



UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

*Appunti di
Laboratorio di Algoritmi e Strutture dati*

Anno 2021

Valentino Bocchetti

Contents

1	Introduzione al corso	8
2	Argomenti del corso	8
3	Gestione del corso (prove intercorso)	8
4	Esame	9
4.1	Esame semplificato e completo	9
5	Conoscenze di base	9
5.1	Le basi	9
5.2	Compilazione	9
5.3	Linking	9
5.4	Modularità	10
5.5	Scopo degli header file	10
5.6	Header guards	10
5.7	Puntatori (<code>tipo* variabile</code>)	10
5.7.1	Puntatori a void	10
5.7.2	Riferimenti (<code>tipo& variabile</code>)	11
5.7.3	<code>std::move()</code>	11
5.7.4	Puntatore a funzione	11
5.8	Stack	11
5.9	Heap	11
5.10	Memory leak	11
5.11	Operatori <code>new</code> , <code>delete</code> , no naked <code>new</code> and <code>delete</code>	12

5.11.1	New	12
5.11.2	Delete	12
5.11.3	No naked new & delete	12
5.12	Tipo string	12
6	Lezione del 11-03	12
6.1	Struttura modulare di un programma	12
6.1.1	Struttura di un progetto	13
7	Lezione del 12-03	14
7.1	Tipi fondamentali, Puntatori e Riferimenti	14
7.1.1	Dichiarazione di variabili	14
7.1.2	Tipi fondamentali in c++	14
7.1.3	Tipi const	15
7.1.4	Puntatori	15
7.1.5	Void pointer	16
7.1.6	Riferimenti	16
7.1.7	Lvalue & Rvalue	18
8	Lezione del 18-03	18
8.1	Allocazione dinamica della memoria	18
8.1.1	Esempio per la locazione	20
8.2	Tipi definiti dall'utente	21
8.3	Le enumeration	22
8.4	Libreria iostream & funzioni di I/O	23
8.4.1	Classe istream	23

8.4.2 Classi ostream	23
9 Lezione del 19-03	24
9.1 Libreria string	24
9.2 Generazione Pseudo-Casuale di numeri	25
9.3 Le eccezioni	26
9.3.1 Esempio	27
9.4 Puntatori a funzione	28
9.4.1 Definizione	28
9.4.2 Chiamata a funzione	28
9.4.3 Esempio con il quick-sort	29
10 Lezione del 25-03	32
10.1 Classi e Oggetti	32
10.1.1 Classi	32
10.1.2 Costruttori	32
10.1.3 Distruttore	32
10.1.4 Operatori (funzioni membro)	32
10.2 Sintassi della classe	33
10.3 Sintassi dei Metodi	33
10.3.1 Definizione concreta di un metodo	33
10.4 Ereditarietà di tipo virtuale	34
10.5 Esempio	34
11 Lezione del 26-03	34
11.1 Template	34

11.1.1	Vector.hpp	35
11.1.2	Vector.cpp	35
11.1.3	main	35
11.2	Differenze e similitudini con i linguaggi C e Java	36
12	Lezione 08-04 09-04	36
13	Lezione del 15-04	36
13.1	Gli alberi binari	36
13.2	Albero binario di ricerca	38
14	Lezione del 16-04	38
14.1	Gli iteratori	38
14.1.1	Implementazione degli iteratori sugli alberi	38
14.2	Tipi di iteratori concreti	39
15	Lezione del 22-04	39
15.1	Visite negli alberi binari	39
16	Lezione del 23-04	40
17	Lezione del 29-04	40
17.1	Costruzione di un ABR dato un Linear Container	41
17.2	Costruttori di copia e spostamento	41
17.3	Operatori di confronto	41
17.4	Exists	42
17.5	Insert/Remove	42

17.6 Max/Min (Get/Remove)	42
17.7 Predecessor/Successor (Get/Remove)	42
18 Lezione del 30-04	42
18.1 Inserimento e Remove negli alberi	42
18.2 Funzione Detach	42
18.3 Predecessor & Successor	42
19 Lezione del 06-05	43
19.1 Matrice	43
19.1.1 Rappresentazione Vettoriale	44
19.1.2 Rappresentazione a lista (lineare)	45
19.1.3 Rappresentazione Compress Row/Column (CSR/CSC)	45
20 Lezione del 07-05	46
21 Lezione del 13-05	46
21.1 Recap su CSR	46
22 Lezione del 20-05	48
22.1 Liste ortogonali	48
23 Lezione del 21-05	51
24 Lezione del 27-05	51
24.1 Definizione	51
24.2 Rappresentazione	52
24.3 Rappresentazione dei grafi etichettati	53

24.4 Algoritmi di visita	53
24.4.1 Visita in ampiezza (la classica map)	53
25 Lezione del 28-05	54
25.1 Visita in profondità (Pre/Post)	55
25.2 Iteratore su Grafo un Ampiezza	55
25.3 Operatore ++()	56
26 Lezione del 03-06	57
26.1 Costrutture e Successor (Iteratori)	57
26.2 Iteratore in Post-Order	59
27 Lezione del 04-06	60
27.1 Acyclicity Test	60
27.1.1 Definizione	60
27.1.2 Idea dell'algoritmo	62
27.1.3 Rappresentazione concreta e ragionamenti	62
27.2 Topological Ordering	62
27.2.1 Algoritmo con Grado entrante	63
27.2.2 DFS	63
27.2.3 Costruttore	64
27.2.4 SuccessorOperator (++)	64
27.3 Iteratori ONLINE/OFFLINE	64
27.4 MEMO	65

1 Introduzione al corso

Ricevimento → Lunedì ore 16-18 (Teams).

Email → fabio.mogavero@unina.it

Link al download del libro di Data Structure & Algorithm Analysis → [clicca per il download](#)

2 Argomenti del corso

- Elementi di Linguaggio C++;
- Tipi di dato astratto e relative implementazioni in C++;
- Progettazione di una libreria contenitore di dati;
- Strutture dati elementari (vettori, liste, pile, code);
- Alberi binari di ricerca e iteratori sui dati;
- Matrici e grafi.

3 Gestione del corso (prove intercorso)

- Esercizio Vettori e liste (26 Marzo - 11 Aprile);
- Esercizio Pile e Code (9 Aprile - 25 Aprile);
- Esercizio Alberi Binari di Ricerca (23 Aprile - 9 Maggio);
- Esercizio Iteratori su Alberi (7 Maggio - 23 Maggio);
- Esercizio su Matrici (21 Maggio - 6 Giugno).

Per la consegna:

- Invio file .zip o tar.gz alla mail del prof con oggetto "Consegna Esercizio *x* (Cognome-Nome-Matricola)"
- Formato File: Cognome-Nome-Matricola(Esercizio *x*).zip *o* Cognome-Nome-Matricola.tar.gz
- Scadenza improrogabili → ore 23:59 dell'ultimo giorno previsto per l'esercizio

4 Esame

Esame in presenza:

- Prova al calcolatore + discussione orale

4.1 Esame semplificato e completo

Per chi consegna tutti gli elaborati si ha la possibilità di accedere all'esame semplificato (reimplementazioni, implementazione di una vecchia/nuova funzionalità di una libreria tra quelle implementate).

Per chi non consegna tutti gli elaborati si è obbligati ad accedere all'esame completo (implementazione da zero di una libreria).

Prerogative per entrambe le modalità è l'obbligo della consegna di tutte le librerie con data ultima precedente alla data di esame a cui si vuole accedere.

Nel caso di modalità telematica nel caso dell'esame semplificato si svolgerà solo la parte orale.

5 Conoscenze di base

Grazie a Umberto de Angelis e Simone Cerrone

5.1 Le basi

Ogni programma in C++ ha esattamente una funzione globale chiamata `main()` di tipo intero che fallisce ritorna un valore diverso da 0.

5.2 Compilazione

A differenza dei linguaggi interpretati, la traduzione avviene una volta per tutte e non a *runtime*.

2 modalità:

- Classico build.sh (lento perchè ripete ogni volta la compilazione di tutti i file, e i tempi quindi aumentano rapidamente);
- Makefile (molto comodo, in quanto ricompila solo le cose che sono variate, rendendo la seconda compilazione molto veloce rispetto alla prima).

5.3 Linking

Si occupa del collegamento tra i file oggetto e i file di libreria.

5.4 Modularità

Generalmente è impensabile creare un programma in un singolo file, per cui viene strutturato in più file di diverse estensioni divise in file .hpp (header file) ed in file .cpp (implementation file) che vengono compilati per creare il file oggetto.

Si assume che ogni header file che viene incluso come prototipo (la firma della funzione) nelle altre librerie inizi e termini con le macro

```
#ifndef TEST_HPP
#define TEST_HPP

// Corpo della macro

#endif
```

5.5 Scopo degli header file

File che aiutano il programmatore nell'utilizzo di librerie durante la programmazione.

Contiene i prototipi delle funzioni definite nel relativo file .cpp

5.6 Header guards

Particolari direttive (macro) che vengono usate nei file header per evitare problemi di doppia definizione in fase di linking (risolte dal pre-processore).

5.7 Puntatori (**tipo*** **variabile**)

Tipi di dato che rappresentano la posizione (utilizzando indirizzi di memoria) di elementi come:

- Variabili;
- Oggetti;
- Strutture dati;
- *etc* ...

Il puntatore `nullptr` non punterà a nessun oggetto.

