Relazione traccia 2

Quesito 1

Davide De Angelis #0124002412

1 DESCRIZIONE DEL PROBLEMA

La traccia richiede di implementare una struttura dati hashGraph che permetta la memorizzazione di un grafo orientato all'interno di un hashtable. I vertici dovranno essere memorizzati ognuno all'interno di una cella dell'hashtable e dovranno anche conservare la loro lista di adiacenza. Le dimensioni e gli archi del grafo vengono date a priori tramite un file con uno standard predefinito. Una volta terminata la creazione dell'hashGraph, bisogna creare un interfaccia grafica che permetta all'utente di aggiungere, trovare e rimuovere un arco all'interno dell' hashGraph e anche di effettuare una dfs su una sorgente stabilita dall'utente. Bisognerà infine mostrare i risultati delle suddette operazioni all'utente. Le operazioni richiedono solamente l'aggiunta e l'eliminazione degl'archi, non dei vertici.

2 DESCRIZIONE STRUTTURE DATI

- **linked list**
  La struttura dati linked list viene utilizzata per memorizzare la lista di adiacendza di ogni vertice. Grazie ai suoi costi ci garantisce una buona complessità, dove l'inserimento è effettuato con complessità costante mentre l'eliminazione e la ricerca con complessità lineare. Vengono utilizzati i template per renderla utilizzabile con più tipi di dati.
  La struttura è dotata di un attributo: nodo <T>*header. Esso indica la testa della linkedList (Viste le operazioni da effettuare e che l'inserimento viene effettuato in testa, si può evitare di salvare il puntatore alla fine. Utilizzandolo, nel caso peggiore avremmo potuto effettuare ricerca ed eliminazione in leght/2 ma asintoticamente la complessità rimane sempre O(lenght)). Le funzioni di cui è dotata questa struttura sono:

  - **nodo<T>*getHeader() const;**
    La funzione restituisce il puntatore all'header.
  - **void setHeader(nodo<T>*header);**
    La funzione permette di settare l'header.
  - **void push(T v);**
    La funzione, dato un dato T, crea un nodo da inserire nella linkedList contenente il dato. L'inserimento viene effettuato in testa.
  - **nodo<T>* search(T v);**
    La funzione, avendo un dato T, verifica se esiste un nodo all'interno della linkedList contenente il dato. In caso di successo viene restituito il puntatore al nodo, altrimenti nullptr.
  - **bool removeNode(T v);**
    La funzione, avendo un dato v di tipo T, rimuove dalla linkedList il nodo

contenente v. In caso di successo viene restituito true, altrimenti, qualora non esistesse un nodo contenente v viene restituito false.

La struttura per essere implementata necessita dell'implementazione della classe nodo.

- **nodo**
  La classe nodo è implementata con i template ed è costituita da due attributi: T info e nodo *pNext. Il primo contiene l'informazione del nodo e il secondo il puntatore al nodo successivo. Le funzioni di cui è dotata sono:
  * **nodo(T info);**
    E' il costruttore. Avendo un dato T inizializza il campo info del nodo con il dato.
  * **T getInfo() const;**
    La funzione restituisce il dato contenuto nel campo info del nodo.
  * **void setInfo(T info);**
    La funzione permette di settare il campo info del nodo.
  * **nodo<T>*getPNext() const;**
    La funzione restituisce il puntatore al nodo successivo.
  * **void setPNext(nodo *pNext);**
    La funzione permette di settare il puntatore al nodo successivo.

- **hashTable**
  La struttura dati hashTable viene utilizzata per immagazzinare i vertici di un grafo. Ogni cella dovrà contenere un vertice con la sua lista di adiacenza. Avendo il numero dei vertici fornito a priori e sapendo che ogni cella dovrà contenere un elemento, senza operazioni di aggiunta vertici successive all'inizializzazione, l'hashTable viene inizializzata utilizzando il metodo dell'indirizzamento aperto. Rispetto al metodo del concatenamento non vengono creati neanche dati aggiuntivi per la creazione delle linkedList. Non viene richiesta nemmeno la rimozione dei vertici che nell'indirizzamento aperto potrebbe modificare i tempi di ricerca. Inoltre, come spiegato nella sezione formato dati input/output, la ricerca di un vertice non esistente non cambia la complessità delle operazioni.
  La struttura è implementata con i template e nelle funzioni (dove richiesto) viene utilizzata una classe dell'std library. Questa permette di memorizzare nei vertici anche altri tipi di dati oltre all'intero in quanto associa un intero al dato(Se è già intero viene restituito l'intero stesso).
  La struttura hashTable è costituita da due attributi: int dim e vertice<I>**T. Il primo contiene la dimensione dell'hashTable mentre il secondo è un array di puntatori a vertici.
  Le funzioni di cui è composta sono:

