PROGETTO LABORATORIO DI ALGORITMI E STRUTTURE DATI A.A. 2021/22

CANDIDATO: DE ROSA SALVATORE

MATRICOLA: 0124001981

Traccia scelta: TRACCIA 2

IDE Utilizzato: Codeblocks 17.12

TRACCIA 2, PRIMO QUESITO: HASHGRAPH

DESCRIZIONE PROBLEMA

Il problema consiste nel creare una struttura dati chiamata HashGraph, dove, tale struttura deve permettere di memorizzare un grafo orientato all'interno di un hash table, dove ogni nodo del grafo viene memorizzato in una cella di una hash table insieme alla sua lista di adiacenza.

Per fare ciò è richiesta per la realizzazione, a partire da un file di input fornitoci, di tale struttura le seguenti operazioni, AddEdge, RemoveEdge, FindEdge, DFS, infine per richiamare tali operazioni bisognerà dotare il programma di un menù in grado di richiamare le quattro operazioni sopra citate.

DESCRIZIONE STRUTTURE DATI

Come richiesto dalla traccia, le prime due strutture dati implementate saranno una hash table e un grafo.

- **Hash Table:**

  Tale struttura dati è stata realizzata attraverso il metodo dell'indirizzamento diretto, dove si è deciso di utilizzare il metodo della divisione per una corretta posizione tra la cella e i nodi del grafo, il metodo della divisione infatti prende la chiave "k" che verrà mappata all'interno in una delle "m" celle prendendo il resto della divisione fra "k" e "m"

  $$h(k)=k \bmod m$$

  Questa struttura dati viene rappresentata da una classe chiamata "HashTable" definita nell'header "HashTable.h", questa classe conterrà un vettore dove conterrà i vertici del grafo.

- **Grafo:**

  Tale struttura dati contiene un insieme di nodi, salvati nelle celle della Hash Table sopra citate, infatti, all'interno di ogni cella della Hash Table vengono salvati non solo i nodi del grafo, ma anche le sue adiacenze.
  La classe Nodo dispone anch'essa dei metodi di ricerca, eliminazione e inserimento in modo da inserire correttamente i nodi nella lista di adiacenza, inoltre verranno aggiunti i metodi di assegnazione del padre, i colori per determinare se il nodo è stato scoperto o meno, infine abbiamo i tempi di scoperta iniziale e finale.
  Per prendere le adiacenze di una determinata sorgente, si utilizza il metodo "getadj" dove viene richiamata la classe "NodoLL" dove ogni nodo viene collegato con un riferimento esplicito della lista di adiacenza.

- **HashGraph:**

  Tale struttura dati, come già citata prima, è composta da una Hash Table e, successivamente, dai Nodi del grafo, tale classe disporrà i metodi richiesti dalla traccia già citati in precedenza.

Formato dati in input/output

- **INPUT:**

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.

- **OUTPUT:**

Le operazioni eseguite dall' HashGraph: Addedge(i,j), RemoveEdge(i,j), FindEdge(i,j), DFS(s).

DESCRIZIONE ALGORITMO

- ## MAIN:

L'algoritmo inizia richiamando la libreria "fstream" per prelevare i dati di input forniti dal file.txt, preleveremo prima il numero dei nodi attraverso il "getline" delimitandolo fino al primo spazio, dopodiché richiamando la classe HashGraph inseriamo i nodi all'interno della Hash Table, dopodiché, usando sempre la libreria "fstream", preleviamo i nodi sorgenti e destinazioni dal file.txt, richiamiamo l'AddEdge per aggiungere un arco tra il nodo sorgente e il nodo destinazione.
Il menù, attraverso un ciclo while(1) compirà le operazioni richieste dalla traccia.

- ## HASHGRAPH:

1. **Filltable(int n)**

Come citato già nel main, inseriamo i nodi all'interno della Hash Table, richiamando il metodo "Hash_Insert".