5.7.1 Puntatori a void

Possono puntare a qualsiasi tipo di variabile, nativa o meno, ma non potranno essere dereferenziati.

Per farlo sarà necessario una conversione esplicita tramite `cast`.

5.7.2 Riferimenti (tipo& variabile)

Variabili che occupano la stessa memoria delle variabili a cui si riferiscono.

Questi oggetti hanno *accesso diretto* alla variabile assegnata, quindi, qualsiasi modifica effettuata tramite riferimento modificherà il valore originale della variabile assegnata.

5.7.3 std::move()

Funzione della standard library che effettua una trasformazione del tipo di valore da un **RValue** ad un **LValue**.

5.7.4 Puntatore a funzione

È un tipo di puntatore che quando viene deferenziato, invoca una funzione passando 0 o più argomenti come ad una funzione normale.

I puntatori a funzione possono essere usati per semplificare il codice fornendo un modo semplice per eseguire codice di base a parametri determinati a **run-time**.

Essi, inoltre, permettono la chiamata di funzioni diverse con lo stesso prototipo conoscendone l'indirizzo

5.8 Stack

L'area di memoria **stack** (memoria fissa in base all'*OS*) è quella in cui viene allocato un pacchetto di dati non appena l'esecuzione passa dal programma chiamante a una funzione.

Questo pacchetto viene *impilato* sopra il pacchetto precedente e poi automaticamente rimosso dalla memoria appena l'esecuzione della funzione è terminata.

5.9 Heap

L'area di memoria è soggetta a regole di visibilità e tempo di vita diverse dallo stack, e precisamente:

- L'area *heap* non è allocata automaticamente, ma può essere allocata o rimossa solo su esplicita richiesta del programma;
- Il suo scope coincide con quello del puntatore che contiene il suo indirizzo;
- Il suo lifetime coincide con l'intera durata del programma a meno che non venga esplicitamente deallocated.

5.10 Memory leak

Se il puntatore va *out of scope*, l'area non è più accessibile, ma continua a occupare memoria inutilmente, ciò potrebbe verificare l'errore di memory leak.

5.11 Operatori `new`, `delete`, `no naked new and delete`

5.11.1 New

Costruisce uno o più oggetti nell'area heap e ne restituisce l'indirizzo.

In caso di memoria non disponibile (*bad-alloc*) non restituirà `nullptr`.

5.11.2 Delete

Dealloca la memoria dell'area heap puntata dall'operando.

Non restituisce alcun valore e quindi deve essere usato da solo in un'istruzione.

Questo operatore non cancella il puntatore né altera il suo contenuto, l'unico effetto è di liberare la memoria puntata rendendola disponibile.

L'operatore `delete` costituisce l'unico mezzo per deallocare memoria head.

5.11.3 No naked new & delete

È consigliato effettuare allocazione di memoria all'interno delle classi

5.12 Tipo string

In C++ non esiste un vero e proprio tipo stringa (viene vista come una sequenza di caratteri contenuta in uno spazio in memoria, il *buffer*).

6 Lezione del 11-03

6.1 Struttura modulare di un programma

Primo esempio (main.cpp)

```
#include <iostream> // libreria standard c++ I/O
// dichiarazione dello spazio dei tipi
//(accesso alle funzionalità senza la specificazione "std::")
using namespace std;
int main(){
    cout << "Hello World!" << endl;
}
```

Di seguito invece il `build.sh` (script bash)

```
#!/bin/bash  
g++ [-O3] -o main main.cpp
```

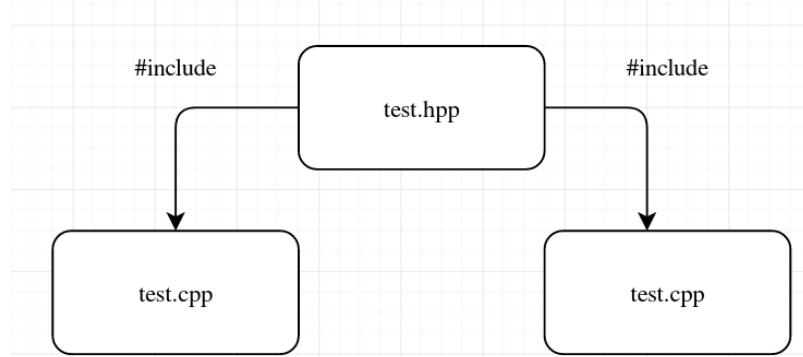
Dove:

- g++ → compilatore GNU;
- [-O3] → parametri di ottimizzazione opzionale;
- -o main → output/file eseguibile;
- main.cpp → codice sorgente

Nel momento in cui ci siano più file da compilare vanno aggiunti manualmente (l'ordine è indeferente)

6.1.1 Struttura di un progetto

Nel caso di più file si avrà un file principale più altri che eseguiranno effettivamente le operazioni:



test.hpp → file header. Si presuppone che abbia delle macro iniziali e finali che vengano lette dal precompilatore

```
// main.cpp  
#include <iostream>  
#include "test.hpp"  
using namespace std;  
int main(){  
    cout << "Chiamata a funzione";  
    test();  
}
```

```

// test.hpp
// Le macro vengono risolte dal preprocessore
#ifndef TEST_HPP
#define TEST_HPP
void test();
#endif

// test.cpp
#include <iostream>
#include "test.hpp"
void test(){
    std::cout<<"OK!"<< std::endl;
}

```

7 Lezione del 12-03

7.1 Tipi fondamentali, Puntatori e Riferimenti

7.1.1 Dichiarazione di variabili

type varname [= default value]

type varname [numero elementi] [{ val₀, ..., val_n }]

7.1.2 Tipi fondamentali in c++

- Bool (true/false);
- [unsigned] char;
- [unsigned] short;
- [unsigned] int;
- [unsigned] long int;
- [unsigned] long long int;
- float;
- double;
- long double

In alcuni compilatori è possibile delle abbreviazioni per tipi unsigned int e long del tipo:

- uint;
- ulong;

7.1.3 Tipi const

```
const type varname = value;
```

Ritornano utili anche nel passaggio dei parametri nelle chiamate a funzione.

7.1.4 Puntatori

```
type* varname [=default address]
```

es.

```
#include <iostream>
using namespace std;
int main(){
    char c = 'a';
    char* p = &c;
    cout << "Il valore di *p è " << *p << endl;
    p= nullptr; //corrisponde alla macro NULL del linguaggio C
    cout << p << endl;
}
```

Il valore di *p è a

```
#include <iostream>
using namespace std;
int main(){
    const char c = 'a';
    char* p = &c; // ERRORE
    const char* p = &c; // CORRETTO
    cout << *p << endl;
}
```

7.1.5 Void pointer

```
void* varname [= ...]
```

Un void pointer è un puntatore che non è associato a nessun tipo specifico. Puo mantenere un indirizzo di ogni tipo e può essere typecasted a ogni tipo.

Indefinire un puntatore in cui non è noto a priori il tipo di dato a cui si vuole puntare.

```
#include <iostream>
using namespace std;
int main(){
    char c = 'a';
    void* p = &c;
    //cout << *p << endl; // ERRORE
    cout << *((char*)p) << endl; // CORRETTO
    cout << *(static_cast<char*>(p)) << endl; // MODO MIGLIORE
}
```

7.1.6 Riferimenti

```
type & varname = ...
```

```
#include <iostream>
using namespace std;
int main(){
    uint d = 7;
    uint &e = d;
    e++;
    cout << "Il valore di d è " << d;
}
// Stamperà il valore 8
```

```
#include <iostream>
using namespace std;
int main(){
    uint d = 7;
    uint e = d;
    e++;
    cout << "Il valore di d è " << d;
}
// Stamperà il valore 7
```

Con i puntatori invece:

```
#include <iostream>
using namespace std;
int main(){
    uint d = 7;
    uint* e = &d;
    (*e)++;
    cout << "Il valore di d è " << d;
}
```

Il valore di d è 8

```
int & i = 1; // ERRORE
const int & i = 1; // CORRETTO
```

7.1.7 Lvalue & Rvalue

Differenze

In C++ un *Lvalue* è qualcosa che punta a una specifica locazione di memoria. Un *Rvalue* invece è qualcosa che non punta da nessuna parte.

