



Dipartimento di Informatica - Corso 2018 / 2019

Algoritmi e Strutture Dati

Traccia 4:

**Fusione Alberi Red-Black  
Grafo Fortemente Connesso**

---

Realizzato da:

Di Costanzo Salvatore

# Contenuti

<b>1</b>	<b>Descrizione Problema 1</b>	<b>2</b>
1.1	Alberi Red-Black . . . . .	3
1.2	Formato Dati I/O . . . . .	3
1.3	Merge Alberi . . . . .	4
1.4	UML - Problema 1 . . . . .	6
1.5	Test e Risultati . . . . .	8
1.6	Codice . . . . .	10
1.6.1	<i>main.cpp</i> . . . . .	10
1.6.2	<i>Nodo.h</i> . . . . .	12
1.6.3	<i>Albero.h</i> . . . . .	15
1.6.4	<i>Merge.h</i> . . . . .	23
1.6.5	<i>Trunck.h</i> . . . . .	25
<b>2</b>	<b>Descrizione Problema 2</b>	<b>26</b>
2.1	Grafo Orientato . . . . .	27
2.2	Formato Dati I/O . . . . .	28
2.3	Ricerca delle Componenti fortemente connesse . . . . .	29
2.4	UML - Problema 2 . . . . .	30
2.5	Test e Risultati . . . . .	31
2.6	Codice . . . . .	33
2.6.1	<i>main.cpp</i> . . . . .	33
2.6.2	<i>Nodo.h</i> . . . . .	35
2.6.3	<i>Grafo.h</i> . . . . .	38

## 1 Descrizione Problema 1

Nel Problema 1 viene richiesto di implementare un algoritmo in grado di eseguire il *Merge* ( ovvero la fusione) di due *Alberi Red-Black* in un tempo pari a  $O(m+n)$  dove  $n$  e  $m$  rappresentano rispettivamente la grandezza dei due alberi *T1* e *T2* dati in input.

Prima di procedere con l'illustrazione del codice procediamo ad introdurre brevemente la struttura dati *Alberi Red-Black* e a descriverne le proprietà. Passeremo poi alla descrizione dell'algoritmo che dato in input un vettore rappresentativo dei due alberi fusi sia in grado costruire un nuovo albero *T3* con la complessità richiesta.

## 1.1 Alberi Red-Black

Un **Albero Red-Black** è un Albero Binario di Ricerca (ABR) con un bit aggiuntivo di memoria per ogni nodo: il **colore** del nodo, che può essere RED (rosso) oppure BLACK (nero). Assegnando dei vincoli al modo in cui i nodi possono essere colorati lungo un qualsiasi cammino semplice che va dalla radice a una foglia, gli **Alberi Red-Black** garantiscono che nessuno di tali cammini sia lungo più del doppio di qualsiasi altro, quindi l'albero è approssimativamente **bilanciato**. Ogni nodo dell'albero contiene gli attributi Color, Key, Sx, Dx e Parent. Se manca un figlio o il padre di un nodo, il corrispondente attributo puntatore del nodo contiene il valore **NIL**.

Un **Albero Red-Black** come già detto è un Albero Binario di Ricerca (ABR) che soddisfa le seguenti **proprietà**:

1. Ogni nodo è rosso o nero.
2. La radice è nera.
3. Ogni foglia (**NIL**) è nera.
4. Se un nodo è rosso, allora entrambi i suoi figli sono neri.
5. Per ogni nodo, tutti i cammini semplici che vanno dal nodo alle foglie sue discendenti contengono lo stesso numero di nodi neri.

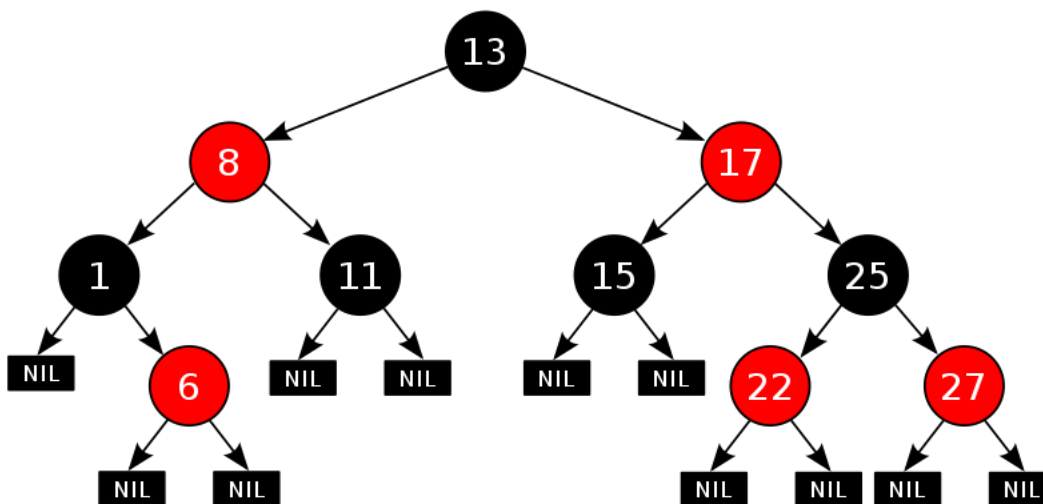


Figura 1: Esempio di un **Albero Red-Black**

## 1.2 Formato Dati I/O

Per questa implementazione è previsto l'utilizzo di valori di Input di tipo **Intero**, quindi forniti inizialmente dei valori da inserire nei due alberi si otterrà una stampa di un **ASCII Tree** contenente i valori dei due alberi fusi.

### 1.3 Merge Alberi

Primo passo essenziale per la realizzazione della Traccia è andare ad effettuare la fusione tra i due alberi **T1** e **T2** in un array di appoggio. Procediamo quindi col richiamare la funzione **AFUS** (Algoritmo di Fusione che banalmente non è altro che la procedura **Merge** di **Merge-Sort**) della classe **Merge** passando gli alberi da fondere.

Gli step che andiamo a realizzare per la costruzione dei vettori utili alla costruzione dell'**Albero T3** sono:

1. Dato l'Albero **T1** utilizzando il metodo **Scarica** della classe **Albero** generiamo un vettore usando una visita in-Order riempiamo il vettore **vetT1**. Complessità pari a  $\Theta(\text{DimT1})$ .
2. Dato l'Albero **T2** utilizzando il metodo **Scarica** della classe **Albero** generiamo un vettore usando da una visita in-Order riempiamo il vettore **vetT2**. Complessità pari a  $\Theta(\text{DimT2})$ .
3. Utilizziamo il metodo **AFUS** per andare a costruire il vettore che rappresenta i due Alberi fusi.

Analizzando la complessità del solo metodo di fusione (**AFUS**) è possibile notare come l'operazione dominante è quella di confronto tra un elemento del vettore rappresentativo dell'albero **T1** e un elemento rappresentativo dell'albero **T2**.

L'**AFUS** effettua  $n = \text{DimT1} + \text{DimT2}$  passi, cioè quanti sono gli elementi che andranno a comporre il vettore **T3**.

Se si è nel primo scenario (primo **if**), allora è necessario esattamente un confronto per determinare un elemento di **T3**.

Se si è nello scenario 2 o nello scenario 3 (secondo e terzo **if**), allora non si effettua alcun confronto. Quindi si può concludere che il numero totale di confronti è al più  $n = \text{DimT1} + \text{DimT2}$ .

Si noti comunque che si tratta di una complessità di caso peggiore.

In conclusione la complessità di tempo dell'AFUS è  $O(n)$  o per meglio dire  $O(\text{DimT1} + \text{DimT2})$  cioè la somma delle dimensioni dei vettori **T1** e **T2**.

Passiamo ora alla vera e propria realizzazione dell'**Albero Red-Black**, è stato predisposto un costruttore all'interno della classe **Albero** del progetto che preso in input il vettore elaborato dalla classe **Merge** sia in grado di andare a costruire ricorsivamente l'**Albero** senza usare la procedura di Inserimento che fa uso dei metodi tipici dei Red-Black, ma utilizzando un metodo **Costruisci** che sia in grado di costruire l'Albero, e un metodo **FixColore** per effettuare la procedura di ricolorazione.

