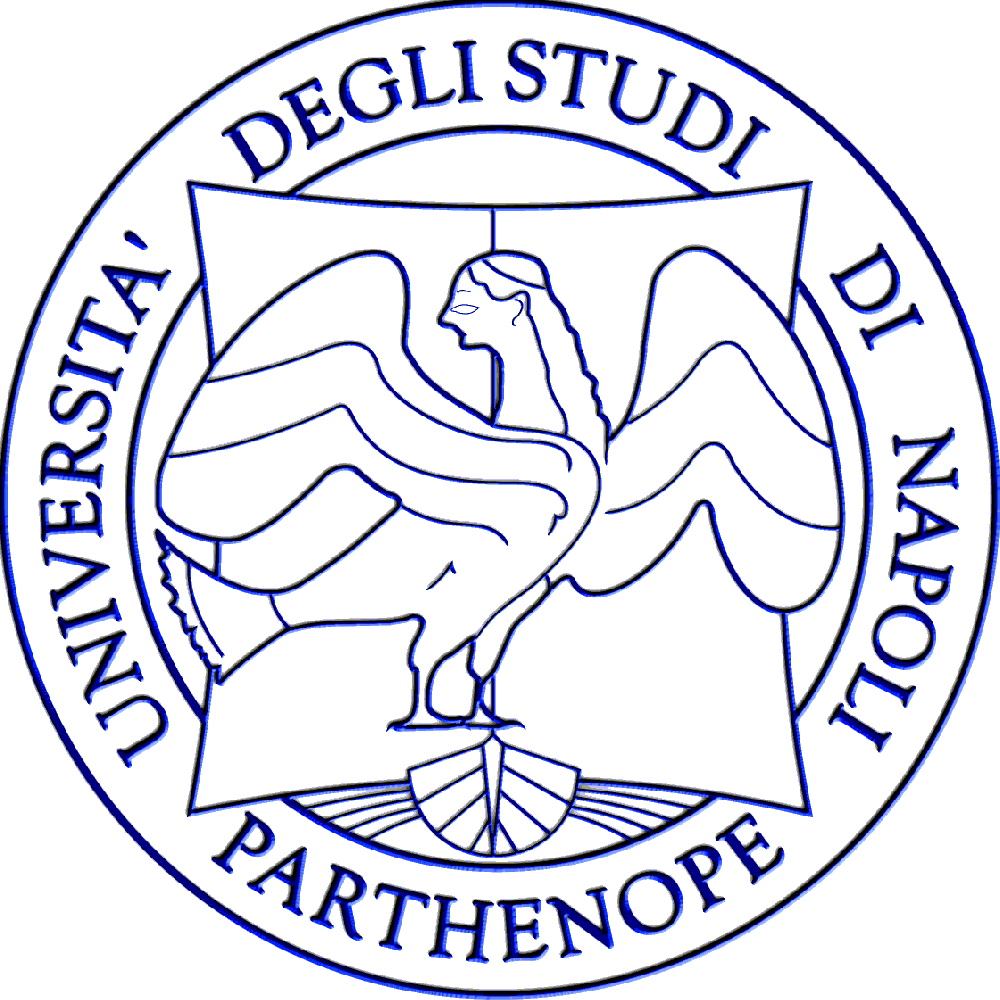
****

**Progetto Algoritmi e Strutture Dati e Laboratorio di Algoritmi e Strutture Dati**

**Gianfranco Terrazzano**

**0124002052**

**Sommario**

[Traccia 1 3](#_Toc120736877)

[Descrizione del problema 3](#_Toc120736878)

[Descrizione strutture dati 4](#_Toc120736879)

[Formato dati input/output 7](#_Toc120736880)

[Descrizione algoritmo 8](#_Toc120736881)

[Class Diagram 16](#_Toc120736882)

[Studio Complessità 16](#_Toc120736883)

[Test e Risultati 18](#_Toc120736884)

[Traccia 2 22](#_Toc120736885)

[Descrizione del problema 22](#_Toc120736886)

[Descrizione strutture dati 22](#_Toc120736887)

[Formato dati input/output 24](#_Toc120736888)

[Descrizione algoritmo 24](#_Toc120736889)

[Class Diagram 29](#_Toc120736890)

[Studio Complessità 30](#_Toc120736891)

[Test e Risultati 31](#_Toc120736892)

# Traccia 1

## Descrizione del problema

Il quesito numero uno chiede di realizzare la struttura dati RBGRaph, ovvero, una struttura basata su un grafo orientato in cui la lista di adiacenza di ogni nodo è rappresentata da un albero RB.

Quindi bisogna realizzare le seguenti funzioni su questa struttura dati: AddEdge(i,j), RemoveEdge(i,j), FindEdge(i,j) e BFS(s).

È stato fornito un file di input all’interno del quale sono stati forniti i nodi e gli archi di ingresso atti a formare il grafo, inoltre, è espressamente chiesto di realizzare un menù dove richiamare le funzioni sopracitate.

Per chiarezza ricordiamo che il **BFS-breadth first search**, ovvero la visita in ampiezza, è un algoritmo di visita dei grafi che funziona nel seguente modo:

Dato un grafo G=(V,E) e un vertice sorgente s, la visita in ampiezza ispeziona sistematicamente gli archi di G per scoprire tutti i vertici che sono raggiungibili da S

* Calcola la distanza (ossia il minimo numero degli archi) da s a ciascun vertice raggiungibile
* Genera anche un albero BF con radice s che contiene tutti i vertici raggiungibili
* Per ogni vertice v raggiungibile da s, il cammino nell’albero BF corrisponde a un cammino minimo da s a v in G
* L’algoritmo opera sia sui grafi orientati che non orientati
* Scopre tutti i vertici che si trovano a distanza k dalla sorgente s prima di quelli a distanza k+1
* L’algoritmo colora ogni vertice di bianco, di grigio o di nero
* All’inizio tutti i vertici sono colorati di bianco, un vertice viene scoperto quando viene incontrato per la prima volta e cessa di essere bianco
* Un vertice è grigio se tra i suoi adiacenti ci sono vertici bianchi (vertici non ancora scoperti)
* Un vertice è nero se tutti i vertici adiacenti sono stati scoperti
* Utilizza una struttura dati CODA che attua una politica FIFO

## Descrizione strutture dati

Per realizzare questo progetto sono state utilizzate principalmente tre strutture dati:

* **Grafo orientato**
* **Albero RB**
* **BST** (necessario all’implementazione di un albero RB)

Un **Grafo** G=(V,E) diretto o orientato:

* Insieme V di nodi (o vertici)
* insieme E di archi
* Un arco è una coppia ordinata (u, v) di nodi

La rappresentazione di un grafo può essere realizzata mediante:

* **lista di adiacenza:** ogni nodo è descritto da una lista di elementi contenenti il nome dei nodi ad esso adiacenti, in questo caso sostituita da un **albero RB.**
* **Matrice di adiacenza:** A = A(aij). Se aij =1 allora il nodo i è connesso a nodo j ,altrimenti se, aij =0 allora il nodo i non è connesso a nodo j. In un grafo non orientato la matrice è simmetrica.

**N.B** Se u, v ∈ V sono collegati da un arco (u, v) ∈ E, si dice che u è **adiacente** a v.

Un albero **Red-Black** è un particolare tipo di albero binario di ricerca **self-balancing**. Le operazioni di ricerca, inserimento e cancellazione hanno complessità O(log n), dove n è il numero degli elementi dell’albero Red-Black. Deve soddisfare i seguenti vincoli:

1. Ogni nodo è colorato di rosso o di nero
2. La radice è nera
3. I nodi NIL (foglie) sono neri
4. Se un nodo è rosso, allora i suoi figli sono neri
5. Ogni percorso da un nodo interno ad una foglia ha lo stesso numero di nodi neri (b-altezza)

Ogni nodo ha:

* Puntatore al padre
* Puntatore al figlio destro
* Puntatore al figlio sinistro
* Chiave
* Colore
* Dati satelliti

Le operazioni SEARCH, MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR definite sugli ABR funzionano anche sugli alberi RB e sono eseguite in tempo O(h) = O(log n).

Gli algoritmi TREE-INSERT e TREE-DELETE devono essere modificati in modo da garantire che l'albero risultante sia ancora un albero RB.

Un **Binary-Search-Tree(BST),** in italiano **Albero binario di ricerca**, è una struttura dati sulla quale vengono definite le seguenti operazioni: SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE. Può essere rappresentato come una struttura dati concatenata di nodi, ed ogni nodo di un **ABR** contiene:

* la chiave (o key);
* il puntatore al PADRE
* il puntatore al FIGLIO DESTRO
* il puntatore al FIGLIO SINISTRO
* Eventuali dati satelliti

In un ABR le chiavi sono sempre memorizzate in modo che sia sempre soddisfatta la proprietà dell’albero binario di ricerca, ovvero, sia **x** un nodo di un albero binario di ricerca:

* Se **y** è un nodo del sottoalbero sinistro di x, allora **key[x] ≥ key[y]**
* Se **y** è un nodo del sottoalbero destro di x, allora **key[x] ≤ key[y]**

A questo punto, descritte in maniera generale le strutture dati, analizziamo come esse sono state organizzate, e come differiscono da un implementazione “classica” in modo da risolvere il problema posto nel quesito uno.