2. **AddEdge(int i, int j)**

Ha il compito di creare un arco tra il nodo sorgente e destinazione (se non già presente, faremo questo controllo attraverso la FindEdge), dove, richiamando la get_cella della nostra hash table andremo a inserire, richiamando il metodo addEdge della classe Nodo, nella rispettiva cella l'arco corrispondente, se l'arco è già presente, oppure si inserirà un valore maggiore della dimensione della hash table, comparirà a schermo una schermata di errore.

3. **RemoveEdge(int i,int j)**

Ha il compito di eliminare un arco tra il nodo sorgente e destinazione, nel caso in cui non si inseriscano valori che superano la dimensione della nostra tabella di hash, cercherà all'interno della hash table (prendendo la rispettiva cella), attraverso il metodo della find edge della classe Nodo, il nodo da eliminare, se il valore sarà presente, allora

richiamando il removeEdge della classe nodo, l'arco verrà eliminato, altrimenti verrà stampato a schermo un messaggio di errore.

4. **FindEdge(int i,int j)**

Ha il compito di cercare un arco tra il nodo sorgente e destinazione, prende in input la chiave del nodo sorgente e la chiave del nodo destinazione, richiamando il metodo findEdge della classe Nodo, se avrà esito positivo restituirà l'esito della operazione positivo, questo sempre se si soddisfa la condizione della dimensione della hash table, restituirà true, altrimenti se non troverà l'arco restituirà falls.

5. **DFS(int s)**

Permette la visita in profondità del grafo, inizialmente tutti i nodi saranno colorati di bianco, poiché non ancora scoperti, dopo aver ricevuto in input la sorgente, se è presente e se rispetta il colore bianco, si effettuerà la DFS_Visit, quando la DFS troverà nuove radici significa che quel nodo non era raggiungibile dalla sorgente.

6. **DFS_Visit(Nodo *u)**

Qui è dove avverrà la visita in profondità, incrementando il tempo di scoperta dove time passerà a time+1 e colorerà la sorgente di grigio, visto che è stato scoperto, e per ogni adiacenza appartenente alla lista di adiacenza della sorgente andrà a controllare se il vertice è bianco, se rispetta tale condizione imposterà il padre del nodo in questione a "v" e avverrà una chiamata alla DFS_Visit, dove si ripeterà il processo di visita, e la sorgente diventerà di colore nero incrementando il tempo+1, altrimenti se non è bianco non farà nulla.

7. **SetTime(int time)**

Imposterò il tempo.

8. **GetTime()**

Prenderò il tempo.

9. **Hash_display()**

Stamperò l'intera hash table, con le sue liste di adiacenze.

- # HASH TABLE

1. **hashFunction(int k)**

Qui è dove effettuerò il metodo della divisione nella quale una chiave k viene mappata in una delle dim celle prendendo il resto della divisione fra k e dim.

2. **Hash_insert(int key)**

Questa metodo ha il compito di eseguire l'inserimento dei nodi del grafo all'interno della nostra tabella di hash, calcolando il corretto posizionamento attraverso la hashFunction.

3. **Get_cella(int x)**

Questo metodo restituisce la cella dell'indice richiesto, calcolando il corretto posizionamento attraverso la hashFunction

- # NODO

1. **SetValue(int value)**

Permette di immagazzinare un intero, che è il numero del nodo.

2. **GetValue()**

Permette di restituire il valore del nodo.

3. **SetColor(short color)**

Permette di immagazzinare il colore.

4. **Getcolor()**

Restituisce il colore.

5. **Setparent(Nodo *x)**

Permette di immagazzinare il padre del nodo.

6. **SetD(int d)**

Permette di immagazzinare il tempo di scoperta.

7. **GetD()**

Restituisce il tempo di scoperta.

8. **SetF(int f)**

Permette di impostare il tempo finale.

9. **GetF()**

Restituisce il tempo finale.

10. **Getadj()**

Restituisce le adiacenze del vertice.

11. **Printadj()**

Stampa l'intera lista di adiacenza.

12. **addEdge(Nodo *nodo)**

Aggiungeremo un arco, in base ai valori di input ricevuti, nella lista di adiacenza, se le adiacenze sono nulle allora si creerà un nuovo nodo, altrimenti verrà effettuato un push.