I passi logici per la creazione di questo nuovo **Albero** sono i seguenti:

1. Dato il **Vettore** di riferimento selezioniamo il mediano e creiamo un nuovo **nodo** di colore nero e con chiave uguale all'elemento mediano selezionato.
2. Settiamo come **Parent** di questo nuovo nodo l'elemento precedente (al primo passo Parent = **NIL**).
3. Andiamo a collocare i figli **Sx** e **Dx** come elementi mediani delle due sotto porzioni del vettore dato in input che avranno come **Parent** l'elemento mediano al passo precedente.
4. Alla fine della procedura di costruzione (quando siamo arrivati alle foglie) verranno settati i **figli** **Sx** e **Dx** come nodi **NIL**.
5. Alla fine di tutto il procedimento ritorniamo la radice.

Analizzando la complessità della funzione di costruzione possiamo notare come questa venga effettuata in maniera ricorsiva andando a costruire l'**Albero** toccando una sola volta ognuno degli elementi che compongono il vettore dato in input.

Possiamo quindi concludere che la complessità di tempo dell'algoritmo di Costruzione è  $\Theta(\text{Dim}T3)$ . Costruito l'**Albero** dobbiamo andare ad effettuare la ri-colorazione dei nodi per rendere il nostro Albero un effettivo **Albero Red-Black**.

I passi eseguiti per andare ad effettuare la **Ricolorazione** (metodo **FixColore**) sono i seguenti:

1. Dato un **nodo** verifica se è diverso da **NIL**.
2. Se il controllo risulta positivo allora verifica che non si tratti della **Radice**.
3. Verificata l'ipotesi precedente controlliamo se il **nodo** in analisi rappresenti una **foglia** (verifichiamo che i puntatori **Sx** e **Dx** del nodo puntino a **NIL**).
- 4.1 Se stiamo trattando una **foglia** effettua la seguente verifica, se il **nodo** in analisi e suo fratello sono entrambi foglie e la sua profondità è uguale all'altezza dell'Albero allora coloriamo il **nodo** di RED (rosso) e setta il colore del **Parent** BLACK(nero).
- 4.2 Altrimenti se il **nodo** in analisi è **figlio Dx**, si trova a una profondità pari all'altezza dell'Albero e suo fratello punta a **NIL** allora coloriamo il **nodo** di RED (rosso) e setta il colore del **Parent** BLACK(nero).
- 4.3 Altrimenti se il **nodo** in analisi è **figlio Sx**, si trova a una profondità pari all'altezza dell'Albero e suo fratello punta a **NIL** allora coloriamo il **nodo** di RED (rosso) e setta il colore del **Parent** BLACK(nero).
5. Se il **nodo** in analisi non è una **foglia** per colorarlo basta verificare che **Parent**, **Figlio Sx** e **Figlio Dx** siano BLACK(nero), in caso affermativo possiamo colorare il **nodo** in esame di RED(rosso).

Per tenere traccia della profondità raggiunta del nodo ho utilizzato una variabile contatore (chiamata **profondità**) che viene incrementata man mano che si scende in profondità, e, decrementata durante il processo di salita. L'altezza dell'**Albero** è determinabile come il  $\lfloor \log_2 \text{Dim}T3 - 1 \rfloor$  nel caso in cui il numero di elementi che compongono il vettore non sia una potenza di 2, in caso contrario bisognerà aggiungere un + 1 a questa quantità.

Analizzando la complessità della funzione di ri-colorazione possiamo notare come questa proceda alla verifica dei nodi anch'essa in maniera ricorsiva, ognuno dei **nodi** viene selezionato una sola volta, mentre la radice non viene mai modificata perchè già di colore nero, possiamo così determinare una complessità pari a  $O(\text{Dim}T3)$  o per meglio dire  $O(\text{Dim}T1 + \text{Dim}T2)$  cioè la somma delle dimensioni dei vettori **T1** e **T2**.

## 1.4 UML - Problema 1

Questo UML descrive le scelte progettuali e le classi che compongono il primo problema del progetto. La classe **Albero** è costituita da un aggregazione di più elementi della classe **Nodo** che vanno effettivamente a comporre la struttura Albero, è composta oltre che dai membri pubblici e privati che siamo soliti trovare, come ad esempio **RotateSx()**, **RotateDx()**, **TreeInsert()** e **FixUp()** contiene anche informazioni e metodi supplementari come ad esempio:

**Scaricato**: Che permette di avere a disposizione un **vector** a disposizione per la rappresentazione dell'albero. Nel progetto viene usato per scaricare l'albero in-Order.

**profondita**: Questo membro tiene traccia durante la visita della profondità a cui si trova un determinato nodo dell'**Albero**.

**altezza**: Rappresenta la profondità massima raggiunta dall'albero.

**Pulisci()**: Permette di svuotare il vettore rappresentativo dell'albero, utile nel caso in cui aggiungendo in un secondo momento altri elementi questo vettore non sarebbe più fedele.

**Scarica()**: Permette di caricare il vettore a partire dagli elementi che compongono l'Albero con una visita in-Order.

**inOrder()**: Permette di effettuare una stampa in-Order dell'**Albero Red-Black**.

**showTrunks()**: Metodo di supporto che permette di stampare i "tronchi" degli Alberi.

**printTree()**: Metodo che permette di stampare un **ASCII Tree**.

**Costruisci()**: Permette di costruire le relazioni tra i **nodi** a partire dal **vector** rappresentativo dell'**Albero Red-Black**.

Non è stato implementato alcun metodo per la cancellazione dei **nodi** in quanto non è stata richiesta alcuna operazione del genere.

La classe **Nodo** è costituita dagli attributi classici dei **nodi Red-Black** come ad esempio gli attributi **Key**, **Parent**, **Sx** (rappresenta il **figlio Sx**), **Dx** (rappresenta il **figlio Dx**) e **Colore**, oltre a tutti i metodi necessari.

La classe **Merge** è costituita dai metodi che permettono di costruire il vettore composto dagli elementi derivati dai due alberi **T1** e **T2**.

La classe **Trunk** viene usata per andare a stampare l'**ASCII Tree** rappresentativo dell'Albero.

E' importante notare che la costruzione dell'**Albero T3** avviene attraverso il costruttore definito all'interno della classe **Albero** a cui viene passato il vettore elaborato dalla classe **Merge**.