In generale gli Rvalues sono temporanei e di vita breve, mentre gli Lvalues hanno una vita lunga tanto quanto quella della variabile.

```
#include <iostream>
using namespace std;
string f(){return string ("Ciao");}
int main(){
    string var = f();
    string stringa1 = move(f());
    string && stringa2 = f(); // Riferimento Rvalue
    string stringa3 = move(stringa2); // Spostiamo il valore di stringa2 in stringa3 senza copiarla
    cout << "var: " << var << endl;
    cout << "stringa1: " << stringa1 << endl;
    cout << "stringa2: " << stringa2 << endl;
    cout << "stringa3: " << stringa3;
}
```

```
var:      Ciao
stringa1: Ciao
stringa2:
stringa3: Ciao
```

8 Lezione del 18-03

8.1 Allocazione dinamica della memoria

Dal linguaggio C in C++:

- *malloc()* → *new*;
- *free()* → *delete*;
- *realloc* invece non ha diretta corrispondenza in C++
- **new** solleva un'eccezione;

- **new** non restituisce *nullptr* quando fallisce ("bad-alloc");
- L'uso di **new/delete** non va combinato con chiamate a funzioni **malloc/realloc/free**;
- No naked "new" "delete".
- Non considerano gli smart-pointer;

```
uint num = espressione;
type* var = new type[num]; //dimensione array
delete[] var;
// Allocazione di tipo unico
// Tra le parentesi il potenziale del valore di default
ObjectType* var = new ObjectType();
...
delete var;
```

8.1.1 Esempio per la locazione

```
#include <iostream>
using namespace std;
int main(){
    try {
        // ulong* ulptr = new ulong; Non inizializzato
        ulong* ulptr = new ulong(5); // Inizializzato
        cout << (*ulptr)++ << endl;
        cout << *ulptr << endl;
        //delete ulptr;
        delete ulptr;
    }catch (bad_alloc exc){
        cout << "Quite rare exception !" << endl;
    }
    try{
        const ulong arraysize = 1000000000000000; // Ottengo il cout della catch
        ulong* ulptr = new ulong[arraysize];

        cout << ulptr[0]++ << endl;
        cout << ulptr[1] << endl;
        cout << ulptr[2]-- << endl;
        cout << ulptr[0] << endl;
        cout << ulptr[1] << endl;
        cout << ulptr[2] << endl;
        delete[] ulptr;

    }catch(bad_alloc exc){
        cout << "Quite rare exception !" << endl;
    }
}
```

Nel caso in cui si chiami una delete due volte di seguito si andrà in contro a `free(): double free detected in tcache 2`. Per evitare ciò un workaround è quello di settare la variabile a `nullptr`

8.2 Tipi definiti dall'utente

Le struct sono classi con accesso pubblico agli attributi e alle funzioni membro; nessuna altra restrizione.

```
struct strudente{ campi };
```

Un esempio di struct:

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct Studente{
    string Nome;
    string Cognome;
    string Matricola;
    uint id;
};

int main(){
}
```

Member access → StructNomeVal.Campo.

Accesso via pointer → poitnerVarName -> campo (equivalente ad (*PointerVarNome).Campo)

```
#include <iostream>
using namespace std;
struct Studente{
    uint Id;
    string Matricola = "N86000";
    string Nome = "Alan";
    string Cognome = "Turing";
    // Creo un costruttore con 0 o tutti gli argomenti
    Studente() = default;
    Studente(ulong id, string matr, string nome, string cognome){
        Id = id;
        Matricola = matr;
        Nome = nome;
        Cognome = cognome;
    }
}
```

```

    }
};

int main(){
    Studente stu;
    cout << "Nome: " << stu.Nome << endl;

    // Crea un costruttore che rimpiazza i valori definiti
    Studente stud1{2, "N860001", "Gennaro", "Esposito"};
    cout << "Nome: " << stud1.Nome << endl;
    // Necessita del costruttore sopra definito
    Studente stud2(3, "N860002", "Alonzo", "Church");
    cout << "Nome: " << stud2.Nome << endl;
}

```

8.3 Le enumeration

```

enum class Nome{elemento, ..., elementoN}; // In C++
enum Nome {...};
enum class Color{white, gray, black};
enum class Color3{white, gray, black};
enum class Color4{white, gray, black};
Colore colore = Color::white;
Color4 colore4 = Color4::white; // Corretto
Color4 colore4 = Color::white; // ERRATO!

```

In C non è possibile definire due enum con campi uguali (gli elementi sono visibili a livello globale). Gli enum possono essere confrontati e assegnati. Le variabili di tipo enumerativo possono ovviamente avere un valore iniziale di default

```

#include <iostream>
using namespace std;
enum class Colore {Bianco, Grigio, Rosso}; // C++
enum class Colore1 {Grigio, Rosso}; // C++
enum Colore2{Rosso, Giallo};
enum Colore3{Blu, Verde};
int main(){
    Colore color = Colore::Grigio;
    cout << (color < Colore::Bianco) << endl;
    cout << (color == Colore::Grigio) << endl;
}

```

Il confronto tra due enumeration diverse non è possibile (non è definito)

8.4 Libreria iostream & funzioni di I/O

Operatori principali » (get-from) e « (put-to). Standard stream:

- cout, cerr, clog;
- cin.

8.4.1 Classe istream

Analisi dello stato:

- *good()*;
- *eaf()*;
- *fail()*;
- *bad()*;
- *get(c) → (char)*;
- *getline(p,n) → (char + numero di caratteri da leggere)*;

8.4.2 Classi ostream

- *put(c)*;
- *write(p,n) → (char + numero di caratteri da scrivere)*;

```
#include <iostream>
using namespace std;
struct Studente{
    uint Id;
    string Matricola = "N86000";
    string Nome = "Alan";
    string Cognome = "Turing";
    friend ostream& operator<<(ostream& outstr, const Studente& stu);
    friend istream& operator>>(istream& instr, Studente& stu);
private:
    ulong SecureNum = 0x232523;
};

// Per la stampa dei dati
```

```

ostream& operator<<(ostream& outstr, const Studente& stu){
    outstr << "Id: " << stu.Id << " Matricola: "
    << stu.Matricola << " Nome: "
    << stu.Nome << " Cognome:" << stu.Cognome;
}

// Per l'inserimento dei dati
istream& operator>>(istream& instr, Studente& stu){
    instr >> stu.Id >> stu.Matricola >> stu.Nome >> stu.Cognome;
    instr >> stu.SecureNum;
}

int main(){
    Studente stud;
    cout << stud << endl;
}

```

9 Lezione del 19-03

9.1 Libreria string

- `string` (`#include <string>`) diverso da string literals C e C++ (`char*`);
- `string nome = "Alan";`
 - uguale a `string nome("Alan");`
 - uguale a `string nome = {"Alan"}`
- Gli oggetti di tipo `string` sono confrontabili lessico-graficamente;
- Operatori di `input/output` → `<<`, `>>` (via overloading);
- Presenta i metodi di
 - `size();`
 - `empty();`
 - `[i]` (prendere l'iesimo elemento della stringa);
 - `front();`
 - `back();`
- La concatenazione avviene con l'operatore `+`;
 - `string var = var + var2;`
 - `string var += var1;`
- `var.substr(i,n)` → sottocorpo da posizione `i` a posizione `i + n - 1` ;

es

```

#include <iostream>
#include <string> //opzionale nel caso si usi iostream

using namespace std;

int main(){
    string Stringa1 = "Alan Turing";
    string Stringa2 = "Kurt Godel";

    cout << Stringa1 << " - " << Stringa2 << endl;

    cout << "Lexicographic comparison (Stringa1 < Stringa2)" << (Stringa1 < Stringa2) << endl;

    cout << "Lexicographic comparison (Stringa1 > Stringa2)" << (Stringa1 > Stringa2) << endl;
    cout << "Lexicographic comparison (Stringa1 == Stringa2)" << (Stringa1 == Stringa2) << endl;
    cout << "Lexicographic comparison (Stringa1 != Stringa2)" << (Stringa1 != Stringa2) << endl;

    //cin >> Stringa2; // prende soltato tutti i caratteri fino inserimento di uno spazio / tab
    //getline(cin , Stringa2); // Per prendere tutto

    cout << "Size: " << Stringa1.size() << endl;
    cout << "È vuota? " << Stringa1.empty() << endl;
    cout << "Primo carattere: " << Stringa1.front() << endl;
    cout << "Ultimo carattere: " << Stringa1.back() << endl;
    cout << "Concatenazione: " << (Stringa1 + " " + Stringa2) << endl;

    Stringa1.clear(); // Svuoto la stringa
}

```

9.2 Generazione Pseudo-Casuale di numeri

In C

```

#include <stdlib.h>
#include <time.h>
int main(){
    /* Nel caso in cui non venga settato il seed
    si avrà sempre la stessa sequenza */
    srand(time(NULL));
    for(uint i = 0; i < 15; i++){

```

```

printf("%d\n", rand());
}
}

```

In C++

```

#include <iostream>
#include <random>
using namespace std;
int main(){
default_random_engine gen (random_device{}());
uniform_int_distribution<uint> dist(7,35);

for(uint i = 0; i < 15; i++){
cout << dist(gen) << endl;
}
}

```

9.3 Le eccezioni

Struttura di gestione elementare

```

try{
    throw SomeException();
} catch(SomeException exe){
//gestione dell'eccezione
} catch(...){
throw
} // Risolleva l'eccezione

```

type f(parameters) noexcept → la funzione non solleva alcune eccezione. Eccezioni utili della standard library:

- `logic_error`;
 - `length_error`;
 - `out_of_range`;
- `runtime_error`;
 - `overflow_error`;
 - `underflow_error`;

- `bad_alloc`;

In C++ non è previsto il `finally` → dipende dalla gestione della memoria

9.3.1 Esempio

```
#include <iostream>
using namespace std;

int main() {

    cout << "Hello everyone... I am rising some exception!" << endl;

    try {

        throw logic_error("Some logic error.");
        throw length_error("A lenght error occurred!");
        throw out_of_range("An out-of-range access to some structure occurred!");

        throw runtime_error("Some runtime error.");
        throw overflow_error("An overflow occurred!");
        throw underflow_error("An underflow occurred!");

        throw bad_alloc(); // Just an example, do not throw this exception.

        throw 25;

    } catch (length_error& exc) {

        cout << "Length error: " << exc.what() << endl;

        // throw;

    } catch (logic_error& exc) {

        cout << "Logic error: " << exc.what() << endl;

    } catch (overflow_error& exc) {

        cout << "Overflow error: " << exc.what() << endl;