13. **removeEdge(int j)**

Permette la rimozione di un arco, in base ai valori di input ricevuti, dalla lista di adiacenza.

14. **findEdge(int j)**

Verifica la presenza di un arco, in base ai valori di input ricevuti, dalla lista di adiacenza, se è presente restituirà true, altrimenti false.

- # NODOLL

1. **SetNext(NodoLL *next)**

Operazione che ci permette di assegnare il nodo successivo a quello sulla quale viene richiamata.

2. **GetNext()**

Operazione che restituisce il nodo successivo.

3. **Getnodo(Nodo *nodo)**

Operazione che permette di restituire il valore del nodo.

4. **SetBottom(NodoLL *bottom)**

Operazione che permette di assegnare l'ultimo elemento inserito.

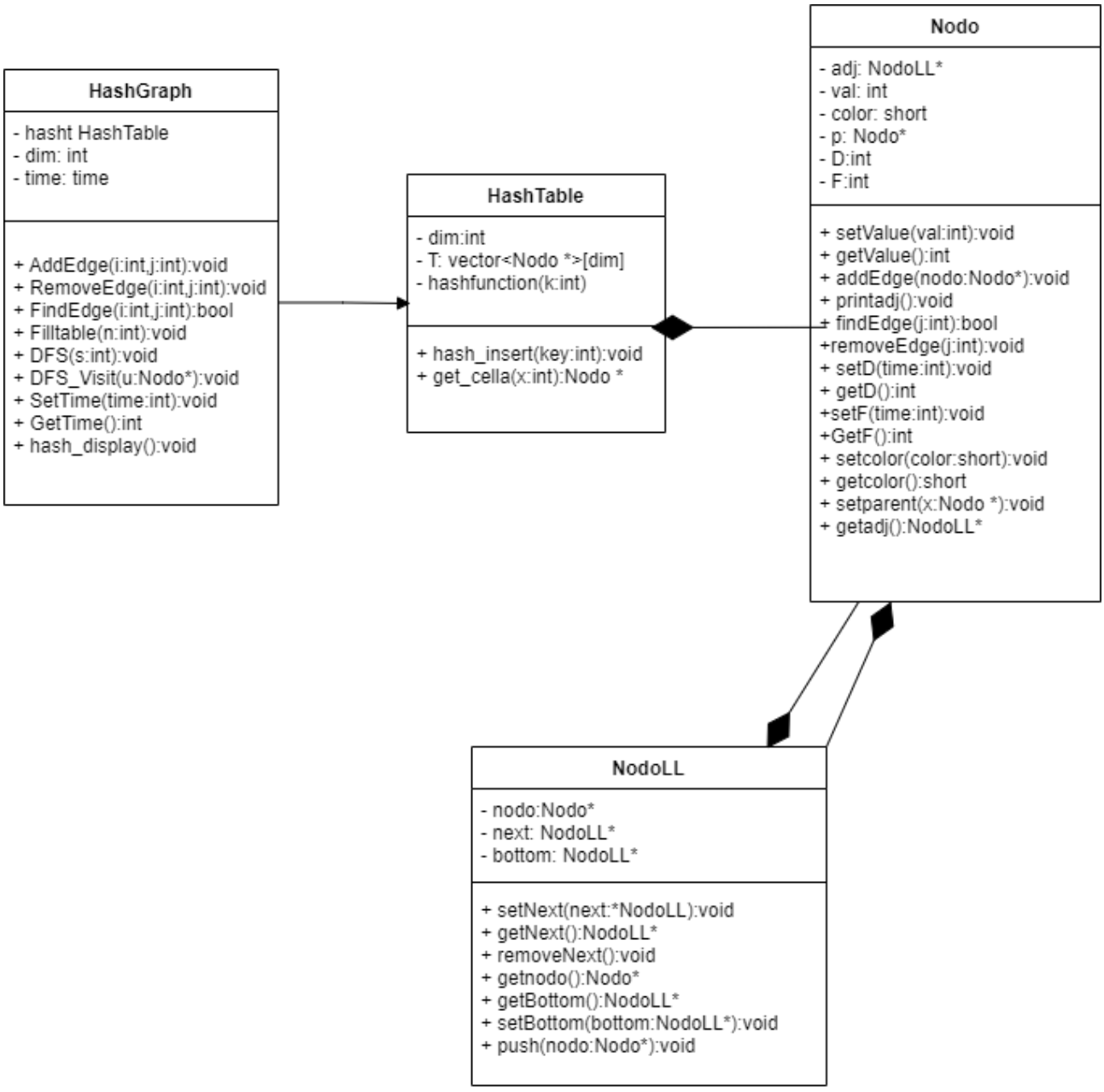
5. **GetBottom()**

Operazione che permette di restituire l'ultimo elemento.

6. **Push(Nodo *nodo)**

Operazione che permette di inserire un nodo all'interno della lista.

CLASS DIAGRAM



STUDIO COMPLESSITA'

Per valutare la complessità dobbiamo valutare le operazioni di AddEdge,FindEdge,RemoveEdge,DFS.

- **AddEdge:**

Siccome l'operazione d'inserimento ha O(1) come costo, ma, per aggiungere dobbiamo richiamare la cella della tabella di hash, la sua complessità sarà data da O(1+α)

- **FindEdge**

In questo punto bisogna considerare il caso in cui la ricerca del nodo scorra per tutta la lista, quindi avrà una complessità O(n) e siccome prima di ciclare la lista dobbiamo accedere alla cella della tabella di hash la complessità sarà pari a O((1+α)+n)

- **RemoveEdge**