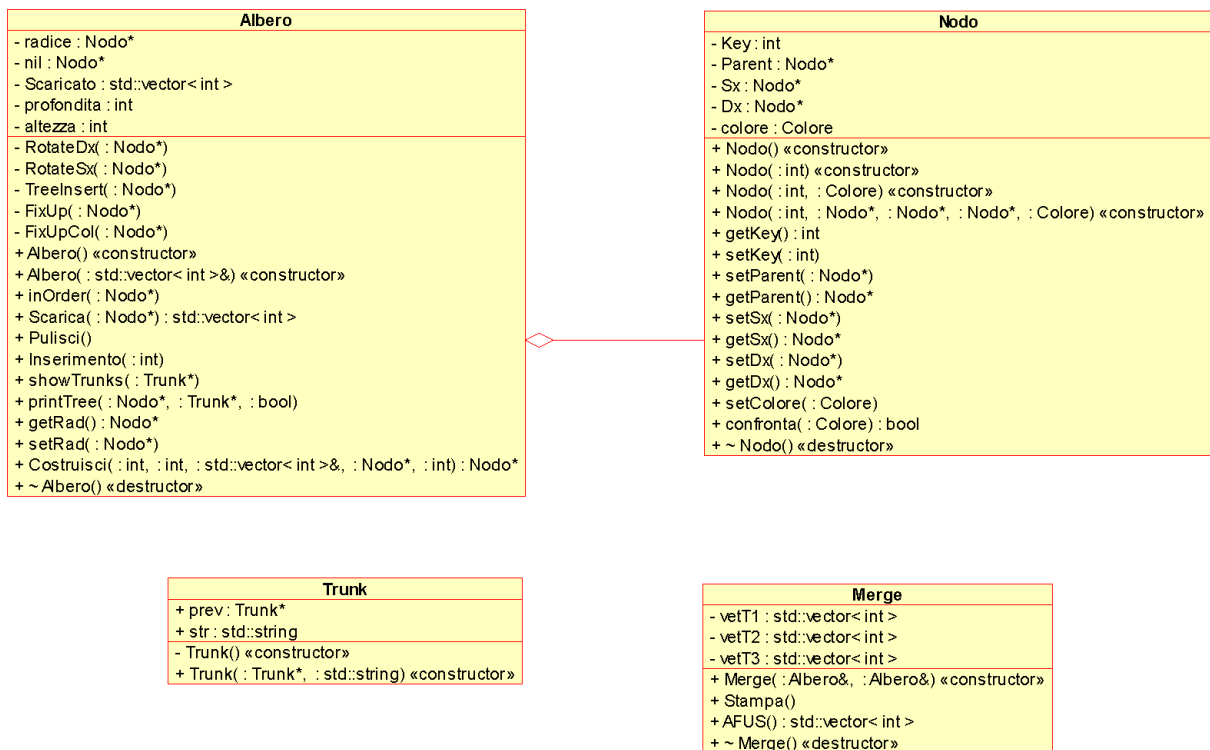


Figura 2: UML problema *Albero Red-Black*



## 1.5 Test e Risultati

Di seguito riportati gli screen dei Test effettuati:

```
Quanti elementi vuoi inserire in T1?
5
Quanti elementi vuoi inserire in T2?
7
Inserire numero per T1: 15
Inserire numero per T1: 2
Inserire numero per T1: 45
Inserire numero per T1: 745
Inserire numero per T1: 1
Inserire numero per T2: 0
Inserire numero per T2: 876
Inserire numero per T2: 45
Inserire numero per T2: 37
Inserire numero per T2: 47
Inserire numero per T2: 65
Inserire numero per T2: 91
```

Figura 3: Input 1

```
Stampo Albero RB T3:
Radice: 45
      .---NIL
      .---0 Nero
      |
      | .---NIL
      | `---1 Rosso
      | `---NIL
      .---2 Rosso
      |
      | .---NIL
      | `---15 Nero
      |   .---NIL
      |   | .---NIL
      |   | `---37 Rosso
      |   | `---NIL
      .---45 Nero
      |
      | .---NIL
      | .---45 Nero
      | | .---NIL
      | | `---47 Rosso
      | | `---NIL
      .---65 Rosso
      |
      | .---NIL
      | .---91 Rosso
      | | .---NIL
      | | `---745 Nero
      | |   .---NIL
      | |   | .---876 Rosso
      | |   | `---NIL
```

Figura 4: Output 1

```

Quanti elementi vuoi inserire in T1?
5
Quanti elementi vuoi inserire in T2?
4
Inserire numero per T1: 9
Inserire numero per T1: 9
Inserire numero per T1: 9
Inserire numero per T1: 9
Inserire numero per T1: 9

Inserire numero per T2: 9
Inserire numero per T2: 9
Inserire numero per T2: 9
Inserire numero per T2: 9

```

Figura 5: Input 2

```

Stampo Albero RB T3:
Radice: 9
      .---NIL
     .---9 Nero
      |  ^---NIL
     .---9 Rosso
      |  .---NIL
      |  ^---9 Nero
      |  |  .---NIL
      |  |  ^---9 Rosso
      |  |  ^---NIL
---9 Nero
   |  .---NIL
   |  .---9 Nero
   |  |  ^---NIL
   |  ^---9 Rosso
   |  |  .---NIL
   |  |  ^---9 Nero
   |  |  |  .---NIL
   |  |  |  ^---9 Rosso
   |  |  |  ^---NIL

```

Figura 6: Output 2

## 1.6 Codice

### 1.6.1 *main.cpp*

---

```
#include <iostream>
#include "Albero.h"
#include "Nodo.h"
#include "Merge.h"
using namespace std;

int main()
{
    ///Dichiariamo i due alberi T1 e T2
    Albero T1,T2;
    int elemT1,elemT2;
    int app;
    ///Chiediamo quanti elementi vogliamo inserire nei due alberi
    cout<<"Quanti elementi vuoi inserire in T1?"<<endl;
    cin>>elemT1;
    cout<<"Quanti elementi vuoi inserire in T2?"<<endl;
    cin>>elemT2;
    ///Effettuiamo il caricamento dei dati in T1 e T2
    for(int i = 0;i < elemT1; i++)
    {
        cout<<"Inserire numero per T1: ";
        cin>>app;
        T1.Inserimento(app);
    }
    cout<<endl;
    for(int i = 0;i < elemT2; i++)
    {
        cout<<"Inserire numero per T2: ";
        cin>>app;
        T2.Inserimento(app);
    }
    ///Istanziamo un oggetto Fondi della classe Merge e passiamo gli alberi T1 e T2
    Merge Fondi (T1,T2);
    ///Ricaviamo in vettore fuso a partire dai due alberi passati precedentemente
    vector<int> VettoreFuso = Fondi.AFUS();
    ///Andiamo a creare un nuovo albero a partire dal vettore fuso
    Albero T3 (VettoreFuso);
    ///Effettuiamone la stampa
    cout<<"Stampo Albero RB T3: "<<endl;
    cout<<"Radice: "<<T3.getRad()->getKey()<<endl;
```

```
T3.printTree(T3.getRad(),nullptr,false);  
return 0;  
}
```

---

### 1.6.2 *Nodo.h*

---

```
#ifndef NODO_H_INCLUDED
#define NODO_H_INCLUDED
typedef enum colore {nero,rosso} Colore;
class Nodo{
private:
    int Key;
    Nodo *Parent;
    Nodo *Sx;
    Nodo *Dx;
    Colore colore;
public:
    Nodo          (void);
    Nodo          (int );
    Nodo          (int ,Colore );
    Nodo          (int ,Nodo *, Nodo *,Nodo *,Colore );
    int getKey    ();
    void setKey    (int );
    void setParent (Nodo *);
    Nodo *getParent ();
    void setSx     (Nodo *);
    Nodo *getSx    ();
    void setDx     (Nodo *);
    Nodo *getDx    ();
    void setColore (Colore );
    bool confronta (Colore );
    ~Nodo()        {};
};

///
```

```

        this->Dx = nullptr;
        this->Key = Key;
    }
    ///Costruttore con Key e Colore
    Nodo::Nodo (int Key,Colore colore)
    {
        this->Parent = nullptr;
        this->Sx = nullptr;
        this->Dx = nullptr;
        this->Key = Key;
        this->colore = colore;
    }
    ///Contruttore completo con tutti gli attributi
    Nodo::Nodo (int Key,Nodo *Parent,Nodo *Sx,Nodo *Dx,Colore colore)
    {
        this->Parent = Parent;
        this->Sx = Sx;
        this->Dx = Dx;
        this->Key = Key;
        this->colore = colore;
    }
    ///Restituisce il valore della Key del Nodo
    int Nodo::getKey()
    {
        return this->Key;
    }
    ///Setta il valore Key del Nodo
    void Nodo::setKey(int Key)
    {
        this->Key = Key;
    }
    ///Setta il Parent del Nodo
    void Nodo::setParent (Nodo *nodo)
    {
        this->Parent = nodo;
    }
    ///Restituisce il Parent del Nodo
    Nodo* Nodo::getParent ()
    {
        return this->Parent;
    }
    ///Restituisce il Figlio Sx
    Nodo* Nodo::getSx ()
    {