Nell’implementazione dei nodi del grafo(**GraphNode.hpp**), particolarità che si può notare è che piuttosto che utilizzare un tipo list per l’implementazione della lista d’adiacenza, come richiesto dalla traccia, dichiariamo oggetto della classe **RBTree** (RBTree<Node<T> \*> \*TreeRBList – Riga 18), successivamente questo viene inizializzato nel costruttore del nodo, simulando il comportamento di una normale lista d’adiacenza ma con un albero Red and Black. Verrà descritto nella descrizione dell’algoritmo come questa modifica sarà utilizzata.

Per quanto riguarda il grafo “**Grafo”**, nella clausola **private**  avremo:

* Vettore dei nodi e vettore(Vector of pair) degli archi per poterli memorizzare
* Il metodo getNodes che ci ritornerà i nodi del grafo

Mentre in **public** avremo:

* Costruttore
* Le operazioni richieste dalla traccia
* Metodi getter e setter
* Funzioni di conversione Indice-Nodo, Nodo-Indice
* Funzioni ausiliare per il funzionamento del programma

Nella classe dei **Nodi del Grafo** avremo, come **private:**

* Puntatore al padre
* Colore
* Data
* Distanza dalla sorgente
* La RBList

Mentre in **public** avremo:

* Costruttore
* Metodi getter e setter
* Metodi di rimozione e inserimento dalla lista di adiacenza

Nell’albero **Red & Black** , come **private** avremo:

* Left e Right Rotate
* La TransplantRB(leggermente differente da quella dell’ABR)
* Un puntatore a nilT

In **public**:

* Il costruttore che fa riferimento al costruttore del BST essendo RBTree una sottoclasse di BinarySearchTree
* Metodi di insert, delete e deleteFixup, per gli altri è possibile utilizzare quelli della classe BinarySearchTree

Nella classe dei nodi dell’albero avremo come **private**:

* Key
* Puntatore al padre
* Puntatore al figlio sx
* Puntatore al figlio dx
* Colore
* Data

In **public**:

* Il costruttore vuoto e quello parametrizzato
* Metodi getter e setter
* Costruttore virtuale con i delete dei puntatori

Ultima struttura da analizzare è il **BST**, in **private** avremo:

* Puntatore alla radice
* La Transplant

In **public**:

* Il costruttore
* Un metodo per distruggere ricorsivamente l’albero richiamato nel costruttore virtuale
* Metodi getter e setter
* Operazioni di: SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
* Visite In order, post order, pre order
* Due metodi VectorRB e InorderRB che ci torneranno utili per la realizzazione del BFS

**N.B**  Questa è una panoramica generale alle strutture, tutto ciò che è stato realizzato verrà approfondito nella fase di Descrizione dell’algoritmo e di Class Diagram

## Formato dati input/output

I dati vengono forniti in input da un file presente nella documentazione del progetto.

Il file di input contiene nel primo rigo due numeri interi, 0<=N<=1000 e 0<=M<=1000, separati da uno spazio che rappresentano rispettivamente il numero di nodi ed il numero di archi. I successivi M righi contengono due numeri interi separati da uno spazio che rappresentano il nodo sorgente ed il nodo destinazione.

Pertanto, nel programma sono scritti i check opportuni in modo da rispettare le indicazioni della traccia.

L’input viene gestito dalla funzione Add\_Edge che tramite manipolazioni sul file prende in ingresso tutti i dati e popola le strutture dati.

L’output invece, viene mostrato nel terminale in base alla scelta selezionata dal menu. Quest’ultimo è gestito come uno switch case dove sarà possibile, tramite dei cin, selezionare la funzione da eseguire.

## Descrizione algoritmo

L’algoritmo prende i dati di input da file tramite la funzione **GetInput,** essa è gestita tramite stream di file; quindi, viene aperto il file tramite la open e successivamente con la getline leggo la prima riga nella quale ci sarà il numero dei nodi e degli archi. La tecnica utilizza è quella di ricavare un separatore tramite la **find(),** ovvero lo spazio, come specificato nella traccia, e quindi costruire due sottostringhe, prima e dopo il separatore così da ricavare i due numeri, successivamente con la **stoi()** vengono convertiti ad interi e quindi resi utilizzabili. In questo modo ho estrapolato anche i nodi del file, inoltre questi sono stati passati alle funzioni **addNode** per aggiungere il nodo al grafo, codificato come un vector ed anche ad **AddEdge** per creare l’arco tra i due nodi, gli archi sono stati codificati come un vector di pair.

Quindi una volta descritto come vengono presi gli input passiamo ad analizzare le funzioni richieste dalla traccia:

**AddEdge(i,j)**

template <class T> void Graph<T>::AddEdge(unsigned int u, unsigned int v){ //Aggiunge un arco

    if ((getNodeAtIndex(u)!=nullptr) && (getNodeAtIndex(v)!=nullptr)){

        if (!FindEdge(u,v)){

            edges.push\_back({ u, v });

            cout << "L'arco è stato inserito correttamente"<< endl;

            getNodeAtIndex(u)->addEdge(v ,this->getNodeAtIndex(v)); //aggiorna la lista di adiacenza(Albero RB) di u

        }else cout<<"L'arco già esiste!"<<endl;

    }else cout<<"Il nodo non esiste!"<<endl;

La funzione AddEdge funziona nelle seguente maniera:

Vengono passati gli indici dei nodi di cui vogliamo creare l’arco, dopodiché si effettuano alcuni check, ovvero, controlliamo se all’indice passato corrisponda effettivamente un nodo con la funzione **getNodeAtIndex** la quale in caso negativo ci restituisce un puntatore a Null. Successivamente se l’ipotesi è vera controlliamo che **FindEdge()** restituisca false, ovvero che quell’arco non esista. Se questi check vanno a buon fine, posso fare il Push back dell’arco nel vettore **edges** e aggiungere v alla lista di adiacenza di u tramite la funzione **addEdge**, trattandosi di un albero RB questa richiamerà l’insert dell’albero RB.

template <class T> void Node<T>::addEdge(int key, Node <T> \*newEdge){

    this->TreeRBList->insertNodeRB(key,newEdge); //richiamo l'insert dell'albero RB per aggiungere alla lista di adiacenza

}

Quindi si procede alla codifica dell’inserimento

Invariante del ciclo:

* Il nodo x è rosso
* Se p[x] è la radice, allora p[x] è nero
* Le violazioni posso riguardare o la proprietà 2 (x è la radice ed è rosso) o la proprietà 4 (x e p[x] sono entrambi rossi)

Ad ogni iterazione:

* Il puntatore x si sposta verso l'alto nell'albero
* Si effettuano delle rotazioni ed il ciclo termina

Tre casi

1. Lo zio di x è rosso
2. Lo zio di x è nero e x è figlio destro
3. Lo zio di x è nero e x è figlio sinistro

* **Lo zio di x è rosso**: Il padre (p[x]) e lo zio (y=right[p[p[x]]]) di x sono entrambi rossi. Il nonno di x (p[p[x]]) è nero (per l'invariante di ciclo) per cui coloriamo di nero p[x] e y –coloriamo di rosso p[p[x]]. si ripete il ciclo con x=p[p[x]]
* **Lo zio di x è nero e x è figlio destro**: Effettuiamo una rotazione sinistra per trasformare il caso 2 nel caso 3
* **Lo zio di x è nero e x è figlio sinistro:** Aggiustiamo i colori ,Il padre di x (p[x]) diventa nero, Il nonno di x (p[p[x]]) diventa rosso. Eseguiamo una rotazione destra ed Il ciclo non viene rieseguito poiché il padre di x è nero

**RemoveEdge(i,j)**

template <class T> void Graph<T>::RemoveEdge(unsigned int u, unsigned int v){ //rimuove un arco

    if ((getNodeAtIndex(u)!=nullptr) && (getNodeAtIndex(v)!=nullptr)){

        if (FindEdge(u,v)){

            int tmp= GetEdgeID(u,v);

            edges.erase(edges.begin()+tmp-1);

            getNodeAtIndex(u)->Remove\_edge(v); //rimuove v dalla lista d'adiacenza di u

            cout << "L'arco è stato eliminato"<< endl;

        }else cout<<"L'arco non può essere eliminato perché non esiste!"<<endl;

    }else cout<<"L'arco non esiste!"<<endl;

}

La funzione RemoveEdge funziona nella seguente maniera:

Gli vengono passati gli indici degli archi da eliminare, dopodiché si effettuano gli stessi check descritti in AddEdge con la differenza che questa volta FindEdge deve restituire valore true, ovvero se l’arco esiste è possibile eliminarlo. Quindi dichiariamo una variabile tmp all’interno della quale salviamo l’indice dell’arco e tramite l’erase, con un opportuno spiazzamento, eliminiamo l’arco dal vettore edges. Successivamente tramite la funzione **Remove\_edge** rimuoviamo v dalla lista d’adiacenza di u, ovvero dall’albero RB.

template <class T> void Node<T>::Remove\_edge(int key){

    this->TreeRBList->RB\_delete(key); //richiamo il delete dell'albero RB per rimuovere dalla lista di adiacenza

}

La procedura **RB-DELETE** presenta delle istruzioni in più rispetto a TREE-DELETE che servono a tenere traccia del colore di un particolare nodo (y) che potrebbe causare delle violazioni delle proprietà degli alberi RB.