In questo punto si deve considerare il caso in cui la ricerca del nodo da eliminare. Bisogna scorrere per tutta la lista, quindi avrà una complessità O(n), dobbiamo inoltre considerare che dobbiamo vedere se il nodo da eliminare esiste quindi richiamiamo la findEdge quindi un altro O(n), e siccome prima di ciclare la lista dobbiamo accedere alla cella della tabella di hash la complessità sarà pari a O((1+α)+n+n)

- DFS

In questo caso la DFS nell'hashgraph aggiunge sia il costo della DFS e sia il costo di ricerca all'interno dell'hash table, la ricerca è effettuata con il metodo della divisione, quindi considerando che la DFS ha complessità O(|V|+|E|) e la hash table O(1+ α) la DFS avrà complessità totale pari a O(|N| + |M| + (1+ α)) (dove N sono il numero dei nodi e M il numero degli archi).

TEST/RISULTATI

Input:

10 21

0 1

0 4

0 2

2 4

4 3

3 1

1 3

3 5

4 5

5 2

1 6

6 5

5 9

7 5

7 9

6 9

9 7

9 8

9 10

8 10

10 8

Menu:

```
0 ADJ LIST: 0  --> 1 --> 4 --> 2
1 ADJ LIST: 1  --> 3 --> 6
2 ADJ LIST: 2  --> 4
3 ADJ LIST: 3  --> 1 --> 5
4 ADJ LIST: 4  --> 3 --> 5
5 ADJ LIST: 5  --> 2 --> 9
6 ADJ LIST: 6  --> 5 --> 9
7 ADJ LIST: 7  --> 5 --> 9
8 ADJ LIST: 8  --> 10
9 ADJ LIST: 9  --> 7 --> 8 --> 10
10 ADJ LIST: 10  --> 8

******QUESITO 1*******
Menu
[1]Aggiunta di un edge
[2]Rimozione di un edge
[3]Cerca un edge
[4]Stampa del DFS
[5]Esci
**************
```

Inserimento:

```
1
Inserisci il nodo sorgente-> 2
Inserisci il nodo destinazione-> 5
0 ADJ LIST: 0  --> 1 --> 4 --> 2
1 ADJ LIST: 1  --> 3 --> 6
2 ADJ LIST: 2  --> 4 --> 5
3 ADJ LIST: 3  --> 1 --> 5
4 ADJ LIST: 4  --> 3 --> 5
5 ADJ LIST: 5  --> 2 --> 9
6 ADJ LIST: 6  --> 5 --> 9
7 ADJ LIST: 7  --> 5 --> 9
8 ADJ LIST: 8  --> 10
9 ADJ LIST: 9  --> 7 --> 8 --> 10
10 ADJ LIST: 10  --> 8
```