```

```

        return this->Sx;
    }
    ///Setta il Figlio Sx
    void Nodo::setSx (Nodo *nodo)
    {
        this->Sx = nodo;
    }
    ///Restituisce il Figlio Dx
    Nodo* Nodo::getDx ()
    {
        return this->Dx;
    }
    ///Setta il Figlio Dx
    void Nodo::setDx (Nodo *nodo)
    {
        this->Dx = nodo;
    }
    ///Setta il Colore del Nodo
    void Nodo::setColore (Colore colore)
    {
        this->colore = colore;
    }
    ///Questo metodo permette, dato in input un colore, di verificare se il Nodo su cui chiamato
    di quel colore o meno
    bool Nodo::confronta (Colore colore)
    {
        bool result ;
        ( this->colore == colore) ? result = true : result = false;
        return result ;
    }

#endif // NODO_H_INCLUDED

```

---

### 1.6.3 *Albero.h*

---

```
#ifndef ALBERO_H_INCLUDED
#define ALBERO_H_INCLUDED
#include "Nodo.h"
#include "Trunk.h"
#include <vector>
#include <math.h>
class Albero{
private:
    Nodo *radice;
    Nodo *nil;
    std::vector<int> Scaricato;
    void RotateDx      (Nodo *);
    void RotateSx      (Nodo *);
    void TreeInsert     (Nodo *);
    void FixUp          (Nodo *);
    void FixColore      (Nodo *);
    int  profondita = 0;
    int  altezza  = 0;
public:
    Albero              (void);
    Albero              (std::vector<int> &);
    void inOrder         (Nodo *);
    std::vector<int> Scarica (Nodo *);
    void Pulisci         ();
    void Inserimento     (int );
    void showTrunks      (Trunk* );
    void printTree       (Nodo *,Trunk* ,bool);
    Nodo *getRad         ();
    void setRad          (Nodo *);
    Nodo *Costruisci     (int , int ,std::vector<int> &,Nodo* ,int);
    ~Albero()            { delete this->radice; delete this->nil; }
};
///Costruttore di default per l'Albero
Albero::Albero ()
{
    this->nil = new Nodo(-1,nullptr,nullptr,nullptr,nero);
    radice = nil;
    nil->setParent(radice);
}
///Costruttore di Albero usato per andare a costruire la struttura a partire dai
///dati presenti all'interno del vettore e lanciare la procedura di correzione
```



```

///dell'informazione sul colore
Albero::Albero(std::vector<int> &vettore)
{
    this->nil = new Nodo();
    float dec = log2(vettore.size());
    ((dec - log2(vettore.size())) == 0) ? this->altezza = floor(log2(vettore.size()-1)) + 1 :
        this->altezza = floor(log2(vettore.size()-1));
    this->setRad(Costruisci(0,vettore.size() - 1,vettore, this->nil,this->altezza));
    std::cout<<std::endl<<std::endl;
    this->FixColore(this->getRad());
}
///Metodo per settare la radice
void Albero::setRad(Nodo *nodo)
{
    radice = nodo;
}
///Metodo di supporto per la stampa dell'ASCII Tree
void Albero::showTrunks(Trunk *p)
{
    if (p == nullptr)
        return;
    showTrunks(p->prev);
    std::cout << p->str;
}
///Metodo di stampa ASCII Tree
void Albero::printTree(Nodo *root,Trunk *prev, bool isLeft)
{
    const char *col;
    if (root == nullptr)
        return;

    std::string prev_str = "";
    Trunk *trunk = new Trunk(prev, prev_str);

    printTree(root->getSx(), trunk, true);

    if (!prev)
        trunk->str = "---";
    else if (isLeft)
    {
        trunk->str = ".---";
        prev_str = " |";
    }
    else

```

```

{
    trunk->str = "‘---’;
    prev->str = prev_str;
}

showTrunks(trunk);
col = (root->confronta(rosso)) ? "Rosso" : "Nero";
(root->getKey() != -1) ? std::cout << root->getKey() << " " << col << std::endl :
    std::cout << "NIL" << std::endl ;
if (prev)
    prev->str = prev_str;
trunk->str = " |";

printTree(root->getDx(), trunk, false);
}
//Metodo per costruire le relazioni tra i nodi dell'Albero
Nodo* Albero::Costruisci( int i, int j, std::vector<int> &vettore, Nodo* predecessore, int
    altezza_max)
{
    if( i > j )
        return this->nil;
    int mediano = ( i + j ) / 2;
    Nodo* nuovo = new Nodo(vettore.at(mediano),nullptr,nullptr,nullptr,nero);
    nuovo->setParent(predecessore);
    nuovo->setSx(Costruisci(i,mediano-1,vettore,nuovo,altezza_max));
    nuovo->setDx(Costruisci(mediano+1,j,vettore,nuovo,altezza_max));
    return nuovo;
}
//Metodo per effettuare la Rotazione Sinistra di un nodo
void Albero::RotateSx(Nodo *nodo)
{
    Nodo *app = nodo->getDx();
    nodo->setDx(app->getSx());
    if (app->getSx() != this->nil)
        app->getSx()->setParent(nodo);
    app->setParent(nodo->getParent());
    if (nodo->getParent() == this->nil)
        this->radice = app;
    else if (nodo == nodo->getParent()->getSx())
        nodo->getParent()->setSx(app);
    else
        nodo->getParent()->setDx(app);
    app->setSx(nodo);
    nodo->setParent(app);
}

```

```

}
///Metodo per effettuare la Rotazione Destra di un nodo
void Albero::RotateDx(Nodo *nodo)
{
    Nodo *app = nodo->getSx();
    nodo->setSx(app->getDx());
    if (app->getDx() != this->nil)
        app->getDx()->setParent(nodo);
    app->setParent(nodo->getParent());
    if (nodo->getParent() == this->nil)
        this->radice = app;
    else if (nodo == nodo->getParent()->getSx())
        nodo->getParent()->setSx(app);
    else
        nodo->getParent()->setDx(app);
    app->setDx(nodo);
    nodo->setParent(app);
}
///Metodo per la stampa in-Order dell'Albero
void Albero::inOrder(Nodo *nodo)
{
    if ( nodo != this->nil )
    {
        inOrder( nodo->getSx() );
        (nodo->confronta(nero)) ? (std::cout << "Nodo: " << nodo->getKey() << " Colore:
        Nero, Padre: " << nodo->getParent()->getKey() << std::endl) : (std::cout <<
        "Nodo: " << nodo->getKey() << " Colore: Rosso, Padre: " <<
        nodo->getParent()->getKey() << std::endl);
        inOrder( nodo->getDx() );
    }
}
///Metodo utile per scaricare all'interno del vettore 'Scaricato' il contenuto dell'albero
std::vector<int> Albero::Scarica(Nodo *nodo)
{
    if ( nodo != this->nil )
    {
        Scarica( nodo->getSx() );
        Scaricato.push_back(nodo->getKey());
        Scarica( nodo->getDx() );
    }
    return Scaricato;
}
///Metodo per il reset del vettore
void Albero::Pulisci ()

```

```

{
    Scaricato.clear();
}
///Metodo di inserimento nodo all'interno dell'albero
void Albero::TreeInsert(Nodo *nodo)
{
    Nodo *y = this->nil;
    Nodo *x = this->radice;
    while(x != nil)
    {
        y = x;
        if(nodo->getKey() < x->getKey())
            x = x->getSx();
        else
            x = x->getDx();
    }
    nodo->setParent(y);
    if (y == this->nil)
        this->radice = nodo;
    else if (nodo->getKey() < y->getKey())
        y->setSx(nodo);
    else
        y->setDx(nodo);
}
///Metodo Fix Up per garantire che le proprieta' degli Alberi Red-Black siano preservate
    effettuando la ricolorazione dei nodi e le rotazioni per bilanciarlo
void Albero::FixUp(Nodo *nodo)
{
    while( (nodo != this->radice) && (nodo->getParent()->confronta(rosso)) )
    {
        if(nodo->getParent() == nodo->getParent()->getParent()->getSx())
        {
            Nodo* app = nodo->getParent()->getParent()->getDx();
            if(app->confronta(rosso))
            {
                nodo->getParent()->setColore(nero);
                app->setColore(nero);
                nodo->getParent()->getParent()->setColore(rosso);
                nodo = nodo->getParent()->getParent();
            }
            else
            {