- **hashTable(int n);**
  E' il costruttore. Dato un interno n inizializza l'hashtable di dimensioni n e imposta il campo dim dell'hashTable ad n.
- **int hashFunction1(I key);**
  La funzione restituisce il risultato della funzione di hash con il metodo della moltiplicazione.
- **int hashFunction2(I key);**
  La funzione restituisce il risultato della funzione di hash con il metodo della divisione.
- **int doppioHashing(I key, int i);**
  La funzione restituisce il risultato del doppio hashing.
- **int hashInsert(vertice<I>*key);**
  La funzione, dato un vertice lo inserisce nell'hashTable. Se l'operazione è avvenuta con successo viene restituito l'indice del vertice nell'hashTable, altrimenti viene restituito -1.
- **vertice<I>*hashSearch(I key);**
  La funzione, data una chiave, permette di cercarla all'interno dell hashTable. Se la chiave è contenuta in un vertice nell'hashTable, esso viene restituito altrimenti viene restituito nullprt.
- **int getDim() const;**
  La funzione restituisce la dimensione dell'hashTable
- **vertice<I>*getVertice(int i);**
  La funzione restituisce l'elemento contenuto nell'hashTable nella posizione data in input. Se la posizione non esiste viene restituito nullptr.

La struttura per essere implementata necessita dell'implementazione della classe vertice.

- **vertice**
  La classe vertice è implementata con i template ed è costituita da 6 attributi: T data(Contiene il dato T); linkedList<vertice*>*adj(Contiene il puntatore alla linkedList); vertice *p(Contiene il puntatore al padre utilizzato nella visita); int color(Contiene un intero che indica il colore nella visita); int d(Contiene un intero che indica il tempo di inizio scoperta della visita); int f(Contiene un intero che indica il tempo di fine scoperta della visita); Le funzioni di cui è composta la struttura sono:
  - **vertice(T data);**
    E' il costruttore.Dato in input un dato di tipo T assegna al campo data del vertice il dato in input. Successivamente inizializza la lista di adiacenza e setta le variabili destinate alla visita.

* **T getData() const;**
  La funzione restituisce il dato contenuto nel campo data del vertice.
* **void setData(T data);**
  La funzione permette di settare il campo data del vertice.
* **linkedList<vertice *>*getAdj() const;**
  La funzione restituisce la lista di adiacenza del vertice.
* **vertice<T>*getP() const;**
  La funzione restituisce il puntatore al padre del vertice.
* **void setP(vertice<T>*p);**
  la funzione permette di settare il puntatore al padre del vertice.
* **int getColor() const;**
  La funzione restituisce un intero corrispondente al colore del vertice.
* **void setColor(int color);**
  La funzione permette di settare il colore del vertice.
* **int getD() const;**
  La funzione restituisce il tempo di inizio scoperta del vertice.
* **void setD(int d);**
  La funzione permette di settare il tempo di inizio scoperta del vertice.
* **int getF() const;**
  La funzione restituisce il tempo di fine scoperta del vertice
* **void setF(int f);**
  La funzione permette di settare il tempo di fine scoperta del vertice
* **void addEdge(vertice *v);**
  La funzione, prendendo in input un vertice, lo inserisce nella lista di adiacenza del vertice corrente creando l'arco.
* **bool searchEdge(vertice* v);**
  La funzione, dato un vertice, restituisce un valore booleano che indica se il vertice è presente o meno nella lista di adiacenza del vertice corrente, verificando dunque l'esistenza dell'arco.
* **bool removeEdge(vertice* v);**
  La funzione, dato un vertice, restituisce un valore booleano che indica se il vertice è stato rimosso o meno dalla lista di adiacenza del vertice corrente. Se il vertice non è stato rimosso significa che non è presente nella lista di adiacenza e dunque l'arco non esiste.
* **void setVisit();**
  La funzione setta tutti i parametri della visita sovrascrivendo i dati della visita effettuata precedentemente e permettendone una nuova.

- **hashGraph**
  Questa è la struttura dati richiesta dalla traccia. E' implementata usando i template e le funzioni (pubbliche) qui presenti verranno richiamate nel main per risolvere

i problemi richiesti. E' costituita da un attributo: hashTable<I>*T. Questa è l'hashTable in cui verranno memorizzati i vertici con le relative liste di adiacenza. I metodi da cui è costituita sono:

- **void dfs_visit(vertice<I>*s, int &time);**
  Implementazione della dfs_visit
- **hashGraph(int n);**
  E' il costruttore, prende in input un intero e crea un hashtable della dimensione dell'intero.
- **vertice<I>*searchVertice(I u);**
  La funzione, dato un intero u, restituisce l'elemento di indice u nell'hashTable. Se questo non esiste restituisce nullptr.
- **int addVertice(I u);**
  La funzione, dato un dato u di tipo I, crea un vertice contenente u e lo inserisce nell'hashTable. Se l'inserimento è avvenuto con successo viene restituito 1. I casi particolari gestiti da questa funzione sono due. Il primo riguarda il caso in cui l'informazione che si sta cercando di inserire nel vertice esiste già in un altro vertice dell'hashGraph. In questo caso non viene creato il vertice e viene restituito -1. Il secondo è quando l'hashTable T è piena. In questo caso il vertice viene creato per provare l'inserimento ma visto l'insuccesso dell'operazione viene deallocato e viene restituito 0.
- **int addEdge(I u, I v);**
  La funzione aggiunge un arco al grafo. La funzione inizialmente crea due puntatori a vertice che contengono rispettivamente il risultato della ricerca dei due vertici richiesti. Se questi esistono e non esiste già l'arco, il vertice contenente v viene inserito nella lista di adiacenza del vertice contente u e viene restituito 1 per segnalare il successo dell'operazione. Se invece esiste già l'arco, quindi il vertice contenente v è già presente nella lista di adiacenza del vertice contenente u, viene restituito 0. Se invece dalla ricerca iniziale uno dei due vertici risulta non esistere viene restituito -1.
- **int searchEdge(I u, I v);**
  La funzione cerca un arco del grafo. La funzione inizialmente crea due puntatori a vertice che contengono rispettivamente il risultato della ricerca dei due vertici richiesti. Se questi esistono ed esiste l'arco, viene restituito 1 per segnalare il successo dell'operazione mentre se l'arco non esiste viene restituito 0. Se invece dalla ricerca iniziale uno dei due vertici risulta non esistere, viene restituito -1.
- **int removeEdge(I u, I v);**
  La funzione rimuove un arco dal grafo. La funzione inizialmente crea due puntatori a vertice che contengono rispettivamente il risultato della ricerca dei due vertici richiesti. Qualora questi esistono viene mandata l'eliminazione