    }
}
```

```

} catch (runtime_error& exc) {

    cout << "Runtime error: " << exc.what() << endl;

} catch (bad_alloc& exc) {

    cout << "Bad allocation: " << exc.what() << endl;

} catch (exception& exc) {

    cout << "All unmanaged standard exceptions reach this point!" << exc.what() << endl;

}

catch (...) {

    cout << "All unmanaged non-standard exceptions reach this point!" << endl;
}
}

```

9.4 Puntatori a funzione

Possibili usi:

- Passaggio di funzione come parametro di altre funzioni;
- Funzioni di callback;

9.4.1 Definizione

In C:

```
typedef type (*FuncName)(lista parametri);
```

In C++:

```
typedef std::function <type(lista parametri)> FuncName;
```

Richiede #include <functional>

9.4.2 Chiamata a funzione

*FuncName(...) o FuncName(...)

9.4.3 Esempio con il quick-sort

```
#include <iostream>
#include <functional>

using namespace std;

void quicksort(int*, uint);
void quicksort(int*, uint, uint);
uint partition(int*, uint, uint);
enum class ComparisonType { LessThan, Equal, GreaterThen};

// // Function pointer a la C
// typedef ComparisonType (*CompareFunction) (int, int);

// Function pointer a la C++
typedef function<ComparisonType(int, int)> CompareFunction;

void quicksort(int*, uint, CompareFunction);
void quicksort(int*, uint, uint, CompareFunction);
uint partition(int*, uint, uint, CompareFunction);

ComparisonType OrdA(int a, int b) {
    if (a < b) {
        return ComparisonType::LessThan;
    } else if (a > b) {
        return ComparisonType::GreaterThen;
    }
    return ComparisonType::Equal;
}

ComparisonType OrdB(int a, int b) {
    if (a < b) {
        return ComparisonType::GreaterThen;
    } else if (a > b) {
        return ComparisonType::LessThan;
    }
    return ComparisonType::Equal;
}
```

```

ComparisonType OrdC(int a, int b) {
    if (a % 2 != b % 2) {
        return ((a % 2 == 0) ? ComparisonType::LessThan : ComparisonType::GreaterThen);
    } else if (a < b) {
        return ComparisonType::LessThan;
    } else if (a > b) {
        return ComparisonType::GreaterThen;
    }
    return ComparisonType::Equal;
}

int main() {

    int A[11] = {5, 7, 6, 8, 4, 9, 3, 10, 2, 0, 1};

    quicksort(A, 11);

    quicksort(A, 11, OrdA);
    quicksort(A, 11, OrdB);
    quicksort(A, 11, OrdC);

    for (uint i = 0; i < 11; i++) { cout << A[i] << ' '; }; cout << endl;

    return 0;
}

void quicksort(int* A, uint size) {
    quicksort(A, 0, size - 1);
}

void quicksort(int* A, uint p, uint r) {
    if (p < r) {
        uint q = partition(A, p, r);
        quicksort(A, p, q);
        quicksort(A, q + 1, r);
    }
}

uint partition(int* A, uint p, uint r) {

```

```

int x = A[p];
int i = p - 1;
int j = r + 1;

do {
    do { j--; } while ( x < A[j] );
    do { i++; } while ( A[i] < x );
    if (i < j) { swap(A[i], A[j]); } // "swap" is standard-library function
} while (i < j);

return j;
}

void quicksort(int* A, uint size, CompareFunction cmp) {
    quicksort(A, 0, size - 1, cmp);
}

void quicksort(int* A, uint p, uint r, CompareFunction cmp) {
    if (p < r) {
        uint q = partition(A, p, r, cmp);
        quicksort(A, p, q, cmp);
        quicksort(A, q + 1, r, cmp);
    }
}

uint partition(int* A, uint p, uint r, CompareFunction cmp) {

    int x = A[p];
    int i = p - 1;
    int j = r + 1;

    do {
        do { j--; } while ( cmp(x, A[j]) == ComparisonType::LessThan );

```

```

        do { i++; } while ( cmp(x, A[i]) == ComparisonType::GreaterThen );

        if (i < j) { swap(A[i], A[j]); } // "swap" is standard-library function

    } while (i < j);

    return j;
}

```

10 Lezione del 25-03

10.1 Classi e Oggetti

10.1.1 Classi

```

class ClassName{
private:
    // valore di default definito non esplicitamente
    // attributi e funzioni membro privati
    // accessibili solo all'interno della Classe
protected:
    // Attributi e metodi visibili solo ad altre classi nella gerarchia

public:
    // Attributi e metodi visibili a tutti
}

```

10.1.2 Costruttori

```

ClassName(); // Costruttore di default
ClassName(parametri); // Costruttore specifico
ClassName( const ClassName&); // Copy constructor
ClassName(ClassName&&) noexcept; // Move constructor

```

10.1.3 Distruttore

```

~ClassName(); // Gestisce la memoria dinamica

```

10.1.4 Operatori (funzioni membro)

Es

```

bool operator == (const ClassName&) const noexcept; // Comparazione
ClassName &operator = (const ClassName&); // Copy assignment
ClassName &operator = (ClassName&& ) noexcept; // Move Assignment

```

10.2 Sintassi della classe

```

class ClassName : [virtual] [private/protected/public] BaseClassName;{
    ...
}

```

Il valore di default è il *private* *ClassName* → classe derivata da *BaseClassName*

In C++ (a differenza di Java dove viene simulato) è presente l'ereditarietà multipla.

Si può ereditare in 3 modalità diverse:

- *private* → tutto ciò che è accessibile dalla classe madre è utilizzabile dalla classe;
- *protected*;
- *public*;

10.3 Sintassi dei Metodi

```
[virtual] type NameFunc (parameterList) [const] [noexcept] [override] = [assignment]
```

Pseudo assignment → = *0* ; = *default* ; = *delete*. Dove:

- *0* → il *pure virtual*;
- *default* → il default implementation (costruttore e distruttore) di *default/copy/move*;
- *delete* → delete di un metodo esistente.

10.3.1 Definizione concreta di un metodo

```

type ClassName:: FuncName(parametres) [specifiers]{
    ...
    code
    ...
}

```

10.4 Ereditarietà di tipo virtuale

Un problema ricorrente con l'ereditarietà multipla è la definizione di classi derivate a partire da classi base che condividono un antenato comune (il cosiddetto **diamond problem**)

La ridondanza può essere eliminata ricorrendo all'eredità virtuale.

L'effetto della parola chiave `virtual` in una clausola di derivazione è quello di forzare il compilatore a includere la base virtuale una sola volta nella definizione degli oggetti derivati, anche se essa appare più volte nella catena di derivazione. In questo modo si ottimizza l'uso delle risorse, e si risolvono a monte eventuali conflitti di nomi.

10.5 Esempio

```
#include <iostream>
using namespace std;
class A{
protected:
    uint size = 0;
    char* str = nullptr;
public:
A() = default;
A(uint num){
    cout << "Nuovo oggetto creato" << endl;
    size = num;
    str = new char[num]();
}
~A(){
    delete[] str;
    cout << "Oggetto eliminato" << endl;
}
};

int main(){

}
```

11 Lezione del 26-03

11.1 Template

Corrisponde al generics di Java.

Permette di definire classi parametriche su tipi di dato specificabili a livello di compilazione

Di solito si suddivide in header e sorgente (hpp e cpp). Nei template non è possibile compilare il file cpp in isolamento

11.1.1 Vector.hpp

```
template <typename Data>
class Vector{
    private:
    uint size = 0;
    Data* Element = nullptr;
    public:
Vector()= default;
Vector(uint n); // rappresentata di seguito
~Vector(); // rappresentata di seguito
...
Data& operator[](ulong)
}
#include "Vector.cpp"
```

No *naked new* and *delete* in Vector

11.1.2 Vector.cpp

```
template <typename Data>
Vector<Data>::Vector(uint n){
    Element = new Data[n];
    size = n;
}
template <typename Data>
Vector<Data>::~Vector(){
    delete[] Elements;
}
```

11.1.3 main