```

```

        if (nodo == nodo->getParent()->getDx())
        {
            nodo = nodo->getParent();
            this->RotateSx(nodo);
        }
        nodo->getParent()->setColore(nero);
        nodo->getParent()->getParent()->setColore(rosso);
        this->RotateDx(nodo->getParent()->getParent());
    }
}
else
{
    Nodo* app = nodo->getParent()->getParent()->getSx();
    if (app->confronta(rosso))
    {
        nodo->getParent()->setColore(nero);
        app->setColore(nero);
        nodo->getParent()->getParent()->setColore(rosso);
        nodo = nodo->getParent()->getParent();
    }
    else
    {
        if (nodo == nodo->getParent()->getSx())
        {
            nodo = nodo->getParent();
            this->RotateDx(nodo);
        }
        nodo->getParent()->setColore(nero);
        nodo->getParent()->getParent()->setColore(rosso);
        this->RotateSx(nodo->getParent()->getParent());
    }
}
}
this->radice->setColore(nero);
}
///Metodo di inserimento che a partire da una chiave va ad inserire un nodo
///con la stessa chiave all'interno dell'Albero
void Albero::Inserimento(int key)
{
    Nodo* app = new Nodo(key, this->nil, this->nil, this->nil, rosso);
    this->TreeInsert(app);
    this->FixUp(app);
}
///Metodo per la ricolorazione dell'Albero costruito con il vector fuso

```

```

void Albero::FixColore (Nodo *nodo)
{
    ///Effettuiamo la ricolorazione poiche' dopo la fusione dei due alberi le proprieta' RB del
    nuovo albero sono errate essendo tutte le foglie nere
    ///procediamo quindi col colorare i nodi nel modo corretto e mantenendo il padre nero
    ///per i nodi diversi dal padre effettuiamo una verifica semplice, se il nodo, i due figli e
    il Parent
    ///sono neri allora possiamo colorare il nodo in analisi di rosso
    if ( nodo != this->nil )
    {
        if (nodo != this->radice)
        {
            if (nodo->getDx() == this->nil && nodo->getSx() == this->nil )
            {
                if (nodo->getParent()->getSx()->getDx() == this->nil &&
                    nodo->getParent()->getSx()->getSx() == this->nil &&
                    nodo->getParent()->getDx()->getSx() == this->nil &&
                    nodo->getParent()->getDx()->getDx() == this->nil &&
                    this->profondita == this->altezza)
                {
                    nodo->setColore(rosso);
                    nodo->getParent()->setColore(nero);
                }
            }
            else if (nodo->getParent()->getDx() == nodo)
            {
                if (nodo->getParent()->getSx() == this->nil && nodo->getSx() ==
                    this->nil && this->profondita == this->altezza)
                {
                    nodo->setColore(rosso);
                    nodo->getParent()->setColore(nero);
                }
            }
            else if (nodo->getParent()->getDx() == this->nil && nodo->getDx() ==
                this->nil && this->profondita == this->altezza)
            {
                nodo->setColore(rosso);
                nodo->getParent()->setColore(nero);
            }
        }
        else if ((nodo->getDx()->confronta(nero) && nodo->getSx()->confronta(nero))
            && nodo->getParent()->confronta(nero))
            nodo->setColore(rosso);
    }
    this->profondita++;
}

```

```

        FixColore(nodo->getSx());
        this->profondita--;
        this->profondita++;
        FixColore(nodo->getDx());
        this->profondita--;
    }
}
///Metodo che restituisce la radice
Nodo* Albero::getRad()
{
    return this->radice;
}

```

```

#endif // ALBERO_H_INCLUDED

```

---

#### 1.6.4 Merge.h

---

```
#ifndef MERGE_H_INCLUDED
#define MERGE_H_INCLUDED
#include "Albero.h"
class Merge{
private:
    std::vector<int> vetT1,vetT2,vetT3;
public:
    Merge                (Albero &, Albero &);
    void Stampa          ();
    std::vector<int> AFUS ();
    ~Merge()             {};
};
///Costruttore classe Merge
Merge::Merge (Albero &T1,Albero &T2)
{
    T1.Pulisci();
    this->vetT1 = T1.Scarica(T1.getRad());
    T2.Pulisci();
    this->vetT2 = T2.Scarica(T2.getRad());
}
///Permette di stampare il contenuto scaricato dai due vettori
void Merge::Stampa()
{
    std::cout<<"Vettore T1 in Merge: ";
    for(unsigned i=0;i<vetT1.size();i++)
        std::cout<<vetT1.at(i)<<" ";
    std::cout<<"Vettore T2 in Merge: ";
    for(unsigned i=0;i<vetT2.size();i++)
        std::cout<<vetT2.at(i)<<" ";
}
///Esegue il Merge dei due vettori
std::vector<int> Merge::AFUS()
{
    unsigned i = 0,j = 0;
    while( i < vetT1.size() && j < vetT2.size() )
    {
        if(vetT1.at(i) < vetT2.at(j))
            vetT3.push_back(vetT1.at(i++));
        else
            vetT3.push_back(vetT2.at(j++));
    }
}
```



```
    while(i < vetT1.size())  
        vetT3.push_back(vetT1.at(i++));  
    while(j < vetT2.size())  
        vetT3.push_back(vetT2.at(j++));  
    return vetT3;  
}
```

```
#endif // MERGE_H_INCLUDED
```

---

### 1.6.5 *Trunck.h*

---

```
#ifndef TRUNK_H_INCLUDED
#define TRUNK_H_INCLUDED
class Trunk
{
private:
    Trunk    ();
public:
    Trunk    (Trunk *, std::string );
    Trunk *prev;
    std::string str;
};
///Costruttore della classe Trunk
Trunk :: Trunk(Trunk *prev, std::string str)
{
    this->prev = prev;
    this->str = str;
}

#endif // TRUNK_H_INCLUDED
```

---

## 2 Descrizione Problema 2

Nel Problema 2 viene richiesto di risolvere un *Problema di Viabilità* ovvero, dato in input un *Grafo Orientato*, andare a verificare che questo sia *fortemente connesso*. In caso affermativo andremo a scrivere nel file *output.txt* il valore *(0 0)*, altrimenti andremo ad indicare i nodi che *non sono fortemente connessi* tra loro visto che nel Problema viene richiesto di segnalarne *almeno* uno.

Prima di procedere con l'illustrazione del codice procediamo ad introdurre brevemente la struttura dati *Grafo Orientato* e a descriverne le proprietà. Passeremo poi alla descrizione dell'algoritmo che dato in input il file *input.txt* sia in grado di elaborare il file *output.txt* con la soluzione del Problema.

## 2.1 Grafo Orientato

Un **Grafo Orientato**  $G$  è una coppia  $(V, E)$ , dove  $V$  è un insieme finito ed  $E$  è una relazione binaria in  $V$ . L'insieme  $V$  è detto **insieme dei vertici** di  $G$  e i suoi elementi sono detti **vertici**. L'insieme  $E$  è detto **insieme degli archi** di  $G$  e i suoi elementi sono detti **archi**.

In questa implementazione sarà usato un Grafo rappresentato con **liste di adiacenza**.

La **rappresentazione con liste di adiacenza** di un grafo  $G = (V, E)$  consiste in un array  $Adj$  di  $|V|$  liste, una per ogni vertice  $V$ . Per ogni  $u \in V$ , la lista di adiacenza  $Adj[u]$  contiene tutti i vertici  $v$  tali che esiste un arco  $(u, v) \in E$ . Ovvero  $Adj[u]$  include tutti i vertici adiacenti a  $u$  in  $G$ .

Un Grafo Orientato è **fortemente connesso** se due vertici qualsiasi sono raggiungibili l'uno dall'altro. Le **componenti fortemente connesse** di un grafo orientato sono classi di equivalenza dei vertici secondo la relazione "sono mutuamente raggiungibili". Un grafo orientato è fortemente connesso se ha una sola componente fortemente connessa.