dell'arco. Se l'arco è stato eliminato viene restituito 1 altrimenti se non è stato eliminato perchè l'arco non esiste viene restituito 0.Infine se invece dalla ricerca iniziale uno dei due vertici risulta non esistere allora viene restituito -1.

- **void dfs(vertice<I>*s);**
  Implementazione della dfs. Qui si inizializzano gli attributi di visita dei vertici solo se questi sono già stati utilizzati in una visita precedente.
- **void printDfsVisit();**
  Stampa i valori ottenuti con la dfs.
- **int getDimension();**
  Restituisce la dimensione dell'hashGraph

3 FORMATO DATI IN INPUT/OUTPUT

Il file di input dato contiene nel primo rigo due numeri interi, 0<=N<=1000 e 0<=M<=1000 , separati da uno spazio e 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. Stabiliamo a priori che prima di leggere gli archi inizializziamo tutti i vertici da 0 ad N-1 inserendoli nell'hashGraph. Questo perchè i vertici mostrati negli M righi vanno da minimo 0 a N-1(Nel caso del file dato arrivano fino ad N ma con gli archi è mostrata l'esistenza di tutti i vertici che vanno da 0 a N-1. Quindi il vertice N è in eccesso) e potrebbe esserci un vertice senza archi che quindi non è mostrato negli M righi. **Grazie a questo formato input è possibile verificare se un vertice dato è maggiore o uguale alle dimensioni dell'hashGraph. In questo caso non viene effettuata nessuna operazione in quanto sicuramente non esiste in esso(Così facendo il costo delle operazioni sull'hashTable resteranno O(1))**.
Tutti gli input inseriti dall'utente dove si richiede un intero per le operazioni da effettuare , sono gestiti in modo tale che se l'utente inserisce un altro tipo di dato, l'errore viene gestito opportunamente e viene mostrato in output un messaggio d'errore. (Questo non vale per la richiesta della scelta del path in quanto si richiede di inserire un qualsiasi altro valore per il path predefinito). Per effettuare ciò vengono utilizzate due funzioni: *inputVertici* e *verifyInt*. Entrambe restituiscono un valore booleano e mostrano un messaggio d'errore in caso di inserimento incorretto.
Per mostrare le operazioni all'utente utilizziamo un menù. Esso innanzitutto stampa il menù delle operazioni possibili. Una volta scelta l'operazione correlata l'utente dovrà inserire i relativi input richiesti in base all'operazione scelta (vertici o singolo vertice). Una volta svolta l'operazione verrà mostrato all'utente sotto forma di frase il relativo risultato dell'operazione ed eventuali errori.

4 DESCRIZIONE DELL'ALGORITMO

Il programma parte creando un hashGraph tramite la funzione *hashGraph<int>*inizializza()*. Questa funzione innanzitutto chiede all'utente se i dati riguardanti il grafo devono essere letti dal path predefinito o bisogna cambiarlo. Dopo che l'utente ha effettuato la scelta viene letta la dimesnione dell'hashGraph (Se il file esiste) e se la dimensione è maggiore di 0, vengono creati i vertici con la funzione *int hashGraph<I>::addVertice(I u)*, altrimenti viene restituito nullptr. Successivamente vengono letti gli archi ed inseriti con la funzione *int hashGraph<I>::addEdge(I u, I v)* mostrando all'utente eventuali errori. Una volta terminata questa funzione e restituito l'hashGraph creato, viene controllato se questo esiste oppure si è avuto un errore nell'apertura del file o le dimensioni dell'hashGraph non permettono di effettuare operazioni. In questi casi viene mostrato in output all'utente un messaggio d'errore. Successivamente viene chiamata la funzione *void menu(hashGraph<int>*T)*. Qui verrà mostrato all'utente un menù con le operazioni da poter effettuare. Una volta che l'utente ha inserito opportunamente l'input (verificato con la funzione *verifyInt*) ci possiamo trovare in 6 casi:

- **caso 1**:
  L'utente ha richiesto di aggiungere un arco inserendo 1. Dopo aver fatto inserire all'utente i due vertici interessati viene verificata la correttezza dell'inserimento e se i due vertici esistono nell'hashGraph. Se si ha esito negativo viene mostrato all'utente un apposito messaggio d'errore. Se si ha esito positivo, viene chiamata la funzione *int hashGraph<I>::addEdge(I u, I v)* e in base al risultato ottenuto viene mostrato all'utente se l'inserimento è avvenuto con successo o se l'arco non è stato inserito perchè già esistente.