Quando vogliamo cancellare un nodo z:

* Se z ha meno di due figli, y punta a z (che viene rimosso)
* Se z ha due figli, y punta al successore di z e sostituirà z

In entrambi i casi dobbiamo ricordare il colore di y prima di rimuoverlo o spostarlo e dobbiamo tenere traccia del nodo x che prende il posto di y poiché potrebbe causare delle violazioni.

Dopo aver cancellato z, se il nodo y era nero allora è necessario richiamare la procedura **RB-DELETE-FIXUP** per ripristinare le proprietà degli alberi cambiando i colori di alcuni nodi ed effettuando delle rotazioni.

L'obiettivo della **DELETE-FIXUP** è di spostare verso l'alto il nero "extra" finché :

1. x punta ad un nodo rosso e nero, nel qual caso x viene colorato di nero (singolo)
2. x punta alla radice, nel qual caso il nero extra può essere rimosso
3. vengono eseguite rotazioni e colorazioni

All'interno del while x punta sempre ad un nodo doppiamente nero che non è la radice. Determiniamo se x è figlio sinistro o destro (casi simmetrici). Manteniamo un puntatore w al fratello di x.

Si possono verificare quattro casi

1. **Il fratello w di x è rosso quindi deve avere entrambi i figli neri**: Possiamo scambiare i colori di w e p[x] e poi effettuare una rotazione sinistra di p[x]. Il nuovo fratello di x che era uno dei figli di w è nero per cui ricadiamo nel caso 2, 3 o 4
2. **Il fratello w di x è nero ed entrambi i suoi figli sono neri:** Togliamo un nero sia da w che da x ottenendo x singolarmente nero e w rosso e lo aggiungiamo a p[x] (che diventerà nero-nero o rosso-nero). Valutiamo la condizione del ciclo con x=p[x] -> Se x è rosso-nero il ciclo non viene ripetuto ed il nero extra viene rimosso, mentre, se x è nero-nero il ciclo viene ripetuto
3. **Il fratello w di x è nero ed il figlio sinistro di w è rosso ed il figlio destro di w è nero:** Scambiamo i colori di w e left[w] e poi effettuiamo una rotazione destra di w. Il nuovo fratello w di x è nero con un figlio rosso per cui ricadiamo nel caso 4
4. **Il fratello w di x è nero ed il figlio destro di w è rosso**: Trasferiamo a w il colore del padre che successivamente diventa nero così come il figlio destro di w. Rimuoviamo il nero extra da x

N.B: Esistono anche i 4 casi simmetrici (con destro e sinistro scambiati) quando x è figlio destro.

**FindEdge(i,j)**

template <class T> bool Graph<T>::FindEdge(unsigned int u, unsigned int v){ //Cerca un arco nel vettore edge

    for ( const auto& edge: edges){

        if (edge.first == u && edge.second == v){

            cout <<"Arco trovato"<<endl;

            return true;

        }

    }

    cout<<"Arco non trovato"<<endl;

    return false;

}

La FindEdge è la terza funzione richiesta nella traccia, è stata strutturata nel seguente modo:

Vengono passati gli indici dell’arco in questione, tramite un iteratore scorriamo tutto il vettore degli edges che è dichiarato nella seguente maniera;

std::vector<std::pair<unsigned int, unsigned int>> edges;

Quindi controlliamo che il first sia uguale all’indice del nodo sorgente e che il second sia uguale all’indice del nodo destinazione. Se l’esito è positivo ritorniamo true, altrimenti false.

**BFS(s)**

Template <class T> void Graph<T>::BFS(Node<T> \* root) {

    resetNodesProperties();

    root->setParent(nullptr);

    root->setColor(GRAY);

    root->set\_dTime(0);

    queue<Node<T> \*> queue;

    queue.push(root);

    cout << "\n"<< "Nodi raggiungibili con visita BFS:" << "\n" << endl;

     while (!queue.empty()) {

        Node<T> \*u = queue.front();

        queue.pop();

        for (auto & adjacentNodeToU: \*(u->getRBList()->VectorRB())){  //BFS utilizzando L'RB Tree

            if (adjacentNodeToU->getColor() == WHITE){

                adjacentNodeToU->setColor(GRAY);

                adjacentNodeToU->setParent(u);

                adjacentNodeToU->set\_dTime(u->get\_dTime()+1);

                queue.push(adjacentNodeToU);

            }

        }

         u->setColor(BLACK);

         cout<< "Nodo: "<< u->getData() <<"  Distanza dalla sorgente: "<< u->get\_dTime()<<endl;

    }

Per l’implementazione del BFS mi sono basato ovviamente sull’implementazione classica, quindi stabilito un nodo sorgente, vogliamo controllare distanza dei nodi raggiungibili da esso.

Quindi rispettiamo tutte le condizioni precedentemente anticipate nella [descrizione del problema](#Uno) quando parliamo del BFS. Particolarità è che la lista di adiacenza, di solito struttura fondamentale di questo algoritmo è un albero RB;

RBTree<Node<T> \*> \*TreeRBList;

Quindi per poter scorrere in maniera agevole i nodi della lista RB, una volta richiamato il get sulla lista, utilizziamo la funzione **VectorRB**, definita nel BST superclasse dell’albero RB.

Template <class T>  vector<T> \* BinarySearchTree<T>::InorderRB(NodeRB<T> \* x, vector<T> \* Vector) {

    if (x != nullptr) {

        InorderRB(x->getLeft(), Vector);

        Vector->push\_back(x->getData());

        InorderRB(x->getRight(), Vector);

    }

    return Vector;

}

template <class T> vector<T> \* BinarySearchTree<T>::VectorRB() {

    vector<T> \* VectorRB= new vector<T>;

    InorderRB(this->getRoot(), VectorRB);

    return VectorRB;

}

In pratica con una visita in order, visitiamo tutti i nodi dell’albero e li inseriamo in un vettore, VectorRB serve a richiamare questa funzione e ritorna il vettore sul quale effettueremo il BFS.

**Menù**

Ultima richiesta della traccia era di dotare il programma di un menù dal quale richiamare le varie funzioni.

Esso è stato strutturato come uno switch case

    //constants for menu choices

     const int NodeList=1,

               AddEdge = 2,

              RemoveEdge = 3,

              FindEdge = 4,

              ShowEdgeList = 5,

              BFS= 6,

              EXIT= 7;

    do

    {

         showWelcome(); // Show Welcome screen

         showMenu(); // Display Menu

         cin >> choice;

         //Validate menu selection

         while ((choice != NodeList) && (choice !=AddEdge) && (choice != RemoveEdge) && (choice!=FindEdge) && (choice!=ShowEdgeList) && (choice !=BFS) && (choice != EXIT))

         {

               cin.clear();

               cin.ignore(numeric\_limits<streamsize>::max(), '\n');

               cout << "Per favore inserisci un opzione valida\n";

               cin >> choice;

         }

         //If user does not want to quit, proceed.

         if (choice != EXIT)

         {

                    switch (choice)

                    {

                         case NodeList:

                              simpleGraph.printNode();

                              break;

                           case AddEdge:

                                Add\_Edge(simpleGraph);

                                break;

                           case RemoveEdge:

                                Remove\_Edge(simpleGraph);

                                break;

                           case FindEdge:

                                Find\_edge(simpleGraph);

                                break;

                           case ShowEdgeList:

                                simpleGraph.printEdge();

                                break;

                            case BFS:

                                int s;

                                cout <<"Inserire indice nodo sorgente"<<endl;

                                while(!(cin >> s)){

                                   cin.clear();

                                   cin.ignore(numeric\_limits<streamsize>::max(), '\n');

                                   cout << "Per favore inserisci un indice valido!\n";

                                }

                                auto tmpBFS=simpleGraph.getNodeAtIndex(s);

                                while(tmpBFS==nullptr){

                                   cin.clear();

                                   cin.ignore(numeric\_limits<streamsize>::max(), '\n');

                                   cout << "Nodo non valido, riprova!\n";

                                   while(!(cin >> s)){

                                          cin.clear();

                                          cin.ignore(numeric\_limits<streamsize>::max(), '\n');

                                          cout << "Per favore inserisci un indice valido!\n";

                                  }

                                   tmpBFS=simpleGraph.getNodeAtIndex(s);

                                }

                                simpleGraph.BFS(tmpBFS);

                                break;