```
#include "Vector.hpp"
int main(){
Vector<int> vec(5);
vec[0] = 3;
cout << vec[0];
return 0;
```

}

11.2 Differenze e similitudini con i linguaggi C e Java

C++ è (quasi) un sovrainsieme del linguaggio C.

C++ è C con le classi (e overloading, riferimenti, template, eccezioni, etc.).

Poche differenze a livello semantico (costrutti validi in entrambi i linguaggi).

Java è un linguaggio ad oggetti inspirato al C++. Esistono tuttavia svariate differenze a livello progettuale, tra cui:

C++	Java
No garbage collector	Garbage collector
Native executable	Virtual Machine
Allocation on stack & heap	Allocation (most) on the heap
Operator overloading	No operator overloading
Type-complete Meta Programming via Template	Type Parametrization via Generics
Eredità multipla	Eredità singola

12 Lezione 08-04 09-04

Descrizione del secondo esercizio + domande e chiarimenti sul primo esercizio

13 Lezione del 15-04

13.1 Gli alberi binari

Un albero binario è un insieme dinamico che è vuoto oppure è composto da 3 insiemi disgiunti di nodi:

- Un insieme di cardinalità 1, detto **nodo radice**;
- Due sottoalberi
 - Sottoalbero sinistro;
 - Sottoalbero destro.
- Presenta un'accesso alla radice

- nodi connessi tramite un collegamento, riferimento (a livello astratto);
- foglie (nodi con 0 successori).

L’albero binario dovrà:

- Essere percorribile in pre-order
 - L’algoritmo esplora la radice dell’albero come primo nodo fino ad arrivare alle foglie, accedendo ai singoli nodi prima di proseguire nel cammino verso i livelli più bassi;
- Essere percorribile in post-order
 - La visita dell’albero parte dalle foglie per poi risalire alla radice, che è l’ultimo nodo ad essere esplorato, al contrario di quanto avviene nella visita pre-order dove la radice è il primo nodo ad essere visitato, per poi finire alle foglie dell’albero;
- Essere percorribile in Ampiezza
 - Si visita per livelli;

Nei primi 2 casi per la loro natura si sceglie una implementazione di tipo ricorsivo. Nel 3 caso sceglieremo l’utilizzo di una coda.

Alberi completi

Un albero binario completo è un albero binario in cui ogni livello, tranne eventualmente l’ultimo, è completamente pieno, e tutti i nodi sono il più a sinistra possibile. Particolarmente vantaggioso in quanto $\mathbf{h} = \lceil \log n \rceil$.

Si sceglie in questo caso una rappresentazione vettoriale. In tutti gli altri casi si sceglie l’utilizzo di liste linkate.

Con una rappresentazione vettoriale la scelta più ovvia per l’implementazione è quella in ampiezza.

Visita in Order

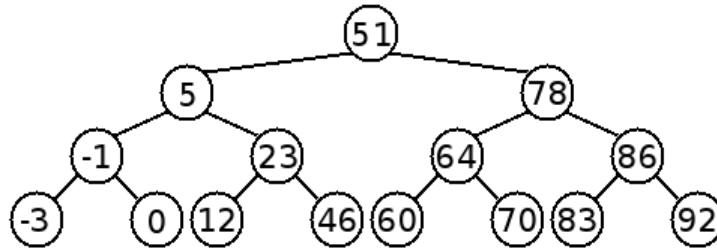
L’algoritmo di visita in-order è un particolare algoritmo usato per l’esplorazione in profondità dei nodi di un albero binario. In questo tipo di visita, per ogni nodo, si esplora prima il sottoalbero sinistro poi si visita il nodo corrente ed infine si passa al sottalbero destro. Più precisamente, l’algoritmo esplora i rami di ogni sottoalbero fino ad arrivare alla foglia più a sinistra dell’intera struttura, solo a questo punto si accede al nodo. Terminata la visita del nodo corrente si procede poi con l’esplorazione del sottoalbero a destra, visitando sempre i nodi a cavallo dell’esplorazione del sottoalbero sinistro e quello destro.

13.2 Albero binario di ricerca

Permette di effettuare in maniera efficiente operazioni come: ricerca, inserimento e cancellazione di elementi.

Un albero binario di ricerca ha le seguenti proprietà:

- Il sottoalbero sinistro di un nodo x contiene soltanto i nodi con chiavi minori della chiave del nodo x ;
- Il sottoalbero destro di un nodo x contiene soltanto i nodi con chiavi maggiori della chiave del nodo x ;
- Il sottoalbero destro e il sottoalbero sinistro devono essere entrambi due alberi binari di ricerca.



14 Lezione del 16-04

14.1 Gli iteratori

Un iteratore è un oggetto che viene costruito a partire da una particolare struttura dati, con il quale sarà poi possibile scorrere la struttura in un determinato ordine.

Presenta quindi:

- Un costruttore;
- Un distruttore;
- Un Access Operator(*, ->);
- Test di terminazione;
- Operatore successore (++);
- Operatore predecessore (--);

14.1.1 Implementazione degli iteratori sugli alberi

Caso di visita in Ampiezza → per l'accesso con l'iteratore:

```
itr Ampiezza :: A (*itr)
```

Con l'incremento dell'iteratore avrà il valore del nodo corrente → A → ++itr → B → ++itr → C e così via.

14.2 Tipi di iteratori concreti

- **BTPreOrderIterator** → naviga l'albero in Pre-Order;
- **BTPostOrderIterator** → naviga l'albero in Post-order;
- **BTInOrderIterator** → naviga l'albero in InOrder;
- **BTBreadthIterator** → naviga l'albero in Ampiezza;

Un oggetto di tipo iteratore su albero deve contenere i seguenti dati:

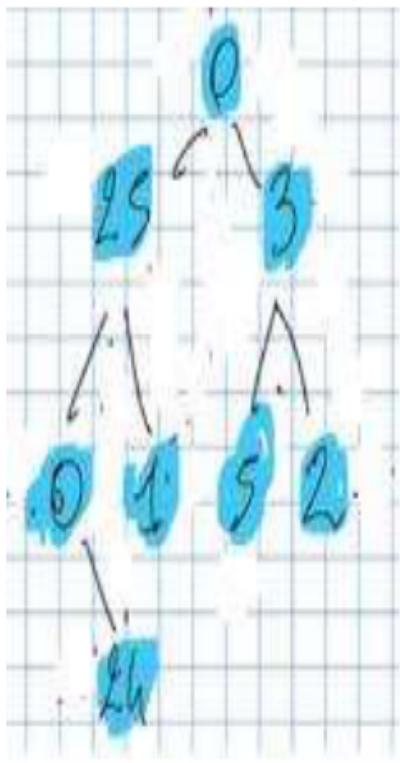
- Puntatore a nodo → punterà al nodo corrente dell'iteratore;
- Struttura dati → necessaria per salvare i nodi scoperti (allo scopo di arrivare al nodo giusto). Si usa in pratica una coda per l'iteratore in ampiezza e uno stack per gli altri tipi di iteratore.

15 Lezione del 22-04

15.1 Visite negli alberi binari

Gli alberi possono essere attraversati in diversi modi:

- **Pre-Order** → prima apro, scopro e visito il nodo corrente, poi il sottoalbero del figlio sinistro e successivamente quello destro;
- **In-Order** → scopro il nodo, visito il sottoalbero del figlio sinistro, poi il nodo e infine visito il sottoalbero destro. Questo tipo di visita è possibile solo per gli alberi binari poichè si perderebbe la sua proprietà per alberi con arit  maggiore di 2 (ad es. ho 3 figli, visito il mio nodo prima o dopo il sottoalbero maggiore centrale?)
- **Post-Order** → prima scopro il nodo, visito il sottoalbero sinistro e poi il destro e infine il nodo stesso. Questa modalità è quella spesso usata per distruggere l'albero (se distruggo il nodo perdo infatti le informazioni dei figli);
- **Aampiezza** → si visita l'albero per livelli, prima tutti i nodi a livello 0 fino a quelli a livello **h**



Ad esempio, dato l'albero a sinistra,
avremo i seguenti tipi di visita:

Pre	0 25 0 24 1 3 5 2
Post	24 0 1 2 5 5 2 3 0
Anep.	0 25 3 0 1 3 2 24
In	0 24 25 1 0 5 3 2

16 Lezione del 23-04

(Descrizione della libreria 3 + domande)

17 Lezione del 29-04

Alberi bilanciati

Utilizzeremo gli ABR con rappresentazione in memoria attraverso il BinaryTreeLink (possibile fare anche con il BinaryTreeVec, ma comporta costi maggiori).

Dato un albero di ricerca i metodi principali sono:

- Ricerca;
- Confronto;
- Rimozione e inserimento di nodi;
- Rimozione e lettura di
 - Exists;

- Insert/Remove;
- Max/Min (Get/Remove);
- Predecessor/Successor (Get/Remove);

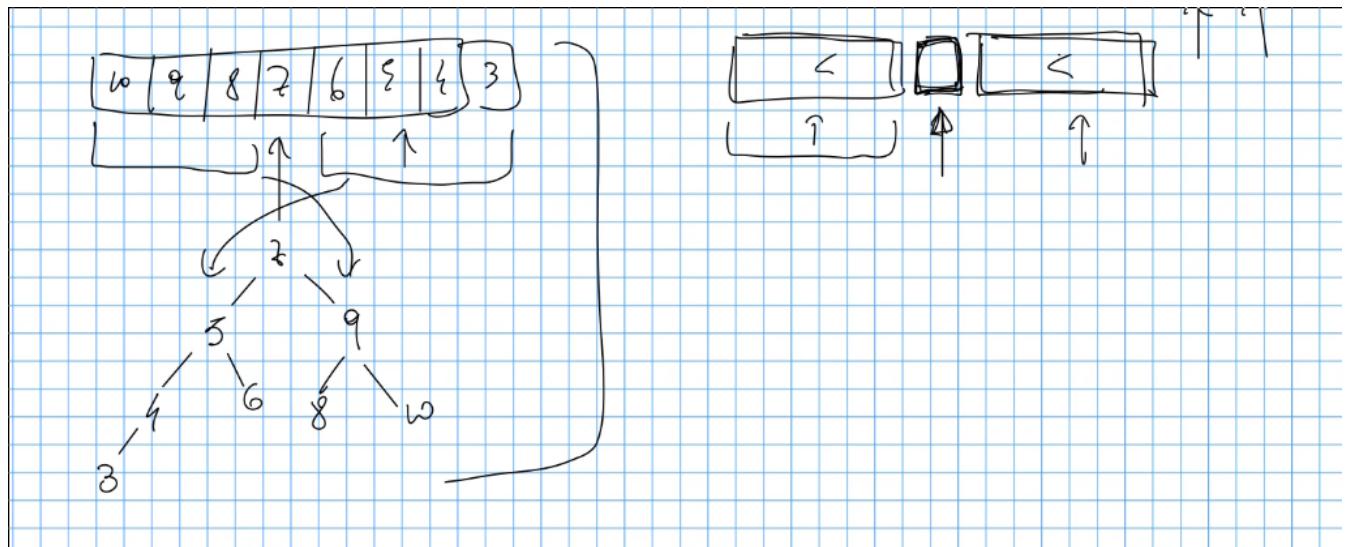
17.1 Costruzione di un ABR dato un Linear Container

Memo

$\forall n \in T$

$$\forall n \in T \rightarrow \text{sx } x.\text{key} < .\text{key}$$

$$\forall n \in T \rightarrow \text{dx } x.\text{key} > .\text{key}$$



17.2 Costruttori di copia e spostamento

Con questo tipo di costruttori (e anche nel caso di assegnamento di entrambi i tipi) è possibile ereditare tutti i costruttori di un ABR con BinaryTreeLink.

17.3 Operatori di confronto

Questi necessitano una reimplementazione, perchè arriviamo ad una specializzazione di alberi.

Per gestire i costi si utilizza la visita in order (che è l'unica che permette di confrontare gli alberi che strutturalmente possono essere diversi) sfruttando gli iteratori.

17.4 Exists

Basta sfruttare la ricerca binaria → Di default ricerca nell'intero albero (deve essere reimplementato). Viene in aiuto il tipo di albero in esame (è un ABR) che ha una complessità minore (logaritmica).

17.5 Insert/Remove

Per l'Insert si utilizza la ricerca binaria per definire la posizione in cui inserire l'elemento.

Per la Remove bisogna prima controllare che il nodo da rimuovere abbia 1 o 2 figli da rimuovere (in input avrò un dato → devo ovviamente ricercarlo).

17.6 Max/Min (Get/Remove)

17.7 Predecessor/Successor (Get/Remove)

18 Lezione del 30-04

18.1 Inserimento e Remove negli alberi

Richiedono inizialmente una fase di ricerca (`Insert(d)` e `Remove(d)`).

Serve quindi una **Accessory Function** che gestisca il caso di assenza/presenza del dato nella struttura → `FindPointerTo(nodo, d)` (la funzionalità della ricerca dovrà restituire un riferimento al puntatore al nodo).

18.2 Funzione Detach

La rimozione del nodo all'interno del nodo avverrà in maniera diversa in base ai casi (3):

- Nel caso di rimozione dell'albero sinistro (unico esistente), si dovrà creare una variabile locale (`tmp`) che punta al nodo in modo da non perderne i riferimenti nel momento in cui venga fatta la detach. La variabile `tmp` poi ci permetterà di fare l'operazione di congiungimento del nodo al figlio (`SkipOnLeft`);
- Nel caso di rimozione dell'albero destro (unico esistente), si dovrà creare una variabile locale (`tmp`) che punta al nodo in modo da non perderne i riferimenti nel momento in cui venga fatta la detach. La variabile `tmp` poi ci permetterà di fare l'operazione di congiungimento del nodo al figlio (`SkipOnRight`);
- Nel caso in cui il nodo da eliminare abbia entrambi i sottoalberi si potrà fare una `DetachMin` (figlio destro che si sostituisce al nodo da eliminare) o una `DetachMax` (figlio sinistro che si sostituisce al nodo da eliminare).

18.3 Predecessor & Successor