Cancellazione:

```
2
Inserisci il nodo sorgente-> 2
Inserisci il nodo destinazione-> 5
0 ADJ LIST: 0  --> 1 --> 4 --> 2
1 ADJ LIST: 1  --> 3 --> 6
2 ADJ LIST: 2  --> 4
3 ADJ LIST: 3  --> 1 --> 5
4 ADJ LIST: 4  --> 3 --> 5
5 ADJ LIST: 5  --> 2 --> 9
6 ADJ LIST: 6  --> 5 --> 9
7 ADJ LIST: 7  --> 5 --> 9
8 ADJ LIST: 8  --> 10
9 ADJ LIST: 9  --> 7 --> 8 --> 10
10 ADJ LIST: 10  --> 8
```

Ricerca:

```
3
Inserisci il nodo sorgente-> 2
Inserisci il nodo destinazione-> 5
Edge Non trovato
```

```
3
Inserisci il nodo sorgente-> 2
Inserisci il nodo destinazione-> 4
Edge Trovato
```

DFS:

```
4
Sorgente:
1
Nodo sorgente: 1
Nodo:1
Nodo:3
Nodo:5
Nodo:2
Nodo:4
Nodo:9
Nodo:7
Nodo:8
Nodo:10
Nodo:6
Radice:
Nodo:0
```

TRACCIA 2, SECONDO QUESITO: CONDOTTE IDRICHE

DESCRIZIONE PROBLEMA

Dopo diversi anni ed ingenti investimenti economici, finalmente lo stato di Grapha-Nui ha la sua diga in grado di produrre energia elettrica e fornire acqua potabile a tutti gli abitanti. E necessario pero terminare la rete idrica in modo che tutte le citta ricevano l'acqua. A tal fine viene convocato un famoso informatico a cui viene fornita la piantina delle citta con l'indicazione delle condotte attualmente presenti, con il compito di determinare il minimo numero di condotte da costruire.
Tale problema lo possiamo interpretare come un grafo, dove le città sono i nodi e le condotte idriche gli archi

DESCRIZIONE STRUTTURE DATI

- **Grafo:**

  Tale struttura dati contiene un insieme di nodi, che sono stati implementati attraverso un vettore dove al suo interno contengono i nodi che compongono il grafo, la classe Nodo dispone del metodo di inserimento, in modo da inserire correttamente i nodi nella lista di adiacenza, inoltre verranno aggiunti i metodi di assegnazione del padre, i colori per determinare, in base a quest'ultimo se il nodo è stato scoperto oppure no, infine abbiamo i tempi di scoperta iniziale e finale.
  Per prendere le adiacenze di una determinata sorgente, si utilizza il metodo "getadj" dove viene richiamata la classe "NodoLL" dove ogni nodo viene collegato con un riferimento esplicito della lista di adiacenza.

Formato dati in input/output

- **INPUT:**

E' assegnato un file di testo contenente nel primo rigo due interi separati da uno spazio: il numero N delle citta (1<= N<=1000, 0 rappresenta il bacino della diga) ed il numero P delle condotte idriche (0<=P<=10000).

- **OUTPUT:**

Determinare il minimo numero e quali condotte idriche costruire in modo da portare l'acqua in tutte le città.

DESCRIZIONE ALGORITMO

MAIN:

L'algoritmo inizia richiamando la libreria "fstream" per prelevare i dati di input forniti dal file.txt, preleveremo prima il numero dei nodi attraverso il "getline" delimitandolo fino al primo spazio, dopodiché richiamando la classe Grafo inseriamo i nodi all'interno del vettore, dopodiché richiamiamo il metodo getnodo per aggiungere un arco tra il nodo sorgente e il nodo destinazione.

1. **Min_costruzione(Grafo g)**

La funzione min_costruzione darà la soluzione al nostro problema, applicando un contatore per ogni condotta da collegare.
N.B.: nei cicli for si è deciso di partire da 1 perché la condotta idrica numero 0 è il bacino della città.