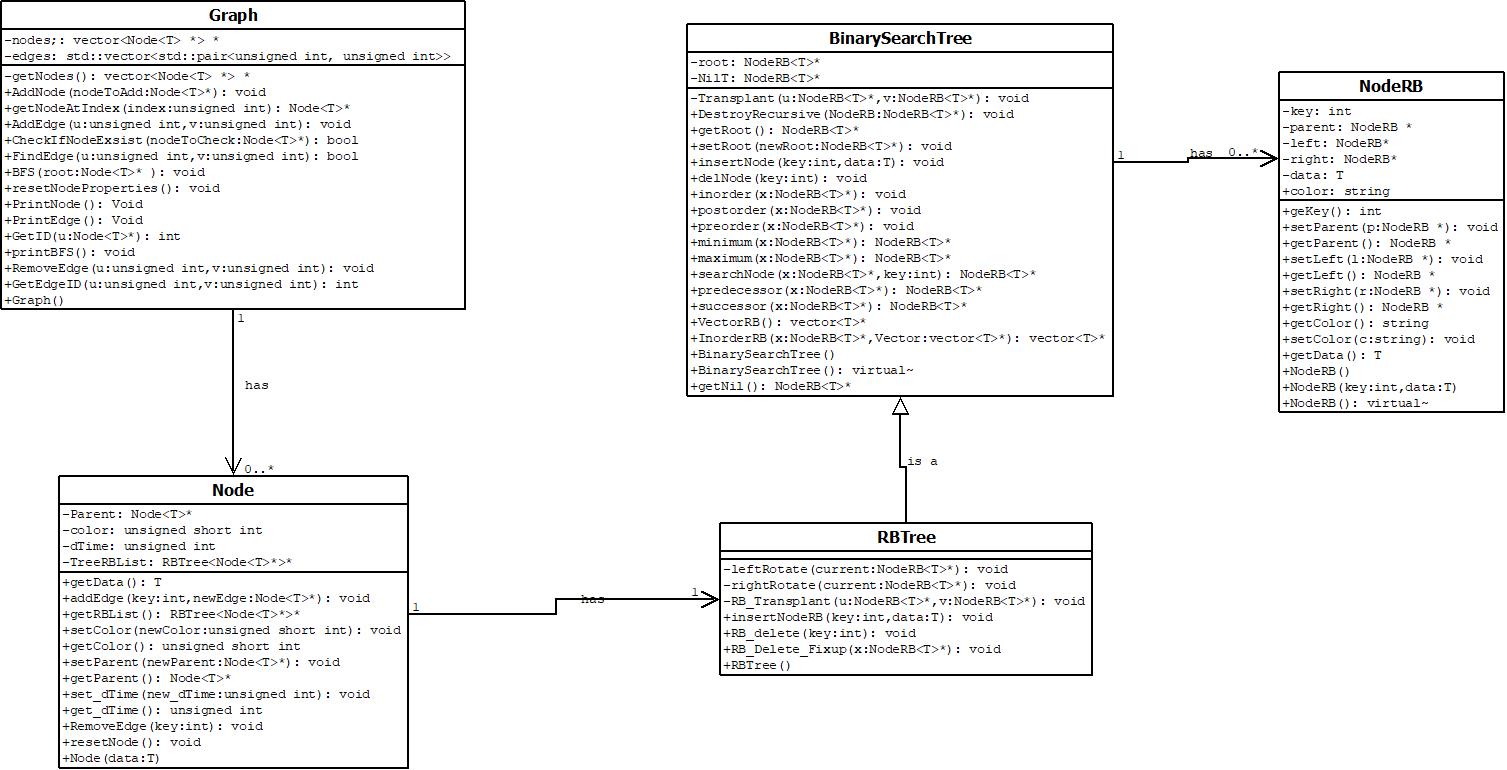
                    }

         }

         } while (choice != EXIT);

Quindi vengono associati dei numeri alle funzioni richiamabili, Add\_Edge è un aggiunta all’AddEdge dove gli indici dei nodi sono passati da terminale, inoltre, nel BFS sarà possibile scegliere il nodo sorgente da cui partire.

## Class Diagram



## Studio Complessità

Analizziamo ora la complessità delle funzioni richieste dalla traccia:

**AddEdge(i,j) & RemoveEdge(i,j)**

Per lo studio della complessità di questa funzione dobbiamo tenere conto principalmente di due funzioni **FindEdge**  ed **addEdge/removeEdge.** Quest’ultima funzione serve a inserire un nodo nella nostra lista di adiacenza, ovvero un albero RB; quindi, al suo interno vi è una chiamata a insertNodeRB che sappiamo avere complessità O(h) = O(log n), dove h è l’altezza dell’albero. FindEdge invece è una funzione che scorre tutti gli archi del nostro grafo, ovvero gli **E** archi nel grafo e restituisce true se trova l’arco altrimenti false, pertanto essa avrà una complessità O(m).

**N.B** in RemoveEdge(i,j) dovremmo considerare anche GetEdgeID() che ha comunque complessità O(m) pertanto la complessità finale è invariata.

La complessità totale, dunque, sarà **T(n)=O(E+logn)**

Complessità di spazio **S(n)=O(E+N)** perché utilizziamo sia il vettore degli archi che l’albero RB.

**FindEdge(i,j)**

FindEdge come discusso precedentemente è una funzione che scorre il vettore degli archi e restituisce true se lo trova altrimenti false.

Ha complessità **T(n)=O(E)** dove E sono il numero di archi.

**S(n)=O(E)**

**BFS(s)**

La complessità asintotica dell’implementazione classica con lista d’adiacenza del BFS è O(V+E).

Le operazioni di inserimento e di cancellazione di un elemento dalla coda è 𝑂 (1) , pertanto la coda in totale richiede 𝑂 (𝑉) . La lista di adiacenza di ogni vertice è esaminata al più una volta, quando questo viene eliminato dalla coda. Il tempo totale per esaminare tutte le liste di adiacenza è pari alla loro somma, ossia Θ(𝐸) .

Nel nostro algoritmo si deve tenere conto principalmente di un'altra funzione, ovvero, **VectorRB()** utilizzata nel ciclo for per poter mettere i nodi dell’albero RB in un vettore. Quest’ultima funzione principalmente richiama una visita **In order** che serve ad esplorare i nodi dell’albero ed inserirli nel vettore. Sappiamo che la visita in order ha complessità al più **O(n).**

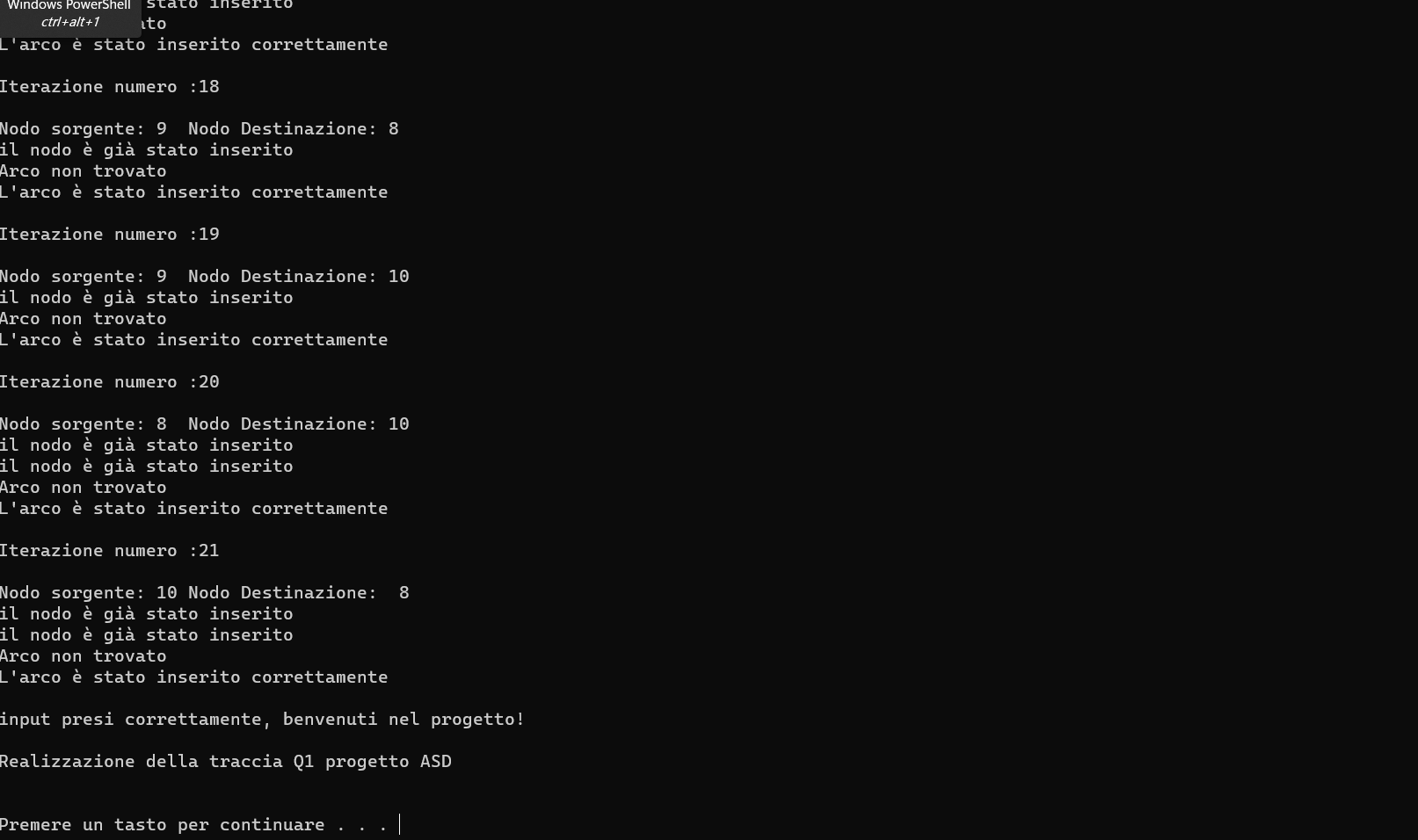
Ogni “Albero di adiacenza” viene esaminato al più una volta per ogni vertice. per tanto avremo che la complessità totale sarà **T(n)=O(V+E\*n).**

La complessità di spazio del BFS è **O(|V|)** a questa però dobbiamo aggiungere la complessità di spazio dell’albero di adiacenza che sappiamo essere **O(n)**.