```
const Data& Predecessor(Data) // Il const per essere sicuri di non modificare il valore del dato
const Data& Successor(Data) // Il const per essere sicuri di non modificare il valore del dato
```

Il **Predecessor** di un dato è l'elemento che tra le chiavi più piccole ha la chiave più grande → è il massimo del sottoalbero sinistro. Se il sottoalbero sinistro è vuoto, si cerca il primo antenato che abbia come figlio destro il nodo stesso o un suo antenato.

Il **Successor** di un dato è l'elemento che tra le chiavi più grandi ha la chiave più piccola → è il minimo del sottoalbero destro. Se il sottoalbero destro è vuoto, si cerca il primo antenato che abbia come figlio sinistro il nodo stesso o un suo antenato

19 Lezione del 06-05

19.1 Matrice

Matrice → tabella ordinata di elementi. Due tipi:

- Sparsa → una matrice con *molti* elementi pari a 0;
- Densa → una matrice con *pochi* elementi pari a 0;

Diversi tipi di rappresentazione concrete. In particolare:

- Rappresentazione *multivettoriale*;
- Rappresentazione tramite *liste ortogonali*;
- Compress **row/column** rappresentazione (Yale Format/Rapresentation).

Supponiamo di avere una matrice 7x6 che abbia pochi dati "pieni" (gli spazi vuoti rappresentano un dato di *default* o la sua assenza):

	0	1	2	3	4	5
0	A	B	C	D	E	F
1	G	H	I	J	K	L
2	M	N	O	P	Q	R
3	S	T	U	V	W	X
4	Y	Z				
5						

19.1.1 Rappresentazione Vettoriale

Nel caso di rappresentazione vettoriale avremo una `size = numCol x numRighe` (array-multidimensionale) in cui verranno serializzati i dati della matrice all'interno di quest'ultimo.

Footprint in memoria → sequenza di tot dimensione di elementi. Di conseguenza ci sarà una delle 2 serializzazioni:

- Una serializzazione per riga (Row major order);
- Una serializzazione per colonna(Column major order).

Per motivi di operazioni matematiche si preferisce la seconda.

L'accesso all'elemento (i, j) sarà:

- Nel caso di Row major order sarà $\rightarrow i \times \text{colonna} + j$;
- Nel caso di Column major order sarà $\rightarrow j \times \text{riga} + i$;

Questa rappresentazione presenta un grande spreco di spazio con matrici sparse

19.1.2 Rappresentazione a lista (lineare)

Per ovviare a questo problema si può avere una rappresentazione per triple → n° riga, n° colonna, valore del dato (o invertito con colonna, riga, valore).

In questo modo verranno occupato solo lo spazio occupato realmente (spreco di memoria minimo nel caso di matrici sparse → viene occupato solo lo spazio che contiene veramente un dato).

I vantaggi/svantaggi di questa rappresentazione rispetto alla precedente si riducono ai vantaggi/svantaggi delle strutture che permettono questo tipo di rappresentazione.