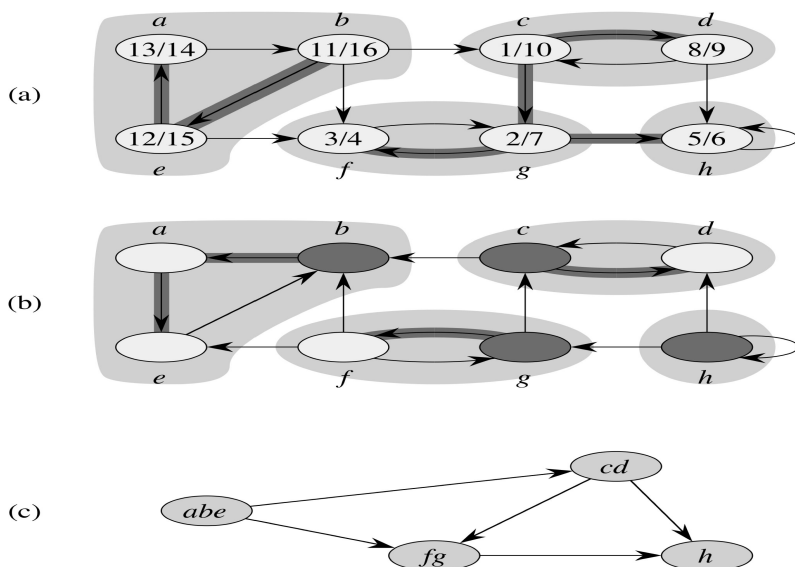


Figura 7: Esempio di un **Grafo fortemente connesso**

## 2.2 Formato Dati I/O

Per questa implementazione è previsto l'utilizzo del file di input ***input.txt*** che contiene nella prima riga due interi separati da uno spazio: il numero  $N$  delle piazze (numerate da 1 a  $N$ ) e il numero  $S$  delle strade che le uniscono. Ciascuno degli  $S$  righi successivi contiene una coppia di interi ( $P1$   $P2$ ) separati da uno spazio, che indica la strada a "senso unico" che unisce la Piazza1 alla Piazza2. Nel file di output ***output.txt*** se la viabilità è completa, viene indicato **0 0**, in caso contrario, vengono segnalate le coppie di interi  $S$  e  $D$  che per indicare che non esistono percorsi per andare dalla piazza  $S$  alla piazza  $D$ .

***Assunzioni fornite:***

$$2 \leq N \leq 1000$$

$$1 \leq S \leq 10000$$

$$1 \leq P1, P2 \leq N$$

## 2.3 Ricerca delle Componenti fortemente connesse

Per andare a ricercare le **componenti fortemente connesse** del Grafo ci avvarremo della visita in profondità (**DFS**) che ci permetterà di scomporre il grafo orientato nelle sue componenti fortemente connesse.

Questo algoritmo permette di trovare le componenti fortemente connesse di un grafo  $G = (V, E)$  utilizzando il grafo trasposto di  $G$ , che è definito come il grafo  $G^t = (V, E^t)$ , dove  $E^t = \{(u, v) : (v, u) \in E\}$ . Ovvero  $E^t$  è formato dagli archi di  $G$  con le direzioni invertite. È interessante osservare come  $G$  e  $G^t$  hanno esattamente le stesse componenti fortemente connesse: i vertici  $u$  e  $v$  sono raggiungibili l'uno dall'altro in  $G$ , se e solo se sono raggiungibili l'uno dall'altro in  $G^t$ . Il seguente algoritmo con tempo lineare  $\Theta(V + E)$  calcola le componenti fortemente connesse di un grafo orientato  $G = (V, E)$  utilizzando due visite in profondità, una su  $G$  e una su  $G^t$ .

Di seguito riportato lo **pseudo codice**:

<pre> DFS(G) L=new stack for each u ∈ V[G]     color[u] ← WHITE     π[u] ← NULL     time ← 0 for each u ∈ V[G]     if color[u] = WHITE         DFS_VISIT1(u, L) create G<sup>T</sup> while L ≠ ∅     v ← L.pop()     if color[v] = WHITE         DFS_VISIT2(v, G<sup>T</sup>) SCC=predecessor-subgraph         </pre>	<pre> DFS_VISIT1(u, L)     color[u] ← GRAY     d[u] ← time ← time + 1     for each v ∈ Adj[u]         if color[v] = WHITE             π[v] = u             DFS_VISIT1(v, L)     color[u] ← BLACK     f[u] ← time ← time + 1     push(L, u)  DFS_VISIT2(u, G<sup>T</sup>)     color[u] ← GRAY     d[u] ← time ← time + 1     for each v ∈ Adj<sup>T</sup>[u]         if color[v] = WHITE             π[v] = u             DFS_VISIT2(v, G<sup>T</sup>)     color[u] ← BLACK     f[u] ← time ← time + 1         </pre>
---	--

Figura 8: Pseudo Codice **Strongly Connected Components**

Una volta lanciato il metodo **DFS** l'algoritmo provvederà ad eseguire i seguenti passi logici come già indicato in precedenza:

1. Chiama DFS( $G$ ) per calcolare i tempi dei completamento  $u.f$  per ciascun vertice  $u$ . Complessità pari a  $O(V + E)$  poichè il grafo è rappresentato mediante liste di adiacenza.
2. Calcola  $G^t$ . Complessità pari a  $O(V + E)$ .
3. Chiama DFS( $G^t$ ), ma nel ciclo principale di DFS considera i vertici in ordine decrescente rispetto ai tempi  $u.f$  (calcolati nella riga 1). Complessità pari a  $O(V + E)$  poichè il grafo è rappresentato mediante liste di adiacenza.
4. Genera l'output dei vertici di ciascun albero della foresta DF che è stata prodotta nella riga 3 come una singola componente fortemente connessa.

## 2.4 UML - Problema 2

Questo UML descrive le scelte progettuali e le classi che compongono il secondo problema del progetto. La classe **Graph** è dipendente dalla classe **Node** in quanto il nostro Grafo possiede un vettore di puntatori ai nodi del Grafo stesso, e contiene al suo interno oltre ai membri e metodi propri di un grafo anche :

**DFS-Visit1 ()**: Esegue una visita **DFS** sul grafo di partenza andando ad inserire in uno **stack** i nodi scoperti man mano durante la visita.

**DFS-Visit2 ()**: Esegue una visita **DFS** sui nodi scaricati dallo **stack** in ordine decrescente sul grafo trasposto.

**getTranspose()**: Permette di ottenere il grafo trasposto a partire dal grafo originale.

**DFS**: Esegue le operazioni necessarie per andare a determinare le **componenti fortemente connesse** nel grafo e andarle a scrivere sia a video sia su **output.txt**.

La classe **Node** è costituita dagli attributi classici che rappresentano un grafo orientato e rappresentato con la classe **liste di adiacenza**.