- **caso 2**: L'utente ha richiesto di rimuovere un arco inserendo 2. Dopo aver fatto inserire all'utente i due vertici interessati viene verificata la correttezza dell'inserimento e se i due vertici esistono nell'hashGraph. Se si ha esito negativo viene mostrato all'utente un apposito messaggio d'errore. Se si ha esito positivo viene chiamata la funzione *int hashGraph<I>::removeEdge(I u, I v)* e in base al risultato ottenuto viene mostrato all'utente se l'eliminazione è avvenuta con successo o se l'arco non è stato rimosso perchè inesistente.

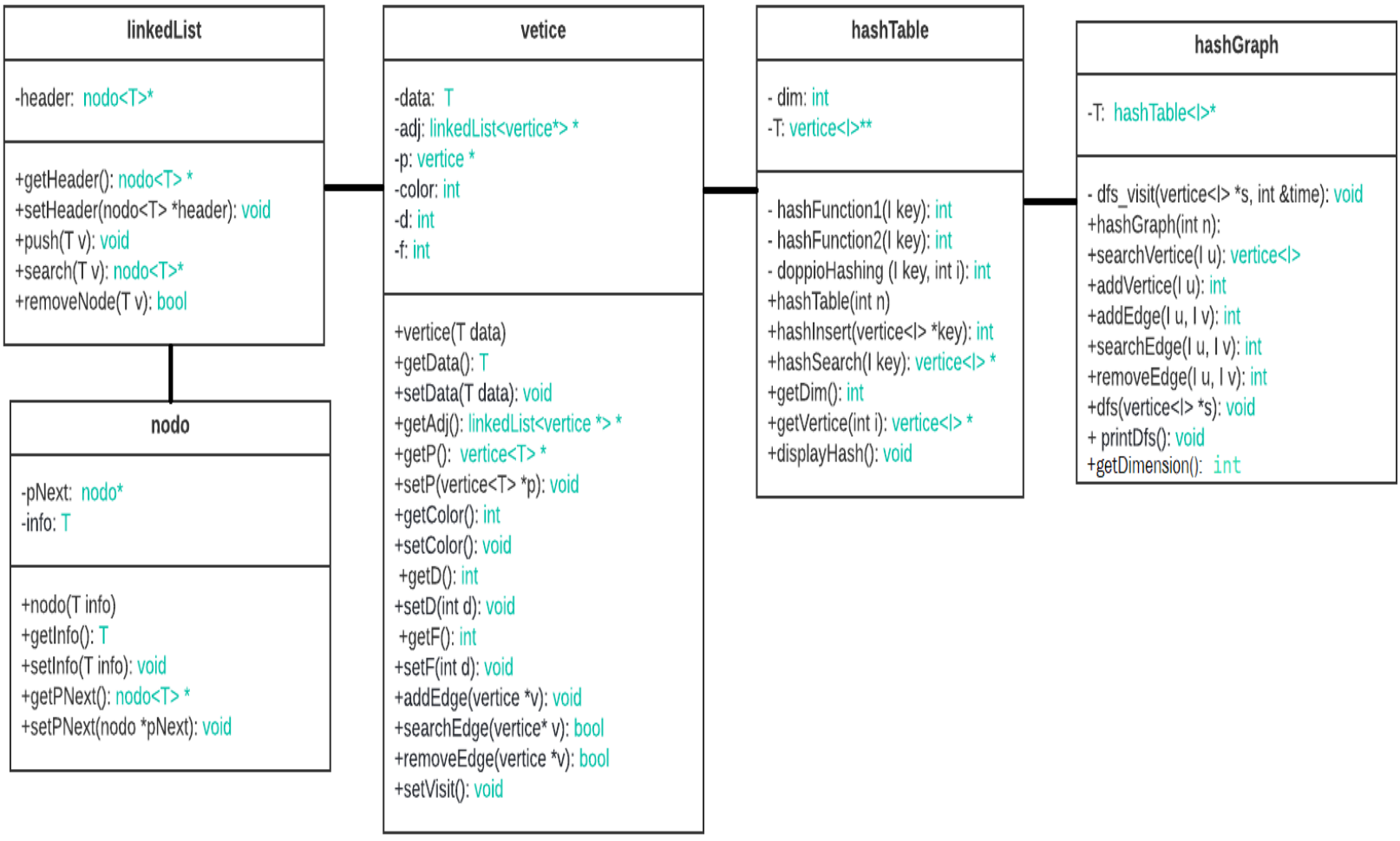
- **caso 3**: L'utente ha richiesto di cercare un arco inserendo 3.Dopo aver fatto inserire all'utente i due vertici interessati viene verificata la correttezza dell'inserimento e se i due vertici esistono nell'hashGraph. Se si ha esito negativo viene mostrato all'utente un apposito messaggio d'errore. Se si ha esito positivo viene chiamata la funzione *int hashGraph<I>::searchEdge(I u, I v)* e in base al risultato ottenuto viene mostrato all'utente se la ricerca è avvenuta con successo o se l'arco non è stato trovato perchè inesistente.