19.1.3 Rappresentazione Compress Row/Column (CSR/CSC)

Sfrutta un'idea diversa per la rappresentazione.

Sfrutta 3 vettori:

- Un vettore dei dati(A);
- Un vettore delle colonne (C);
- Un vettore delle righe (R).

The compressed sparse row (CSR) or compressed row storage (CRS) or Yale format rappresenta una matrice M per tre matrici (unidimensionali), che contengono rispettivamente valori diversi da zero, le estensioni delle righe e gli indici delle colonne.

Nel caso di una matrice:

5	0	0	0
0	8	0	0
0	0	3	0
0	6	0	0

Avremo:

- A → [5 8 3 6];
- C → [0 1 2 1];
- R → [0 1 2 3 4].

La Compressed sparse column (CSC) è simile al CSR tranne per il fatto che i valori vengono letti prima per colonne, un indice per le righe è mantenuto in memoria per ogni valore e vengono archiviati i puntatori di colonna.

Per rendere più efficiente la ricerca all'interno di questo tipo di rappresentazione si sfrutta il fatto che il vettore

di riga/colonna sia ordinato → si fa una ricerca binaria su questo dato (bisezione dell'array). Otteniamo una performance di ricerca di tipo logaritmica.

Un eventuale estensione di riga di questo tipo di matrice potrebbe essere addirittura a tempo medio costante (anche se nel caso di matrici un ridimensionamento oltre a essere una operazione rara, è molto costosa).

Non vale lo stesso ragionamento nel caso dell'altro indice (la colonna, nel caso di una rappresentazione di una matrice tramite compress sparse row).

Stesso ragionamento vale per una matrice con rappresentazione Compress sparse column.

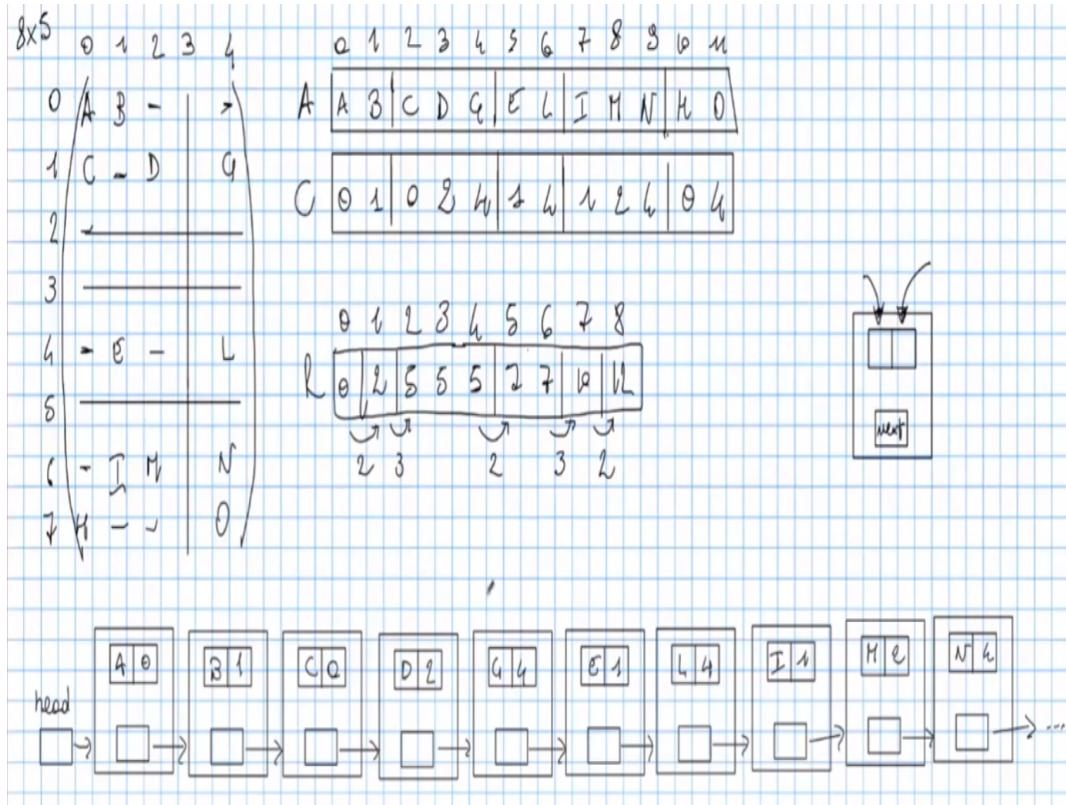
Per ovviare a questo problema si può pensare di sostituire i 2 vettori (A, C) con delle liste (con lo scopo di diminuire il costo del ridimensionamento).

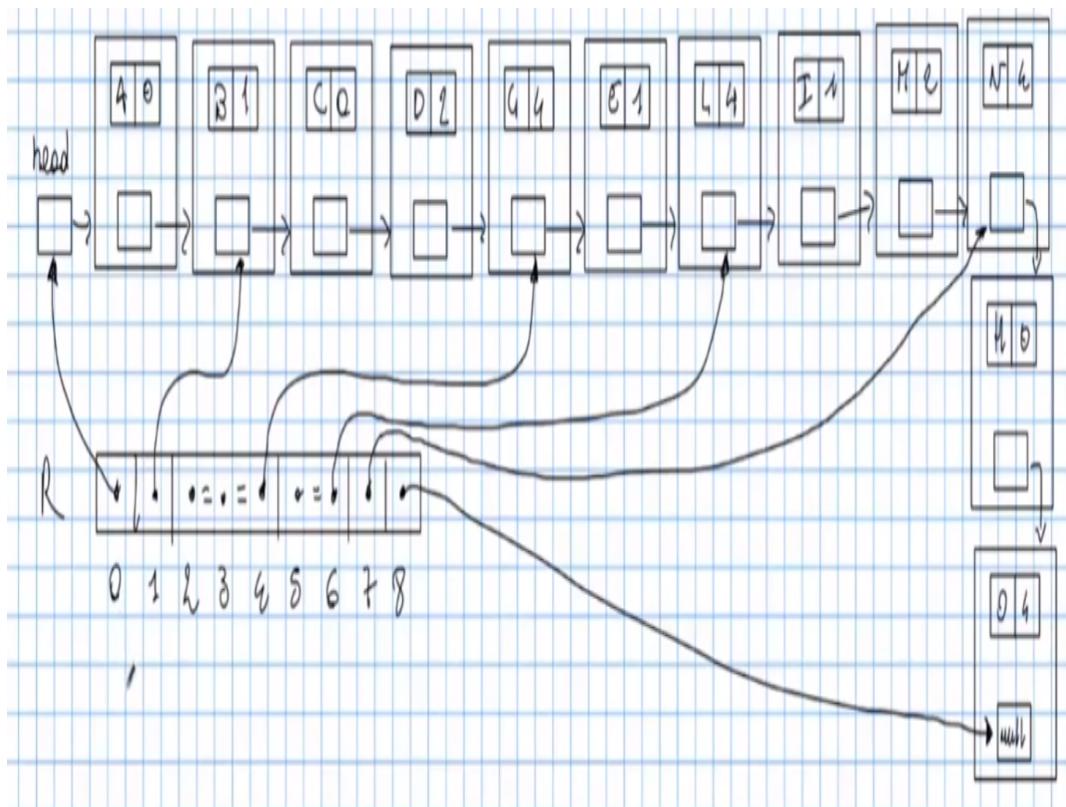
20 Lezione del 07-05

(Descrizione della libreria 4 + domande inerenti alla libreria 3/4)

21 Lezione del 13-05

21.1 Recap su CSR





Questo tipo di rappresentazione è estremamente succinta ed ha un accesso ai dati relativamente rapidi (tempo logaritmico nel numero degli elementi per riga).

Presenta il problema del ridimensionamento (nell'ampliare la struttura si, nel ridimensionamento a restringere no → si potrebbe inoltre eliminare i dati senza eliminare memoria per sfruttarla poi in seguito).

Il problema quindi è quello dell'aggiunta di un elemento non precedentemente esistente → obbliga a riallocare totalmente il vettore A e C (richiede un tempo lineare).

Per ovviare a questo problema un possibile metodo è quello di allocare un maggiore spazio (richiede una gestione non banale e si va incontro anche a un certo spreco di spazio, che è quello che si cerca di evitare con questo tipo di rappresentazione).