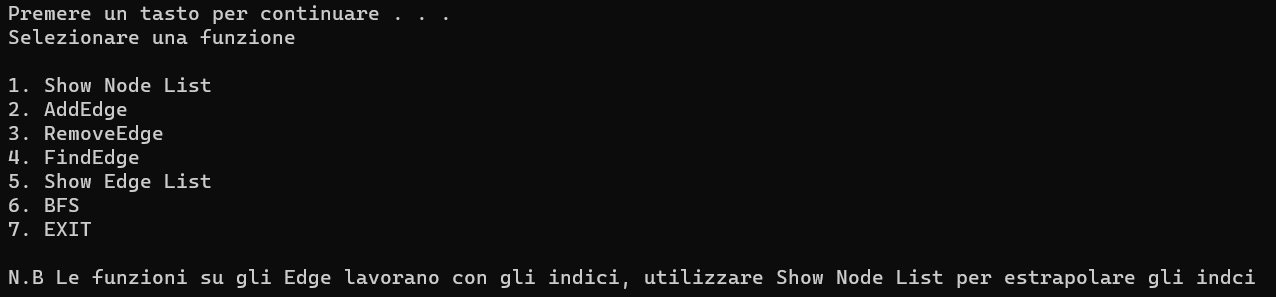
**S(n)=O(|V|+n)**

## Test e Risultati

All’inizio il programma ci mostrerà i dati presi in input da file



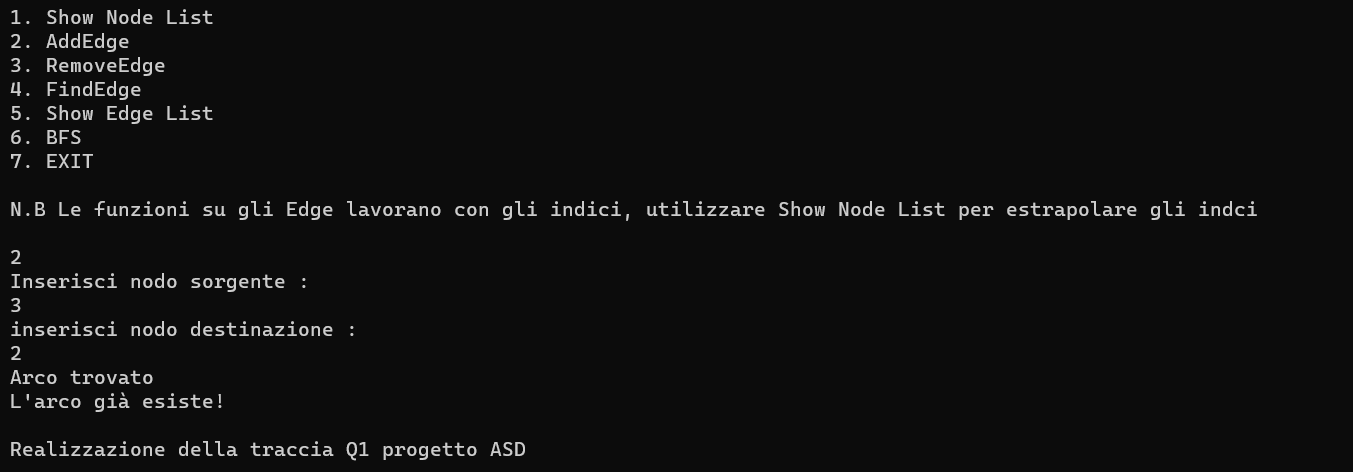
Dopodiché premendo un tasto qualsiasi avremo il menù



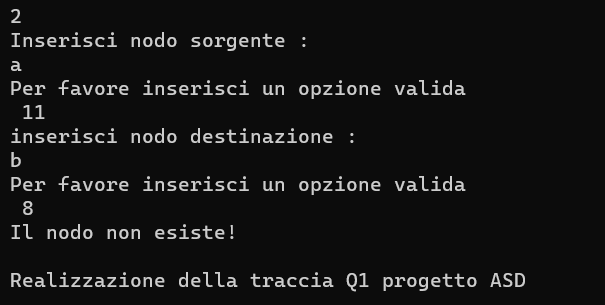
Esso ci mostrerà le funzioni disponibili con un piccolo N.B per la guida all’utilizzo.

Ora vedremo vari Test fatti sulle funzioni richieste dalla traccia.

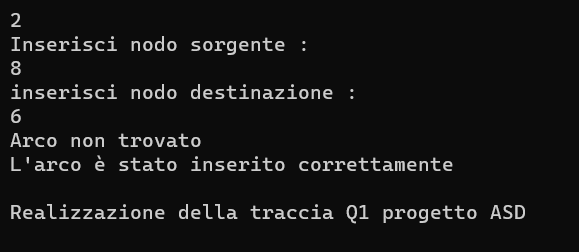
Questo è un test su AddEdge nel caso volessimo inserire un arco che già esiste.



Qui invece vediamo il controllo di due errori. È gestito sia l’inserimento di lettere, ovviamente non ammesso, ma anche l’inserimento di nodi non esistenti nel file di input; pertanto, non è possibile creare un arco.

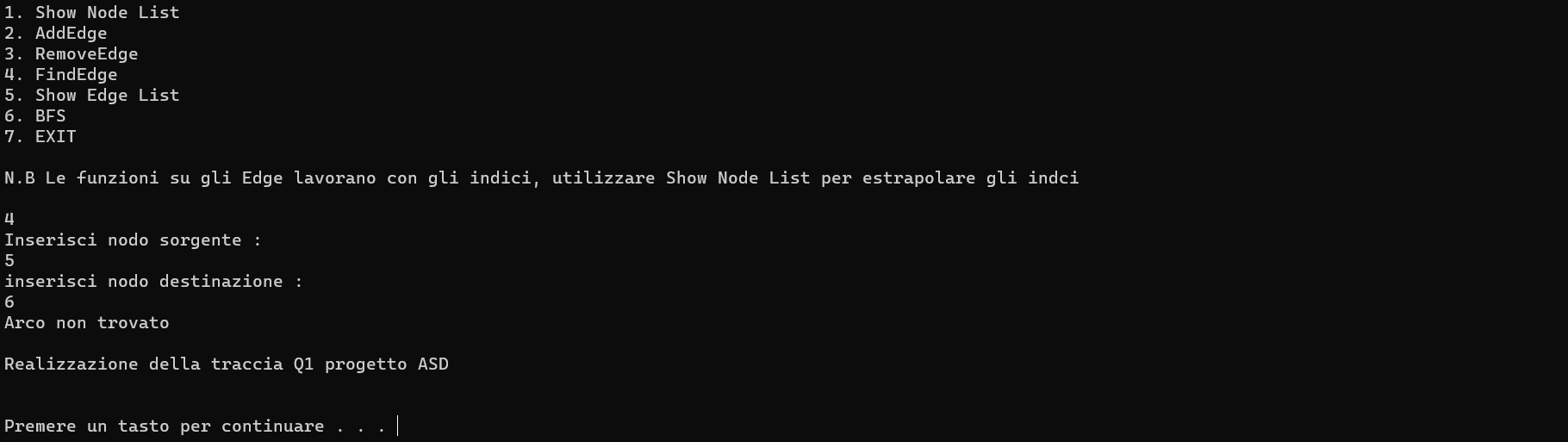


Qui invece vediamo un inserimento andato a buon fine. Sarà possibile controllare l’effettivamente aggiunta con l’opzione **ShowEdgeList**

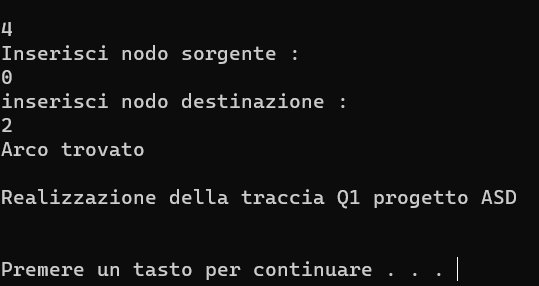


Altri risultati per le altre funzioni:

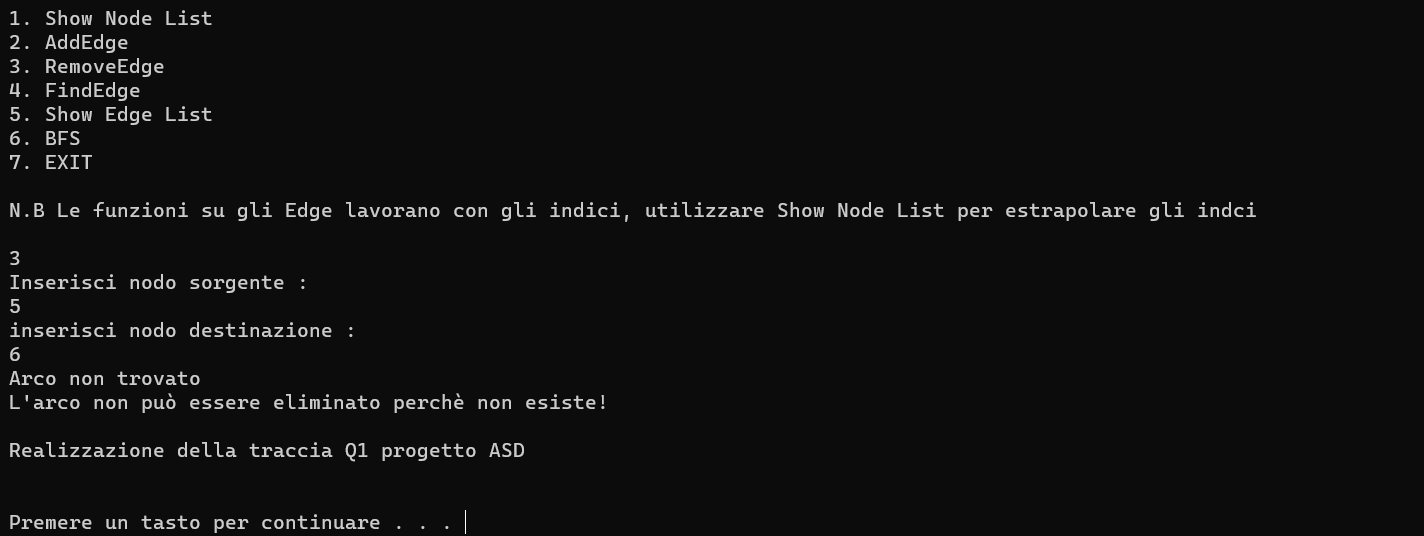
Nel caso cercassimo con FindEdge un arco che non esiste



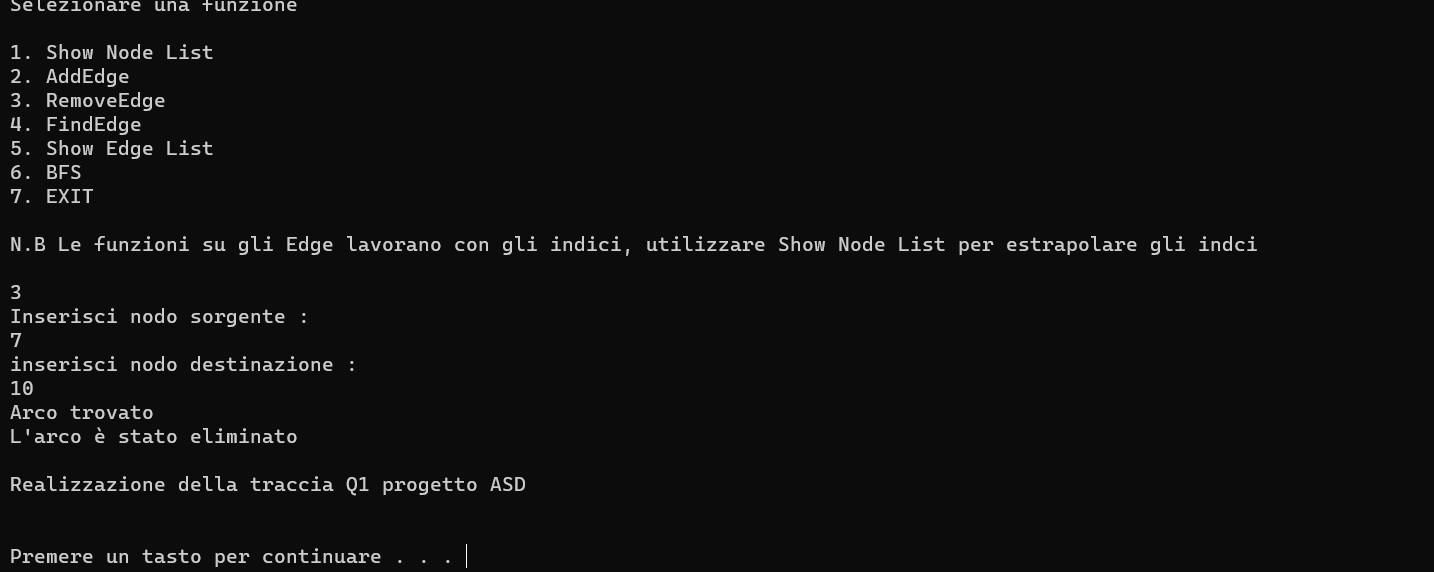
Tentativo andato a buon fine



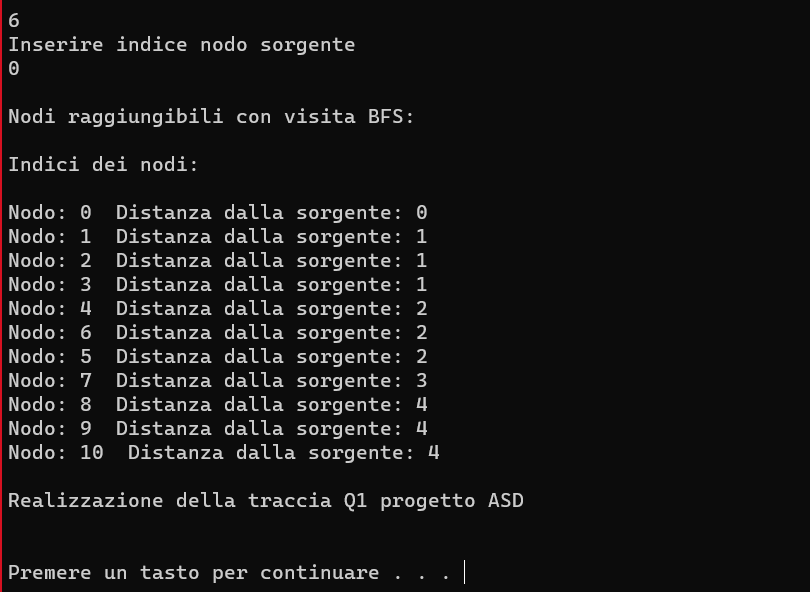
Tentativo non riuscito di RemoveEdge



Tentativo a buon fine



Esempio di BFS con errori e a buon fine



# Traccia 2

## Descrizione del problema

“Il primo ministro dello stato di Grapha-Nui ha necessità di risparmiare sulle spese di gestione. Per ottenere questo risultato, decide che di notte solo alcune tratte di autobus restino attive, con il vincolo che tutte le città debbano essere servite. A tal fine viene convocato un famoso informatico a cui viene fornita la piantina delle città con l’indicazione del costo di ogni tratta, con il compito di indicare quali percorsi mantenere attivi. Per prevenire eventuali interruzioni del servizio, viene richiesto di indicare l’insieme delle tratte di costo minimo e quello di costo immediatamente successivo.”

È assegnato un file di input contenente nel primo rigo due interi con il numero di città e di tratte, mentre, i successivi P righi contengono le tratte da una città all’altra e il loro costo.

Il quesito è stato risolto tramite la realizzazione di due MST.

## Descrizione strutture dati

Per realizzare la traccia abbiamo utilizzato principalmente due strutture dati:

* Grafi
* Insiemi Disgiunti (Per l’utilizzo dell’algoritmo di Kruskal)

Un **Grafo** G=(V,E) diretto o orientato:

* Insieme V di nodi (o vertici)
* insieme E di archi
* Un arco è una coppia ordinata (u, v) di nodi

La rappresentazione di un grafo può essere realizzata mediante:

* **lista di adiacenza:** ogni nodo è descritto da una lista di elementi contenenti il nome dei nodi ad esso adiacenti.
* **Matrice di adiacenza:** A = A(aij). Se aij =1 allora il nodo i è connesso a nodo j ,altrimenti se, aij =0 allora il nodo i non è connesso a nodo j. In un grafo non orientato la matrice è simmetrica.

**N.B** Se u, v ∈ V sono collegati da un arco (u, v) ∈ E, si dice che u è **adiacente** a v.

Un **Grafo** G=(V,E) indiretto o non orientato:

* Insieme V di nodi (o vertici)
* insieme E di archi
* Un arco è una coppia non ordinata (u, v) di nodi

In questa traccia è stato importante utilizzare un grafo non orientato e **pesato** per poter realizzare i due **MST**. Nella descrizione dell’algoritmo approfondiremo il concetto di MST.

**Strutture per insiemi disgiunti**

Un struttura per insiemi disgiunti mantiene una collezione S={S1,...,Sk} di insiemi dinamici disgiunti.

Ciascun insieme è identificato da un elemento dell'insieme detto rappresentante. Se si richiede il rappresentante di un insieme senza che questo venga modificato si deve ottenere la stessa risposta.