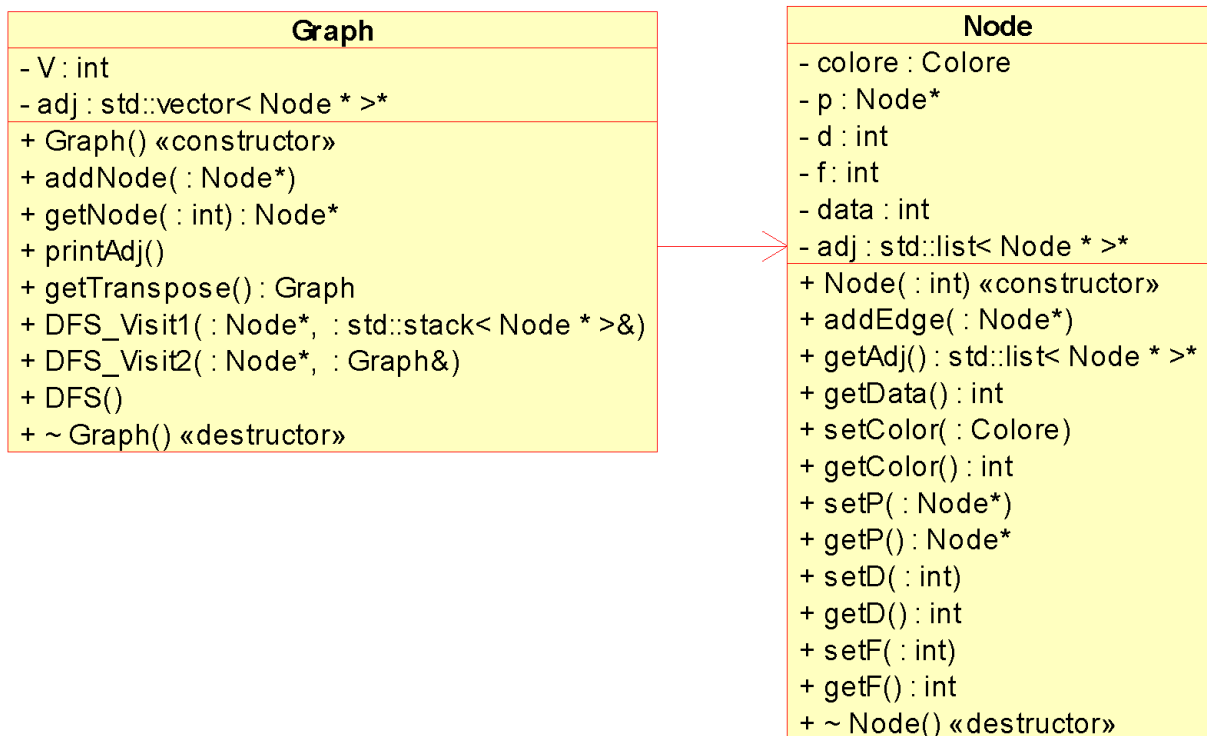
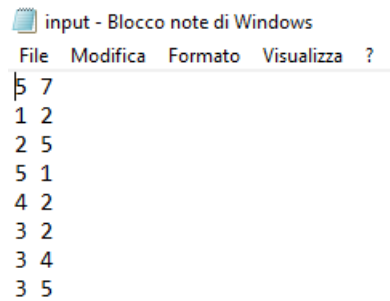


Figura 9: UML *Problema Grafo*

## 2.5 Test e Risultati

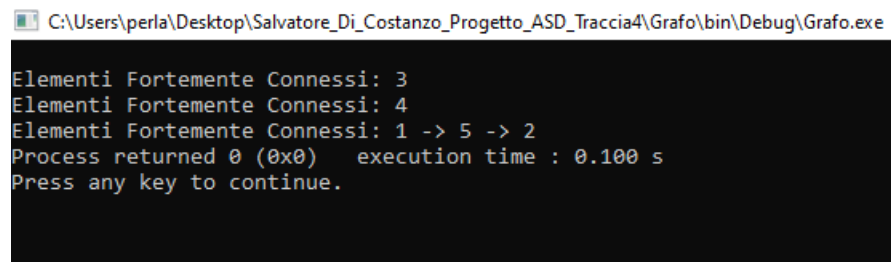
Di seguito riportati gli screen dei Test effettuati:

*Test con Grafo non fortemente connesso:*



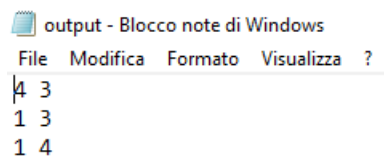
```
input - Blocco note di Windows
File  Modifica  Formato  Visualizza  ?
7
1 2
2 5
5 1
4 2
3 2
3 4
3 5
```

Figura 10: Input 1



```
C:\Users\perla\Desktop\Salvatore_Di_Costanzo_Progetto_ASD_Traccia4\Grafo\bin\Debug\Grafo.exe
Elementi Fortemente Connessi: 3
Elementi Fortemente Connessi: 4
Elementi Fortemente Connessi: 1 -> 5 -> 2
Process returned 0 (0x0)   execution time : 0.100 s
Press any key to continue.
```

Figura 11: Elaborazione 1

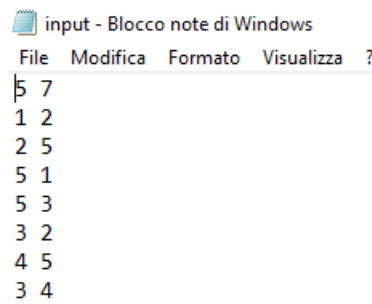


```
output - Blocco note di Windows
File  Modifica  Formato  Visualizza  ?
3
1 3
1 4
```

Figura 12: Output 1

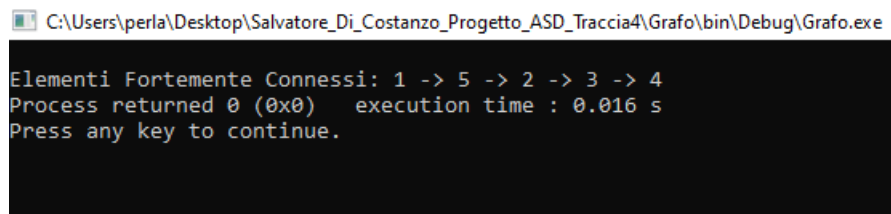


*Test con Grafo fortemente connesso:*



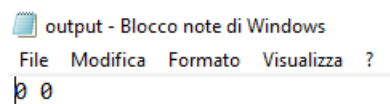
```
input - Blocco note di Windows
File  Modifica  Formato  Visualizza  ?
5 7
1 2
2 5
5 1
5 3
3 2
4 5
3 4
```

Figura 13: Input 2



```
C:\Users\perla\Desktop\Salvatore_Di_Costanzo_Progetto_ASD_Traccia4\Grafo\bin\Debug\Grafo.exe
Elementi Fortemente Connessi: 1 -> 5 -> 2 -> 3 -> 4
Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

Figura 14: Elaborazione 2



```
output - Blocco note di Windows
File  Modifica  Formato  Visualizza  ?
0 0
```

Figura 15: Output 2

## 2.6 Codice

### 2.6.1 *main.cpp*

---

```
#include <iostream>
///Assunzioni
#define Max_Nodi 1000
#define Max_Archi 10000
#include "Grafo.h"
using namespace std;

int main()
{
    ///Istanzio un oggetto 'g' della classe 'Grafo'
    Graph g;
    ///Predispongo due variabili per scaricare le informazioni dal file di input
    int nodi,archi;
    ///Procedo all'apertura del file di input
    std::ifstream File_I("input.txt");
    ///Se fallisce ritorna errore sullo standard error
    if (!File_I.is_open())
    {
        std::cerr << "File non aperto" << std::endl;
        exit(-1);
    }
    ///Altrimenti prendiamo il numero di nodi
    File_I >> nodi;
    ///Controllo se il numero di nodi rientra nel range, in caso negativo esci
    if (nodi > Max_Nodi || nodi < 2)
    {
        std::cout << "Superato Limite Nodi" << std::endl;
        exit(-2);
    }
    ///Altrimenti prendiamo il numero di archi
    File_I >> archi;
    ///Controllo se il numero di archi rientra nel range, in caso negativo esci
    if (archi > Max_Archi || archi < 1)
    {
        std::cout << "Superato Limite Archi" << std::endl;
        exit(-3);
    }
    ///Aggiungo tanti nodi al grafo tanti quanti sono quelli letti dal file + 1 in quanto i
    nostri nodi partono dalla posizione 1 e non 0
    for(int i = 0; i < nodi + 1 ; i++)
```

```

        g.addNode(new Node(i));
    ///Leggo dal file le informazioni dal file e le inserisco nei nodi del grafo
    for(int i = 0; i < archi; i++)
    {
        int piazza;
        File_I >> piazza;
        int adiacenza;
        File_I >> adiacenza;
        g.getNode(piazza)->addEdge(g.getNode(adiacenza));
    }
    ///Chiudo il file
    File_I . close();
    ///Faccio partire la ricerca delle componenti fortemente connesse
    g.DFS();
    return 0;
}

```

---

### 2.6.2 *Nodo.h*

---