GRAFO:

1. **Filltgraph(int i)**

Come citato già nel main, inseriamo i nodi all'interno del vettore.

2. **DFS(int s)**

Permette la visita in profondità del grafo, inizialmente tutti i nodi saranno colorati di bianco, poiché non ancora scoperti, dopodiché si effettuerà la Visita_DFS, quando la DFS troverà nuove radici significa che quel nodo non era raggiungibile dalla sorgente. Per individuare il minimo numero di condotte da costruire bisognerà attuare una modifica

3. **Visita_DFS(Nodo *u)**

Qui è dove avverrà la visita in profondità, incrementando il tempo di scoperta dove time passa a time+1 e colorerà di grigio i nodi scoperti, e per ogni adiacenza appartenente alla lista di adiacenza della sorgente andrà a controllare se il vertice è bianco, se rispetta tale condizione imposterà il padre del nodo in questione a "v" e avverrà una chiamata alla Visita_DFS, dove si ripeterà il processo di visita, e la sorgente diventerà di colore nero incrementando il tempo+1, altrimenti se non è bianco non farà nulla.
La modifica necessaria affinché il problema si risolvi è quella di attuare un else if, ossia oltre a verificare i vertici di colore bianco verificheremo se il colore è nero del vertice (quindi tutti i nodi sono stati scoperti) impostiamo il padre dalla sorgente in cui si è fatta la visita.

4. **GetV()**

Mi restituirà il vettore.

5. **SetTime(int time)**

Imposterò il tempo.

6. **GetTime()**

Restituisce il tempo.

- # NODO

1. **SetValue(int value)**

Permette di immagazzinare un intero, che è il numero del nodo.

2. **GetValue()**

Permette di restituire il valore del nodo.

3. **SetColor(short color)**

Permette di immagazzinare il colore.

4. **Getcolor()**

Restituisce il colore.

5. **Setparent(Nodo *x)**

Permette di immagazzinare il padre del nodo.

6. **SetD(int d)**

Permette di immagazzinare il tempo di scoperta.

7. **GetD()**

Restituisce il tempo di scoperta.

8. **SetF(int f)**

Permette di impostare il tempo finale.

9. **GetF()**

Restituisce il tempo finale.

10. **Getadj()**

Restituisce le adiacenze del nodo.

11. **Printadj()**

Stampa l'intera lista di adiacenza.

12. **addEdge(Nodo *nodo)**

Aggiungeremo un arco, in base ai valori di input ricevuti, nella lista di adiacenza, se le adiacenze sono nulle allora si creerà un nuovo nodo, altrimenti verrà effettuato un push.

13. **removeEdge(int j)**

Permette la rimozione di un arco, in base ai valori di input ricevuti, dalla lista di adiacenza.

14. **findEdge(int j)**

Verifica la presenza di un arco, in base ai valori di input ricevuti, dalla lista di adiacenza, se è presente restituirà true, altrimenti false.

- # NODOLL

1. **SetNext(NodoLL *next)**

Operazione che ci permette di assegnare il nodo successivo a quello sulla quale viene richiamata.

2. **GetNext()**

Operazione che restituisce il nodo successivo.

3. **Getnodo(Nodo *nodo)**

Operazione che permette di restituire il valore del nodo.

4. **SetBottom(NodoLL *bottom)**

Operazione che permette di assegnare l'ultimo elemento inserito.

