//000955              *\n";
   cout <<"\t*                QUINZIO ANTONIO                   *\n";
   cout <<"\t*                                          *\n";
   cout <<"\t************************************************************\n";
}

void sceltaVocabolario()
{
   cout << "\n\n\n\n\n";
   cout << "\t*******************************************" << endl;
   cout << "\t|                                         |" << endl;
   cout << "\t|                 MENU                    |" << endl;
   cout << "\t|                                         |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t|   [1]       Vocabolario Hash            |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t|   [2]       Vocabolario Red&Black       |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t|  [ESC]      ESCI DAL PROGRAMMA          |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t       SCELTA:       ";



}
```

```
void opzioniH()
{
   cout << "\n\n";
   cout << "\t*******************************************" << endl;
   cout << "\t|                                         |" << endl;
   cout << "\t|           VOCABOLARIO HASH              |" << endl;
   cout << "\t|                                         |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t|   [1]       INSERISCI                   |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t|   [2]       CERCA                       |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t|   [3]       ELIMINA PAROLA              |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t|   [4]      STAMPA VOCABOLARIO           |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t|  [<--]     RETURN SCELTA INIZIALE       |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t      SCELTA:       ";


}

void opzioniRB()
{
   cout << "\n\n";
   cout << "\t*******************************************" << endl;
   cout << "\t|                                         |" << endl;
   cout << "\t|         VOCABOLARIO RED & BLACK         |" << endl;
   cout << "\t|                                         |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t|   [1]       INSERISCI                   |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t|   [2]       CERCA                       |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t|   [3]       ELIMINA PAROLA              |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t|   [4]      STAMPA VOCABOLARIO           |" << endl;
   cout << "\t*******************************************" << endl;
```

```
   cout << "\t|   [5]       VERIFICA ALBERO-RB        |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t|   [<--]      RETURN SCELTA INIZIALE     |" << endl;
   cout << "\t*******************************************" << endl;
   cout << "\t       SCELTA:       ";
}

void buttContinue()
{
   cout << endl << endl << "\t PREMERE 'INVIO/ENTER' PER CONTINUARE";
   while ( _getch()!= 13); // Mentre è diverso dall' Invio...
}
```

7.7 MAIN

Codice :

```
#include "projectheader.h"
#include "absvocabolario.h"
#include "vocabolariohash.h"
#include "nodorb.h"
#include "vocabolariorbtree.h"

/*************************************************************************
*            Quinzio Antonio  - Matricola 0124/955                *
*         ----------------TRACCIA 1 ------------------             *
* 1. Costruire un vocabolario V utilizzando un Hash Table con il metodo *
* della concatenazione, che abbia le seguenti funzioni:              *
* – Inserimento del termine                                          *
* – Cancellazione                                                    *
* – Ricerca del termine. In caso di fallimento deve restituire una     *
*   lista delle parole pi´u prossime, utilizzando un approccio basato   *
*   sulla Distanza di editing.                                       *
*                                                                    *
```

```
* 2. Costruire un vocabolario V', utilizzando un albero RED BLACK che   *
* abbia le stesse funzioni, dell' esercizio precedente.                 *
*                                                                       *
* Nell' implementazione di entrambi i quesiti si faccia uso delle Classi*
* Astratte.                                                             *
*************************************************************************
*/



int main()
{
   int scelta,scelta2 ;
   list <string> sameW;  // Lista di appoggio per le parole simili
   VocabolarioHash *V ; // vocabolario hash
   VocabolarioRbTree *V_perm; // vocabolario RB
   firma();
   buttContinue();

   while(scelta != 27)//TASTO ESC
   {
      scelta2 = 0 ;
      system("CLS");
      sceltaVocabolario();//stampa il menu di scelta del tipo del vocabolario sullo schermo
      switch(scelta = _getch())
      {
         case '1' : //Vocabolario Hash

            V = new VocabolarioHash;
            while(scelta2 != 8 ) //TASTO BACKSPACE
            {
               system("CLS");
               opzioniH();  // Stampa opzioni vocabolario hash
               switch(scelta2 = _getch())
               {
                  // INSERISCI PAROLA;
                  case '1' :
                  system("CLS");
                  if(V->Insert())   // SE RIESCE A INSERIRE
                   {
```

```
   cout<< endl<<endl<<endl<<"\tVOCABOLO INSERITO CORRETTAMENTE";
   cout<<endl<<"\t_____"<<endl;
 }
buttContinue();
break;


 // CERCA PAROLA;
case '2' :
system("CLS");
if (!V->Search(sameW) && sameW.begin() != sameW.end())
{
   cout << endl << "\tFORSE CERCAVI  : ";
   cout << endl<<endl;
   for (auto it = sameW.begin();it!= sameW.end();it++)
      cout<<"\t"<< *it << endl;
   cout<<"\t_____"<<endl;
}
buttContinue();
break;

case '3' : // ELIMINA PAROLA
system("CLS");
if(V->Delete())
{
   cout<<endl<<endl<<endl<<"\tVOCABOLO ELIMINATO CORRETTAMENTE";
   cout<<endl<<"\t_____"<<endl;
}
buttContinue();
break;

case '4' : //STAMPA VOCABOLARIO
system("CLS");
if (!V->Print())
{
   cout <<endl<<endl<<endl<< "\tVOCABOLARIO VUOTO";
   cout<<endl<<"\t_____"<<endl;

}
```

```
            buttContinue();
            break;




            default :
            break;



        }
    }
    delete V;
    break;
case '2' ://Vcocabolario Rb
    V_perm = new VocabolarioRbTree;
    while (scelta2 != 8) //TASTO BACKSPACE O ESC
    {
        system("CLS");
        opzioniRB();  // stampa opzioni vocabolario RB
        scelta2 = _getch() ;
        switch(scelta2)
        {
            case '1' : // INSERISCI PAROLA;
            system("CLS");
            if(V_perm->Insert())
            {
                cout<< endl<<endl<<endl<<"\tVOCABOLO INSERITO CORRETTAMENTE";
                cout<<endl<<"\t_____"<<endl;
            }
            buttContinue();
            break;

            case '2' : // CERCA PAROLA;
            system("CLS");
            if (!V_perm->Search(sameW) && sameW.begin() != sameW.end())
            {
                cout << endl << "\tFORSE CERCAVI  : ";
```

```
   cout << endl<<endl;
   for (auto it = sameW.begin();it!= sameW.end();it++)
      cout<<"\t" << *it << endl;
   cout<<endl<<"\t_____"<<endl;

}
buttContinue();
break;

case '3' : // ELIMINA PAROLA
system("CLS");
if(V_perm->Delete())
{
   cout<< endl <<"\tVOCABOLO ELIMINATO CORRETTAMENTE";
   cout<<endl<<"\t_____"<<endl;
}
buttContinue();
break;


case '4' : //STAMPA VOCABOLARIO
system("CLS");
if (!V_perm->Print())
{
   cout <<endl<<endl<<endl<< "\tVOCABOLARIO VUOTO";
   cout<<endl<<"\t_____"<<endl;

}
buttContinue();
break;

case '5' : //STAMPA ALBERO BINARIO
system("CLS");

if (!V_perm->PrintRB())
{
   cout <<endl<<endl<<endl<< "\tALBERO VUOTO";
   cout<<endl<<"\t_____"<<endl;
```

```
                }
                buttContinue();
                break;

                default :
                break;
              }
            }
            delete V_perm;
            break;
        }

    }

    return EXIT_SUCCESS;
}
```