- **caso 4**: L'utente ha richiesto di effettuare la dfs inserendo 4. Dopo aver fatto inserire all'utente la sorgente interessata viene verificata la correttezza dell'inserimento

e se il vertice esiste nell'hashGraph. Se si ha esito negativo viene mostrato all'utente un apposito messaggio d'errore. Se si ha esito positivo viene chiamata la funzione *vertice<I>*searchVertice(I u)* che cerca il vertice richiesto nell'hashgraph e viene chiamata la funzione *void dfs(vertice<I>*s)* sul vertice.Infine viene chiamata la funzione *void printDfsVisit()* che mostra i risultati della visita. I vertici non raggiungibili dalla sorgente verranno mostrati segnalando che non sono stati scoperti.

- **caso 5**: L'utente ha richiesto di terminare l'esecuzione del programma inserendo 0. Si esce dalla funzione menu e si termina l'esecuzione
- **caso 6**: L'utente non ha inserito nessuna delle opzioni precedenti. Viene mostrato un messaggio d'errore e viene fatto inerire nuovamente il valore all'utente.

5 CLASS DIAGRAM



6 STUDIO COMPLESSITA'

- **int hashGraph<I>::addEdge(I u, I v);**
  La funzione chiama al suo interno altre due funzioni: *bool vertice<T>::searchEdge(vertice *v)* e *void vertice<T>::addEdge(vertice *v)*. Per la prima la complesità varia in base alla lunghezza della lista di adiacenza mentre nella seconda è O(1). Quindi nel caso peggiore avremmo che la complessità è O(V).

- **int hashGraph<I>::searchEdge(I u, I v);**
  La funzione chiama al suo interno la funzione *bool vertice<T>::searchEdge(vertice *v)*. La sua complessità varia in base alla lunghezza della lista di adiacenza. Quindi nel caso peggiore avremmo che la complessità è O(V).

- **int hashGraph<I>::removeEdge(I u, I v);**
  La funzione chiama al suo interno la funzione *bool vertice<T>::removeEdge(vertice *v)*. La sua complessità varia in base alla lunghezza della lista di adiacenza. Quindi nel caso peggiore avremmo che la complessità è O(V).

- **void hashGraph<I>::dfs(vertice<I>*s);**
  La funzione inizialmente inizializza gli attributi di visita di tutti i vertici che sono stati utilizzati in una visita precedente. Successivamente richiama la funzione *void hashGraph<I>::dfs_ visit(vertice<I>*s, int & time)*. Nel caso peggiore dunque il costo è O(V + E).

7 TEST/RISULTATI

- **Inizializzazione del file di input**:

```
1 pet inserire il path del percorso file
Un altro valore per ../../T2/Q1/input0_2_1.txt
1
Inserire il path
file.txt
Errore nell'apertura del file o il numero dei vertici non rende possibile alcuna operazione
Process finished with exit code -1
```

```
1 pet inserire il path del percorso file
Un altro valore per ../../T2/Q1/input0_2_1.txt
5
Non e' stato possibile creare l'arco 9-10. Uno o entrambi i vertici non sono presenti nell'grafo
Non e' stato possibile creare l'arco 8-10. Uno o entrambi i vertici non sono presenti nell'grafo
```

- **Creazione di un arco**:

```
Inserisci:
 '1' per addEdge
 '2' per removeEdge
 '3' per findEdge
 '4' per DFS
'0' per terminare
1
Arco (u,v)
 Inserisci u:
2
Inserisci v
3
L'arco e' stato inserito con successo
```

```
Inserisci:
 '1' per addEdge
 '2' per removeEdge
 '3' per findEdge
 '4' per DFS
'0' per terminare
1
Arco (u,v)
 Inserisci u:
2
Inserisci v
3
L'arco non e' stato inserito perche' gia' esistente
```

```
Inserisci:
 '1' per addEdge
 '2' per removeEdge
 '3' per findEdge
 '4' per DFS
'0' per terminare
1
Arco (u,v)
 Inserisci u:
43
Inserisci v
2
Uno o entrambi i vertici non esistono
```

Figure 2: Questo vale per tutte le operazioni su archi

- **Inserimento di un valore diverso da un intero (Questo esempio vale per tutte le operazioni)**:

```
Inserisci:
 '1' per addEdge
 '2' per removeEdge
 '3' per findEdge
 '4' per DFS
'0' per terminare
1
Arco (u,v)
 Inserisci u:
ret
Errore nell'input u
Inserisci:
 '1' per addEdge
 '2' per removeEdge
 '3' per findEdge
 '4' per DFS
'0' per terminare
```

- **Eliminazione arco**:

```
Inserisci:
 '1' per addEdge
 '2' per removeEdge
 '3' per findEdge
 '4' per DFS
'0' per terminare
2
Arco (u,v)
 Inserisci u:
2
Inserisci v
3
Eliminazione avvenuta con successo
```

```
Inserisci:
 '1' per addEdge
 '2' per removeEdge
 '3' per findEdge
 '4' per DFS
'0' per terminare
2
Arco (u,v)
 Inserisci u:
2
Inserisci v
3
Eliminazione non avvenuta: L'arco non esiste
```