**Operazioni**:

* Make-Set(x) crea un insieme con unico elemento x
* Union(x,y) unisce gli insiemi dinamici che contengono x e y
* Find-Set(x) restituisce un puntatore al rappresentante dell'insieme che contiene x

Ciascun insieme può essere rappresentato tramite **lista concatenata** o **albero radicato**. In questo caso abbiamo scelto di rappresentarli come alberi radicati, per questo motivo approfondiremo questa rappresentazione.

**Albero radicato(**ogni nodo contiene un elemento**):**

* Ogni elemento punta solo al padre
* La radice contiene il rappresentante dell'insieme
* La radice è padre di sé stesso

Quindi avremo una foresta di insiemi disgiunti:

* Make-Set(x) crea un albero con un solo nodo
* Find-Set(x) segue i puntatori ai padri fino alla radice
* Union(x,y) fa puntare la radice di un albero alla radice dell'altro

Per l’unione è possibile utilizzare due euristiche, **l’unione per rango** e **la compressione del cammino**. Ho scelto di utilizzare l’unione per rango:

* Per ogni nodo si conserva il rango ovvero un limite superiore per l'altezza del nodo
* La radice con il rango più piccolo viene fatta puntare alla radice con il rango più grande
* Inizialmente ogni insieme di un solo elemento ha rango 0
* Se si uniscono due insiemi con lo stesso rango, l'insieme risultante ha rango incrementato di 1
* Se si uniscono due insiemi di rango diverso, l'insieme risultante ha rango pari al maggiore dei due

## Formato dati input/output

**Dati di input**:

È assegnato un file di testo contenente nel primo rigo due interi separati da uno spazio: il numero **N** delle città (1<=N<=1000, 0 rappresenta lo stazionamento degli autobus) ed il numero **P** delle tratte (0<=P<=10000). I successivi P righi contengono ciascuno tre numeri, separati da uno spazio, **C1 C2 C3** per indicare la presenza di una tratta dalla città C1 alla città C2 con costo C3.

**Dati di output**:

Determinare l’insieme delle tratte di costo minimo e quello di costo immediatamente successivo.

**Assunzioni**

* 1 ≤ N ≤ 1000
* 0 ≤ P ≤ 10000
* 1 ≤ C3 ≤ 10000

Pertanto, nel programma sono scritti i check opportuni in modo da rispettare le indicazioni della traccia.

L’input viene gestito dalla funzione AddEdge che tramite manipolazioni sul file prende in ingresso tutti i dati e popola le strutture dati.

Quindi l’input sarà assegnato da file, mentre l’output sarà mostrato in terminale.

## Descrizione algoritmo

L’algoritmo per prima cosa richiama la funzione **AddEdge** che serve a prendere i dati in input da file. Essa apre il file e controlla per prima la cosa la prima linea tramite la **getline()**, spezziamo la stringa in ingresso in due sottostringhe separate dallo spazio e poi utilizziamo **stoi()** per convertirle in intero. Facciamo i check opportuni e successivamente scorriamo le restanti M righe del file, corrispondenti al numero di tratte.

Il processo è molto simile solo che la stringa viene spezzata in tre, tramite stoi estrapoliamo gli interi, creiamo dei nuovi nodi **int** con gli interi estrapolati come argomento e successivamente possiamo richiamare le funzioni **addNode** e poi **Add\_Edge** del nostro grafo per inserire i dati nella struttura.

Di seguito sono riportati i codici per chiarire il percorso che le funzioni fanno.

Template <class T> void Graph<T>::addNode(Node<T> \* nodeToAdd) {

     if (!CheckIfNodeExsist(nodeToAdd)){

        nodes->push\_back(nodeToAdd);

    }else{

        cout <<"il nodo è già stato inserito"<<endl;

    }

}

template <class T> void Graph<T>::Add\_Edge(int u, int v, int w){

    if (!(FindEdge(u,v,w))){

    edges.push\_back({ w, { u, v } });

    getNodeAtIndex(u)->addEdge(this->getNodeAtIndex(v));

    cout << "arco inserito correttamente!"<<endl;

    }else{ cout<<"l'arco già esiste!"<<endl;}

}

template <class T> bool Graph<T>::FindEdge(unsigned int u, unsigned int v, unsigned int w){ //Cerca un arco nel vettore edge

    for ( const auto& edge: edges){

        if (edge.first == w && edge.second.first == u && edge.second.first == v){

            cout <<"Arco trovato"<<endl;

            return true;

        }

    }

    cout<<"Arco non trovato"<<endl;

    return false;

}

Parte focale dell’algoritmo è trovare **l’insieme delle tratte di costo minimo e quello di costo immediatamente successivo.**

Questo è stato tradotto in termini di programmazione nel cerca l’**MST** e poi l’**MST** immediatamente successivo.

Prima definiamo cos’è un **MST**:

Dato un grafo connesso non orientato pesato G=(V,E) con pesi w(u,v), vogliamo trovare un sottoinsieme aciclico T ⊆ E che collega tutti i vertici V, il cui peso totale w(T) = ∑(u,v)∈T w(u, v) sia minimo. Poiché T è aciclico e collega tutti i vertici, forma un albero chiamato albero di connessione minimo.

È possibile trovare un MST con due algoritmi **Prim** e **Kruskal** entrambi **greedy**. In questo caso è stato scelto di utilizzare l’algoritmo di Kruskal in quanto più opportuno alla traccia.

**L'algoritmo di Kruskal** trova un arco sicuro da aggiungere alla foresta scegliendo, tra tutti gli archi che collegano due alberi della foresta, quello con peso minimo.

* Siano C1 e C2 due alberi collegati da (u,v)
* L'arco (u,v) è un arco leggero e dunque un arco sicuro
* La scelta greedy prevede di scegliere ad ogni passo l'arco con il minor peso possibile

L'algoritmo utilizza una **struttura dati per insiemi disgiunti**, ogni insieme contiene i vertici di un albero della foresta corrente, con l'operazione Find-Set() verifichiamo se due nodi appartengono alla stessa componente connessa, mentre con l'operazione Union() uniamo i due alberi.

Quindi l’algoritmo creerà tanti insieme disgiunti quanti sono i nodi, e dopo aver ordinato gli archi in ordine non decrescente, tramite un iteratore scorriamo il vettore degli archi ed effettuiamo le operazioni di Kruskal precedentemente descritte, aggiornando il peso totale e creando un ulteriore vettore di archi **MST** il quale conterrà gli archi del **minimum spanning tree.**

template <class T> unsigned int Graph<T>::kruskalMST() {

    unsigned int MSTWeight = 0;

    std::sort(edges.begin(),edges.end());

    //per ogni u in G\_v

    //    MAKE-SET(u)

    DisjointSet disjointSet((unsigned int) this->nodes->size());

    //std::vector<std::pair<unsigned int, std::pair<unsigned int, unsigned int>>>::iterator it;

    for (auto it = edges.begin(); it != edges.end(); it++){

        unsigned int u = it->second.first; //indice nodo sorgente dell'arco

        unsigned int v = it->second.second; //indice nodo destinazione dell'arco

        unsigned int setU = disjointSet.findSet(u);

        unsigned int setV = disjointSet.findSet(v);

        if (setU != setV){

            unsigned int w = it->first;  //peso dell'arco

            MST.push\_back({w, {u, v}});

            MSTWeight += it->first;

            disjointSet.unionSet(setU, setV);

        }

    }

    return MSTWeight;

}

Quindi una volta trovato l’MST per trovare il secondo MST più grande codifichiamo la seguente idea:

Rimuoviamo uno alla volta dal vettore degli archi, gli archi dell’MST, così facendo possiamo ricalcolare il minimum spanning tree sul vettore degli edge, tante volte quanto è il size del vettore MST, per controllare il risultato più piccolo. Quindi ho utilizzato variabili temporanee che verranno azzerate ad ogni iterazione, attuando una “ricerca del minimo” ovviamente allargata al concetto del MST.