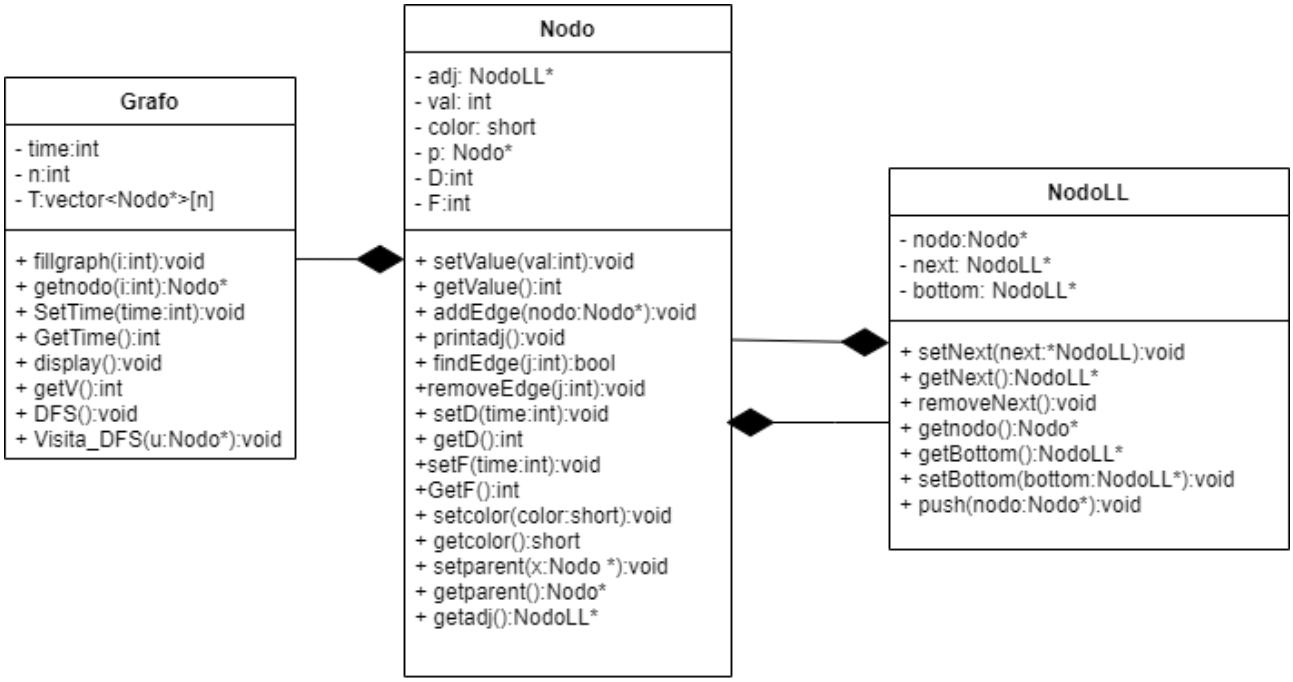
5. **GetBottom()**

Operazione che permette di restituire l'ultimo elemento.

6. **Push(Nodo *nodo)**

Operazione che permette di inserire un nodo all'interno della lista.

CLASS DIAGRAM



STUDIO COMPLESSITA'

- **DFS**

La DFS è un algoritmo noto la cui complessità è O(|V|+|E|), dove V è il numero dei nodi ed E il numero di archi. L'assegnazione dell'else if sarà un O(1) quindi sommandola con la DFS avremo una complessità pari a O(|N|+|P|) (dove N sono le città e quindi il numero dei nodi e P le condotte idriche ossia gli archi)

- **Min_costruzione**

Siccome effettuerà la ricerca dal nodo sorgente per trovare il minimo numero di condotte da inserire e quali saranno le condotte idriche avremo una complessità pari a O(N)

TEST/RISULTATI

Test file input0_2_2.txt

25 24
2 1
2 3
2 5
4 1
4 3
4 5
6 11
7 6
8 6
9 6
10 6
11 12
11 13
11 14
11 15
16 24
17 24
18 24
19 24
20 24
21 24
22 24
23 24
24 25

Output:

```
Per portare l'acqua nelle citta' bisogna costruire nei condotti
[0]-[2]
[0]-[4]
[0]-[7]
[0]-[8]
[0]-[9]
[0]-[10]
[0]-[16]
[0]-[17]
[0]-[18]
[0]-[19]
[0]-[20]
[0]-[21]
[0]-[22]
[0]-[23]

Il risultato e'->14
```

Test dal pdf Tracce_21_22.pdf

4 2

3 1

2 1

Output:

```
Per portare l'acqua nelle citta' bisogna costruire nei condotti
[0]-[2]
[0]-[3]
[0]-[4]

Il risultato e'->3
```