- **Ricerca di un arco**:

```
Inserisci:
 '1' per addEdge
 '2' per removeEdge
 '3' per findEdge
 '4' per DFS
'0' per terminare
3
Arco (u,v)
 Inserisci u:
0
Inserisci v
4
L'arco esiste
```

```
Inserisci:
 '1' per addEdge
 '2' per removeEdge
 '3' per findEdge
 '4' per DFS
'0' per terminare
3
Arco (u,v)
 Inserisci u:
0
Inserisci v
7
L'arco non esiste
```

- **Dfs**:

```
Inserisci:
 '1' per addEdge
 '2' per removeEdge
 '3' per findEdge
 '4' per DFS
'0' per terminare
4
Inserisci la sorgente s:
0
Il risultato della dfs dalla sorgente stabilita e'
Vertice:0
Tempo di inizio scoperta: 0
Tempo di fine scoperta: 19
Vertice:1
Tempo di inizio scoperta: 12
Tempo di fine scoperta: 15
Vertice:2
Tempo di inizio scoperta: 1
Tempo di fine scoperta: 18
Vertice:3
Tempo di inizio scoperta: 11
Tempo di fine scoperta: 16
Tempo di fine scoperta: 16
Vertice:4
Tempo di inizio scoperta: 2
Tempo di fine scoperta: 17
Vertice:5
Tempo di inizio scoperta: 3
Tempo di fine scoperta: 10
Vertice:6
Tempo di inizio scoperta: 13
Tempo di fine scoperta: 14
Vertice:7
Tempo di inizio scoperta: 7
Tempo di fine scoperta: 8
Vertice:8
Tempo di inizio scoperta: 5
Tempo di fine scoperta: 6
Vertice:9
Tempo di inizio scoperta: 4
Tempo di fine scoperta: 9
```

```
 '1' per addEdge
 '2' per removeEdge
 '3' per findEdge
 '4' per DFS
'0' per terminare
4
Inserisci la sorgente s:
8
Il risultato della dfs dalla sorgente stabilita e'
Vertice:0 non scoperto dalla sorgente
Vertice:1 non scoperto dalla sorgente
Vertice:2 non scoperto dalla sorgente
Vertice:3 non scoperto dalla sorgente
Vertice:4 non scoperto dalla sorgente
Vertice:5 non scoperto dalla sorgente
Vertice:6 non scoperto dalla sorgente
Vertice:7 non scoperto dalla sorgente
Vertice:8
Tempo di inizio scoperta: 0
Tempo di fine scoperta: 1
Vertice:9 non scoperto dalla sorgente
```

- **Inserimento di un opzione non esistente**:

```
Inserisci:
 '1' per addEdge
 '2' per removeEdge
 '3' per findEdge
 '4' per DFS
'0' per terminare
43
Inserisci un valore valido
```

Relazione traccia 2

Quesito 2

Davide De Angelis #0124002412

1 DESCRIZIONE DEL PROBLEMA

Data una diga, si necessità di finire la costruzione della rete idrica in modo tale da far arrivare l'acqua in tutte le città. Si vuole costruire però il minor numero possibile di condotte. Va considerato che viene già dato in input il grafo delle città con le condotte già costruite. E' possibile rendere disponibile l'acqua a tutte le città con il minor numero di condotte tramite configurazioni differenti. Da ricordare che il grafo è orientato quindi una città può essere collegata alla sorgente ma la sorgente può non essere collegata alla città.

2 DESCRIZIONE STRUTTURE DATI

Per risolvere il problema dato vengono utilizzate le strutture viste nel quesito 1. Di seguito verranno riportate solo le strutture modificate.

- linkedList
  Rispetto a quella vista in precedenza viene aggiunto un attributo int lenght che contiene il numero degli elementi presenti nella linkedList. Per tenerlo aggiornato nella funzione di aggiunta viene incrementato mentre in quella di rimozione viene decrementato. In più abbiamo una funzione **int getLenght() const;** che ci restituisce lenght.

- hashGraph
  Viene utilizzata la struttura hashGraph per conservare i vantaggi relativi alle operazioni svolte in O(1) utilizzate soprattutto nella fase di inizializzazione. E' costituita sempre da un attributo hashTable<I>*T in cui verranno memorizzati i vertici con le relative liste di adiacenza. Rispetto al quesito precedente però la struttura è caratterizzata dai seguenti metodi:

  - **hashGraph(int n);**
    E' il costruttore, prende in input un intero e crea un hashtable della dimensione dell'intero.
  - **void dfs_visit(vertice<I>*s, int &time);**
    Implementazione della dfs_visit
  - **void dfs_visit2(vertice<I>*s, linkedList<vertice<I>*>*, int &time);**
    A differenza della dfs_visit, quando su un vertice viene finita la scoperta e impostato a nero, esso viene inserito in una linkedList con inserimento in testa.
  - **void setDfsVisit();**
    La funzione setta i parametri dei vertici relativi alla visita. I vertici non utilizzati in precedenti dfs (da sorgente) non vengono resettati.