A fine di ogni iterazione l’arco inizialmente rimosso viene re-inserito nel vettore, per poi rimuovere l’arco successivo all’iterazione successiva e ripetere **Kruskal**. Il risultato finale verrà memorizzato nel vector **MST2** e il peso in **SecondBest**.

template <class T> unsigned int Graph<T>::kruskalMST2() {

    //dichiaro un cammino minimo tmp

    std::vector<std::pair<unsigned int, std::pair<unsigned int, unsigned int> >> MSTtmp;

    //Dichiaro una variabile che memorizzerà il peso del secondo cammino minimo

    unsigned int SecondBest=std::numeric\_limits<int>:: max();

    //Faccio un doppio ciclo for che rimuove uno alla volta gli archi dell'MST

    for (auto st = MST.begin(); st != MST.end(); st++){

        for (auto st2 =edges.begin(); st2 != edges.end(); st2++){

            if (st->second.first == st2->second.first && st->second.second == st2->second.second){

                int tmp= GetEdgeID(st->second.first,st->second.second);

                edges.erase(edges.begin()+tmp-1); //rimuovo l'arco uguale all'arco dell'MST

            }

        }

        //Ricalcolo Kruskal senza l'arco dell'MST rimosso

        unsigned int MSTWeight2 = 0;

        MSTtmp.clear();

        std::sort(edges.begin(),edges.end());

            //per ogni u in G\_v

            //    MAKE-SET(u)

        DisjointSet disjointSet2((unsigned int) this->nodes->size());

        for (auto it = edges.begin(); it != edges.end(); it++){

            unsigned int u = it->second.first; //indice nodo sorgente dell'arco

            unsigned int v = it->second.second; //indice nodo destinazione dell'arco

            unsigned int setU = disjointSet2.findSet(u);

            unsigned int setV = disjointSet2.findSet(v);

            if (setU != setV){

                unsigned int w = it->first;  //peso dell'arco

                MSTtmp.push\_back({w, {u, v}});

                MSTWeight2 += it->first;

                disjointSet2.unionSet(setU, setV);

            }

        }

        if (SecondBest > MSTWeight2){

            MST2.clear();

            SecondBest= MSTWeight2; //Aggiorno la variabile del secondo peso minimo migliore

            MST2.insert(MST2.end(), MSTtmp.begin(),MSTtmp.end()); //inserisco l'MST tmp dentro L'MST2

        }

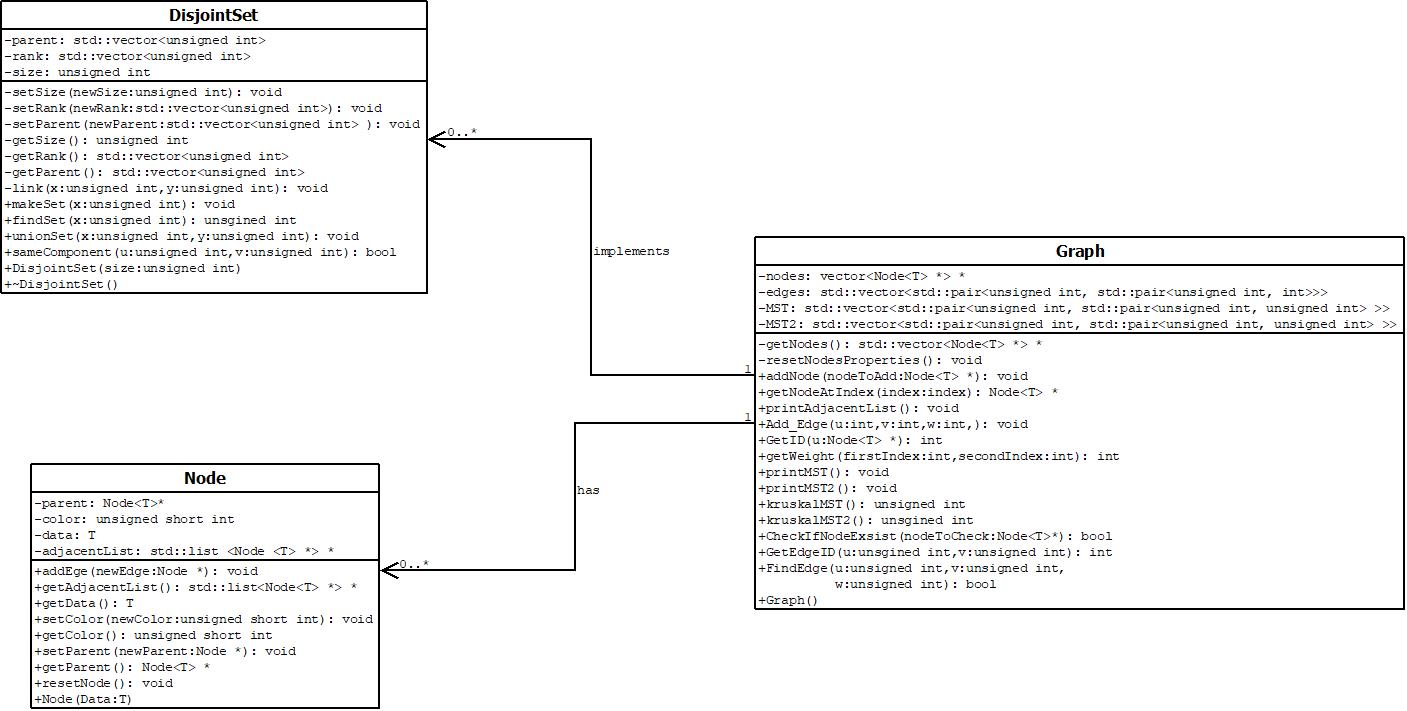
        edges.push\_back({ st->first, { st->second.first, st->second.second } }); //Riaggiungo l'arco che avevo rimosso e passo al successivo

    }

    return SecondBest;

}

## Class Diagram

****

## Studio Complessità

Per quanto riguarda la complessità ci concentriamo sul punto focale dell’algoritmo ovvero la richiesta di calcolare l’MST e il secondo MST.

Per farlo è stato utilizzato di base l’algoritmo di **Kruskal**, L’ inizializzazione richiede tempo 𝑂(𝑉), il tempo necessario per ordinare gli archi è 𝑂(𝐸 log 𝐸) = 𝑂(𝐸 log 𝑉 2 ) = 𝑂(𝐸 log 𝑉).

Le operazioni sulla foresta di insiemi disgiunti sono 𝑂(𝐸 𝛼(𝐸, 𝑉)) circa 𝑂(𝐸)

Quindi il tempo di esecuzione totale è pari a 𝑂(𝑉 + 𝐸 log 𝑉 +𝐸) = **O(E log V)**.

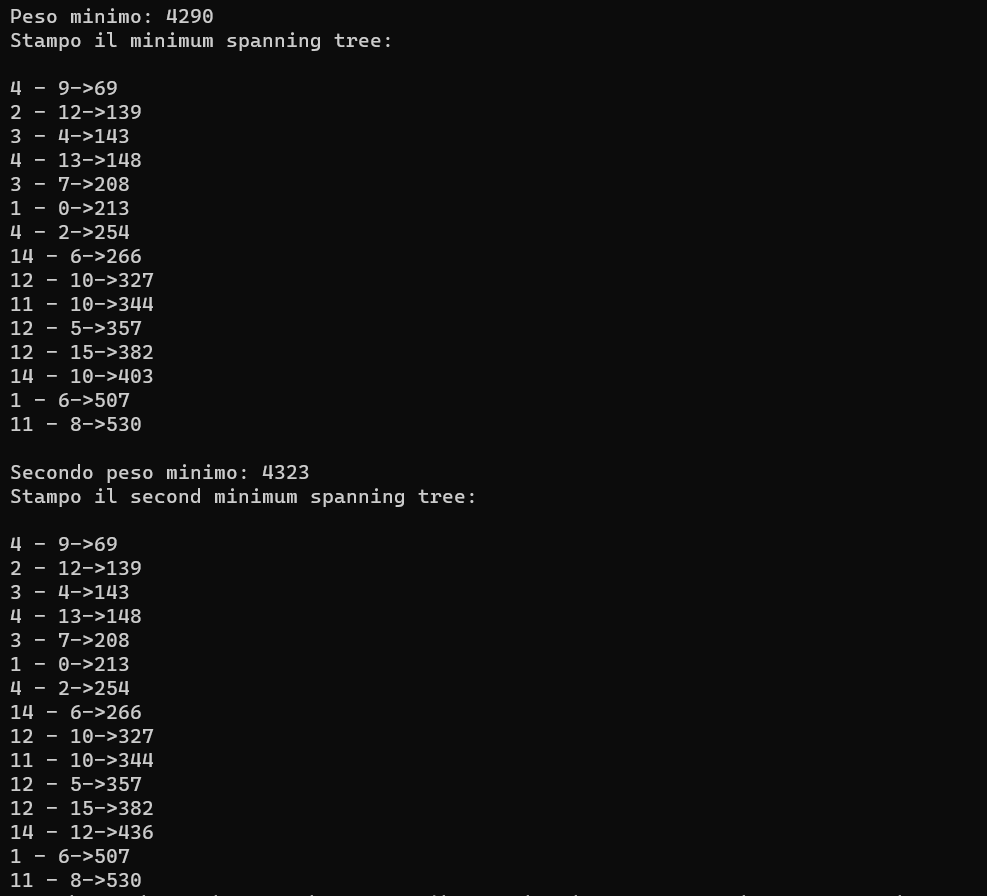
Per poter trovare il secondo cammino minimo dobbiamo ripetere Kruskal escludendo volta per volta un arco dall’insieme dell’MST; quindi, sarà ripetuto N volte dove N è il numero di archi nell’MST; quindi, avremo

**T(n) = O(N \* E log V).**

Posto che il vettore degli MST abbia size N, e che per l’implementazione del secondo MST più grande, abbiamo bisogno di ben 3 vettori per memorizzare l’MST primario, l’MST temporaneo ad ogni iterazione e l’MST finale. A questo poi sommiamo lo spazio occupato dal vettore degli Edge ottenendo così una complessità di spazio pari a **S(n)=(3N+E)=O(N+E)**

## Test e Risultati

Test e risultati con file di ingresso fornito dal professore:



Test e risultati violando i check della traccia