```
#ifndef NODO_H_INCLUDED
#define NODO_H_INCLUDED
#include <list>
#include <limits>
typedef enum colore {Bianco,Grigio,Nero} Colore;
class Node{
    Colore colore;
    Node *p;
    int d;
    int f;
    int data;
    std::list<Node*> *adj;
public:
    Node                (int);
    void addEdge        (Node *);
    std::list<Node*> *getAdj ();
    int  getData        ();
    void setColor       (Colore );
    int  getColor       ();
    void setP           (Node *);
    Node *getP          ();
    void setD           (int );
    int  getD           ();
    void setF           (int );
    int  getF           ();
    ~Node()             {delete adj;}
};
///Costruttore nodo
Node::Node (int data)
{
    this->data=data;
    this->colore = Bianco;
    p=nullptr;
    d=std::numeric_limits<int>::max();
    f=std::numeric_limits<int>::max();
    adj=new std::list<Node*>;
}
///Metodo per aggiungere un elemento alla lista di adiacenza del nodo
void Node::addEdge(Node *w)
{
    adj->push_back(w);
}
```

```

}
///Metodo utilizzato per restituire la lista di adiacenza del nodo
std::list<Node*> * Node::getAdj()
{
    return adj;
}
///Metodo utilizzato per restituire il campo data del nodo
int Node::getData()
{
    return this->data;
}
///Metodo utilizzato per settare il campo colore del nodo
void Node::setColor(Colore colore)
{
    this->colore = colore;
}
///Metodo utilizzato per restituire il colore del nodo
int Node::getColor()
{
    return this->colore;
}
///Metodo utilizzato per settare il parent del nodo
void Node::setP (Node *p)
{
    this->p = p;
}
///Metodo utilizzato per restituire il parent del nodo
Node * Node::getP()
{
    return this->p;
}
///Metodo utilizzato per settare il campo d
void Node::setD(int d)
{
    this->d = d;
}
///Metodo utilizzato per restituire il campo d
int Node::getD()
{
    return this->d;
}
///Metodo utilizzato per settare il campo f
void Node::setF(int f)
{

```

```
    this->f = f;
}
///Metodo utilizzato per restituire il campo f
int Node::getF()
{
    return this->f;
}

#endif // NODO_H_INCLUDED
```

---

### 2.6.3 *Grafo.h*

---

```
#ifndef GRAFO_H_INCLUDED
#define GRAFO_H_INCLUDED
#include <fstream>
#include <stack>
#include <queue>
#include <vector>
#include <limits>
#include "Nodo.h"
class Graph
{
private:
    int V;
    std::vector<Node*> *adj;
public:
    Graph                ();
    void addNode          (Node *);
    Node *getNode         (int );
    void printAdj         ();
    Graph getTranspose    ();
    void DFS_Visit1       (Node *,std::stack<Node*> &);
    void DFS_Visit2       (Node *,Graph &);
    void DFS              ();
    ~Graph()              {delete adj;};
};
///Costruttore grafo
Graph::Graph ()
{
    V=0;
    this->adj = new std::vector<Node*>;
}
///Metodo per aggiungere in nodo
void Graph::addNode (Node *s)
{
    this->adj->push_back(s);
    this->V++;
}
///Metodo per ottenere un node della lista in posizione v
Node* Graph::getNode(int v)
{
    return this->adj->at(v);
}
```

```

//Metodo per la stampa della lista di adiacenza
void Graph::printAdj(){
    for(auto v:*adj){
        std::cout<<" Adj("<<v->getData()<<"):";
        for(auto vv:*v->getAdj())
            std::cout<<" "<<vv->getData();
        std::cout<<std::endl;
    }
}

//DFS per la visita sul grafo di partenza
void Graph::DFS_Visit1(Node *s,std::stack<Node *> &L)
{
    s->setColor(Grigio);
    s->setD(s->getD()+1);
    for(auto v : *s->getAdj())
    {
        if (v->getColor() == 0)
        {
            v->setP(s);
            DFS_Visit1(v,L);
        }
    }
    s->setColor(Nero);
    s->setF(s->getD()+1);
    L.push(s);
}

//DFS per la visita sul grafo trasposto
void Graph::DFS_Visit2(Node *s,Graph &g)
{
    s->setColor(Grigio);
    s->setD(s->getD()+1);
    int index = 0;
    for(int i = 0;i < V ;i++)
    {
        if(g.getNode(i)->getData() == s->getData())
        {
            index = i;
            i = V;
        }
    }
    for(auto x : *g.adj->at(index)->getAdj())
    {
        if (x->getColor() == Bianco)
        {

```



```

        std::cout<<" -> "<<x->getData();
        x->setP(s);
        DFS_Visit2(x,g);
    }
}
s->setColor(Nero);
s->setF(s->getD()+1);
}

///Metodo per ottenere il grafo trasposto
Graph Graph::getTranspose()
{
    Graph g ;
    for(int i = 0; i<V ;i++)
        g.addNode(new Node(this->getNode(i)->getData()));

    for(auto x : *adj)
        for(auto y : *x->getAdj())
        {
            for(int i = 0;i < V ;i++)
            {
                if(adj->at(i) == y)
                {
                    g.getNode(i)->addEdge(x);
                    i = V;
                }
            }
        }
    return g;
}

```

```

///Metodo per la ricerca delle componenti fortemente connesse
void Graph::DFS ()
{
    ///Dichiaro lo stack
    std::stack<Node *> L;
    ///Coloro tutti i nodi di bianco
    for(unsigned i = 0; i < adj->size(); i++)
    {
        adj->at(i)->setColor(Bianco);
        adj->at(i)->setP(nullptr);
        adj->at(i)->setD(Bianco);
    }
}

```

```

///Eseguo una visita DFS passando anche lo stack per riempirlo
for(unsigned i = 0; i < adj->size(); i++)
    if (adj->at(i)->getColor() == 0)
        DFS_Visit1(adj->at(i),L);

///Calcolo il trasposto
Graph g = getTranspose();
///Dichiaro il grafo SCC
Graph SCC;
///Settiamo i nodi del grafo di partenza di colore Bianco
for(unsigned i = 0; i < adj->size(); i++)
    adj->at(i)->setColor(Bianco);
///Dichiaro un vettore per collezionare gli elementi non collegati
std::vector<int> Non_Coll;
///Tengo un contatore per contare le compotenti fortemente connesse
int control_elem = 0;
///Avvia la procedura per la ricerca degli elementi fortemente connessi
while (!L.empty())
{
    ///Scarica l'elemento testa dallo stack
    Node * v = L.top();
    ///Rimuovi la testa dallo stack
    L.pop();
    ///Procedi a verificare il nodo
    if (v->getColor() == 0)
    {
        if (v->getData() != 0)
        {
            std::cout<<std::endl<<"Elementi Fortemente Connessi: "<<v->getData() ;
            Non_Coll.push_back(v->getData());
            control_elem++;
        }
        DFS_Visit2(v,g);
    }
    ///Aggiungiamo il nodo che fortemente connesso al grafo SCC
    SCC.addNode(v);
}
///Apriamo il file di output
std::ofstream File_O("output.txt");
if (!File_O.is_open())
{
    std::cerr << "File Output non aperto" << std::endl;
    exit(-1);
}

```

```

    ///Se esistono piu' elementi fortemente connessi allora stampali sul file
    if(control_elem != 1)
    {
        for(unsigned i = 0;i < Non_Coll.size() - 1 ;i++)
            for(unsigned j = i+1;j < Non_Coll.size() ;j++)
                File_O<<Non_Coll.at(j)<<" " <<Non_Coll.at(i)<<std::endl;
        File_O.close();
    } else ///Altrimenti setta il valore come 0 0
        File_O<<0<<" " <<0<<std::endl;

    File_O.close();
}

#endif // GRAFO_H_INCLUDED

```

---