- **vertice<I>*searchVertice(I u);**
  La funzione, dato un intero u, restituisce l'elemento di indice u nell'hashTable. Se questo non esiste restituisce nullptr.
- **int addVertice(I u);**
  La funzione, dato un dato u di tipo I, crea un vertice contenente u e lo inserisce nell'hashTable. Se l'inserimento è avvenuto con successo viene restituito 1. I casi particolari gestiti da questa funzione sono due. Il primo riguarda il caso in cui l'informazione che si sta cercando di inserire nel vertice esiste già in un altro vertice dell'hashGraph. In questo caso non viene creato il vertice e viene restituito -1. Il secondo è quando l'hashTable T è piena. In questo caso il vertice viene creato per provare l'inserimento ma visto l'insuccesso dell'operazione viene deallocato e viene restituito 0.
- **int addEdge(I u, I v);**
  La funzione aggiunge un arco al grafo. La funzione inizialmente crea due puntatori a vertice che contengono rispettivamente il risultato della ricerca dei due vertici richiesti. Se questi esistono e non esiste già l'arco, il vertice contenente v viene inserito nella lista di adiacenza del vertice contente u e viene restituito 1 per segnalare il successo dell'operazione. Se invece esiste già l'arco, quindi il vertice contenente v è già presente nella lista di adiacenza del vertice contenente u, viene restituito 0. Se invece dalla ricerca iniziale uno dei due vertici risulta non esistere viene restituito -1.
- **linkedList <vertice<I>*>*componentiConnesse();**
  La funzione restituisce una linkedList contenente il minor numero di vertici i quali se collegati alla sorgente stabilita permettono l'esplorazione dell'intero grafo partendo da essa. La funzione, dopo aver settato i parametri di visita dei vertici, manda una dfs solamente sulla sorgente. Successivamente viene effettuato un ordinamento topologico sui vertici utilizzando però una linkedList con inserimento in testa. In questo modo nella linkedList avremo tutti i vertici non raggiungibili dalla sorgente. Inoltre, grazie all'ordinamento topologico, sappiamo che i vertici che troviamo prima nella linkedList permettono di arrivare a quelli dopo nella visita se collegati. A questo punto, dopo aver resettato i valori di visita dei vertici nella linkedList, viene mandata una dfs su di loro. Quando però nel cambio sorgente quest'ultima è gia stata scoperta, viene eliminata dalla linkedList. In questo modo nella linkedList avremmo il minor numero di vertici i qual se collegati alla sorgente stabilita permettono l'esplorazione dell'intero grafo partendo da essa.

3 FORMATO DATI IN INPUT/OUTPUT

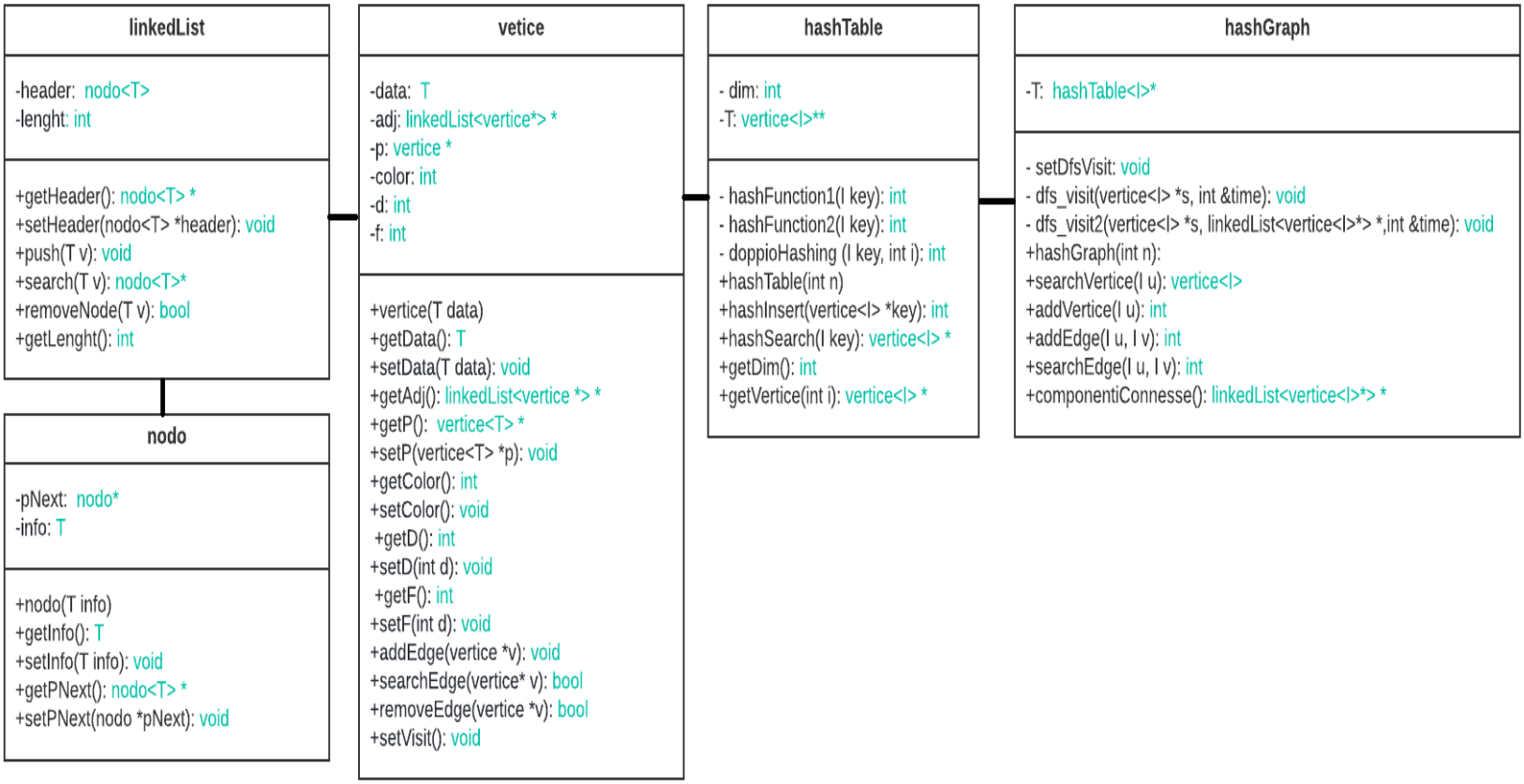
Il file di input dato contiene nel primo rigo due numeri interi, 1<=N<=1000 e 0<=P<=1000 , separati da uno spazio e rappresentano rispettivamente il numero di città ed il numero

di condotte idriche. I successivi P righi contengono due numeri interi separati da uno spazio che segnalano la presenza di una condotta idrica dalla prima alla seconda città. Stabiliamo a priori che prima di leggere le condotte inizializziamo tutti i vertici da 0 ad N inserendoli nell'hashGraph(0 è la diga). Questo perchè i vertici mostrati negli M righi vanno da minimo 1 ad N ma potrebbe esserci un vertice senza archi che quindi non è mostrato negli M righi. **Grazie a questo formato input è possibile verificare se un vertice dato è maggiore o uguale alle dimensioni dell'hashGraph. In questo caso non viene effettuata nessuna operazione in quanto sicuramente non esiste in esso(Così facendo il costo delle operazioni sull'hashTable resteranno O(1))**. In output, oltre alla scelta riguardante il cambio o meno del path del file di input, viene mostrato il numero minimo e le relative condotte da costruire.

4 DESCRIZIONE ALGORITMO

Il programma parte creando un hashGraph tramite la funzione *hashGraph<int>*inizializza()*. Questa funzione innanzitutto chiede all'utente se i dati riguardanti il grafo devono essere letti dal path predefinito o bisogna cambiarlo. Dopo che l'utente ha effettuato la scelta viene letta la dimesnione dell'hashGraph (Se il file esiste) e vengono creati i vertici con la funzione int *hashGraph<I>::addVertice(I u)*. Successivamente vengono letti gli archi ed inseriti con la funzione *int hashGraph<I>::addEdge(I u, I v)* mostrando all'utente eventuali errori. Una volta terminata questa funzione e restituito l'hashGraph creato, viene controllato se esiste o si è avuto un errore nell'apertura del file. In caso di errore viene mostrando in output all'utente un messaggio d'errore. Successivamente viene effettuata la chiamata alla funzione *linkedList<vertice<I>*>*hashGraph<I>::componentiConnesse()* la cui linkedList restituita viene opportunamente salvata e passata alla funzione *void printArchiDaCostruire (linkedList<vertice<int>*>*compConnesse, vertice<int>*s)*. Quest'ultima utilizza le dimensioni della lista per restituire il numero minimo di condotte da costruire ed i vertici all'interno della lista per mostrare quali condotte devono essere costruite.

5 CLASS DIAGRAM



6 STUDIO COMPLESSITA'

- **linkedList<vertice<I>*>*hashGraph<I>::componentiConnesse()**
  La funzione chiama prima la *void hashGraph<I>::setDfsVisit()*, successivamente la *void hashGraph<I>::dfs_ visit(vertice<I>*s, int & time)* solo sulla sorgente predefinita e poi la *void dfs_visit2(vertice<I>*s, linkedList<vertice<I>*>*, int &time)* sui vertici non scoperti. Queste operazioni risultano essere uguali al costo di una dfs quindi O(V + E). Infine dal reset dei valori di visita dei vertici della linked list e dalla nuova chiamata alla *void hashGraph<I>::dfs_ visit(vertice<I>*s, int & time)* abbiamo nel caso peggiore nuovamente un costo di O(V + E). Dunque il costo finale è O(V + E)

7 TEST/RISULTATI

- **Inizializzazione del file di input**:

```
1 pet inserire il path del percorso file
Un altro valore per ../../T2/Q2/input0_2_2.txt
1
Inserire il path
cartelle/input.txt
Errore nell'apertura del file
Process finished with exit code -1
```

```
1 pet inserire il path del percorso file
Un altro valore per ../../T2/Q2/input0_2_2.txt
5
Le condotte da costruire sono : 14
0-23
0-22
0-21
0-20
0-19
0-18
0-17
0-16
0-10
0-9
0-8
0-7
0-4
0-2
```