Un altro modo è quello di sostituire i vettori A e C con delle liste (come si vede in figura). Nel vettore R ogni elemento è un puntatore al puntatore al nodo (non è diretto al nodo, perché non permetterebbe una modifica pulita al nodo). Questa modifica permetterebbe di rendere l'inserimento di un elemento non precedentemente esistente molto meno costoso (tempo costante), ma di conseguenza la ricerca diventa per forza di cose lineare.

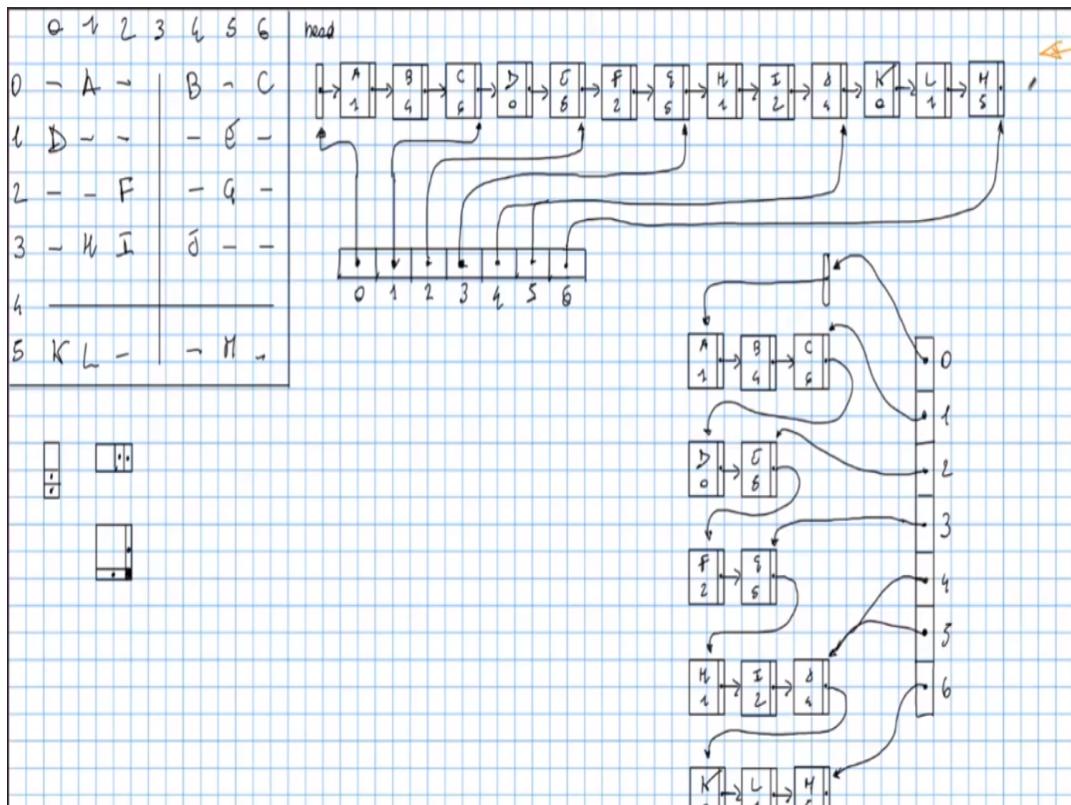
Per estendere il numero di righe → si fa una resize e vanno duplicati i valori terminali (per non distruggere poi successivamente la corrispondenza con le colonne).

Per diminuire il numero di righe → si fa una resize in base all'indice di riga che si vuole mantenere, di seguito si fa una delete di tutto quello che non si vuole tenere (dereferenziando ovviamente).

Lo svantaggio rispetto a delle liste ortogonali è che l'implementatore forza ad avere una rappresentazione **row/column major based**

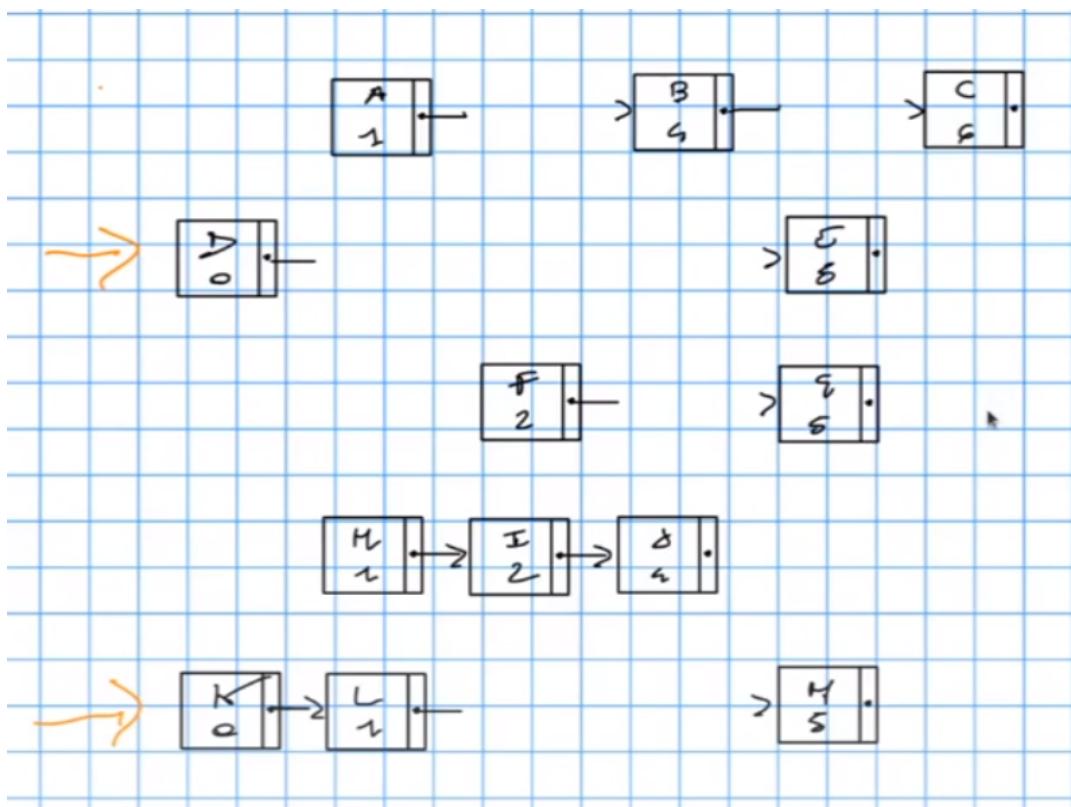
22 Lezione del 20-05

22.1 Liste ortogonali

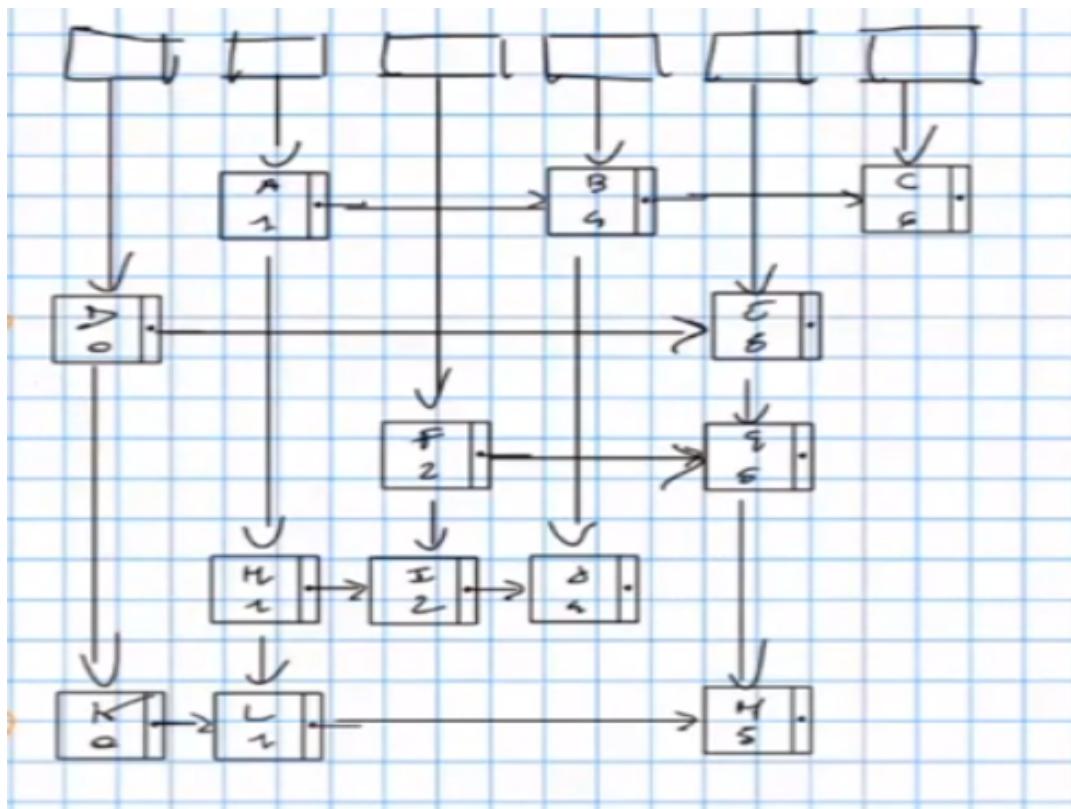


Se andassimo a sezionare la sequenza in base alle righe otteremo la rappresentazione in basso → una lista con una vista separata per righe.

Bisogna poi aggiustare il tiro per l'accesso in verticale (in colonna). Si riordano i dati in modo tale che corrispondano alle colonne corrispondenti:



A questo punto se avessimo una struttura in cui ci siano dei puntatori alle colonne, potremmo intuitivamente puntare agli inizi delle colonne e collegare queste con gli altri nodi nella stessa colonna (ogni nodo ha in sè l'informazione del prossimo nodo in riga e del prossimo nodo in colonna).



Se andassimo a rappresentare R in una lista ogni cella conterrà 3 informazioni:

- L'indice numerico;
- Il puntatore che punta alla prima cella della riga;
- L'informazione relativa alla prossima cella nella stessa lista.

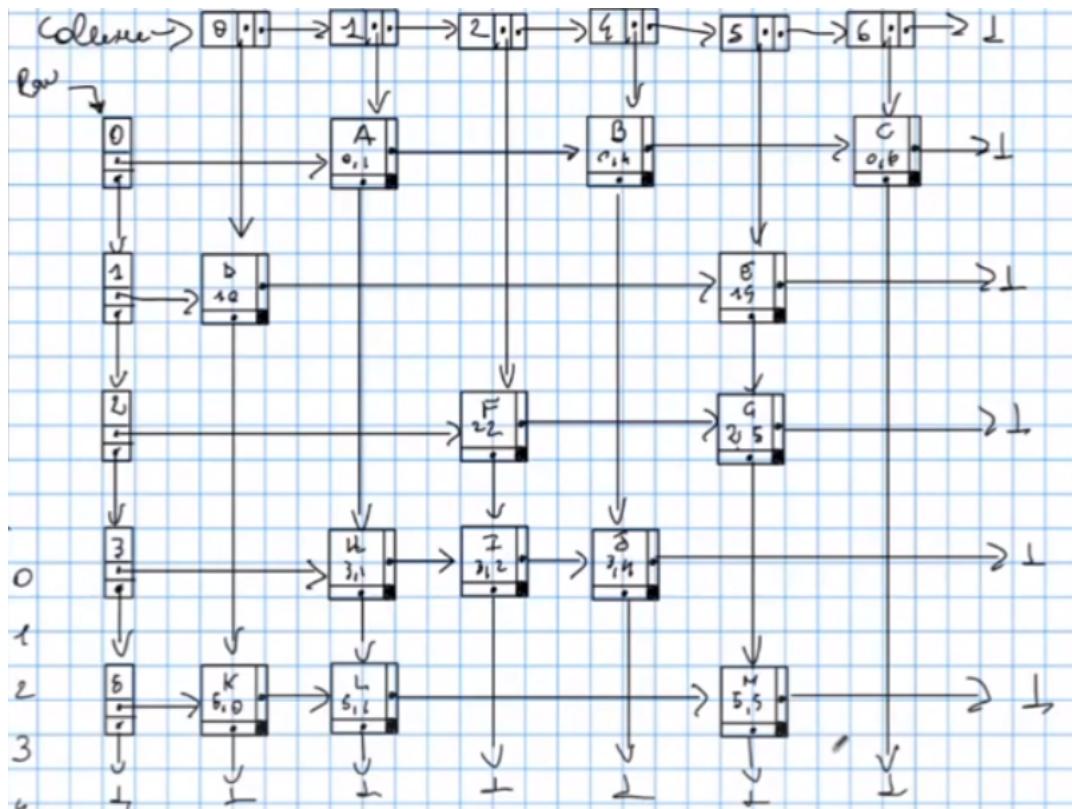
Avremo in modo simile una lista per *column* con:

- L'indice di colonna;
- L'informazione relativa alla prossima cella nella stessa colonna.

I dati avranno:

- Puntatori agli adiacenti (sia in verticale che in orizzontale);
- Il contenuto del dato;
- Indice di riga e colonna.

Otteremo quindi una vista di questo tipo:



La rappresentazione ortogonale la si può vedere come una trasformazione delle CSR e CSC e una fusione delle stesse.

23 Lezione del 21-05

(Descrizione della libreria 5 + domande inerenti alla libreria)

24 Lezione del 27-05

24.1 Definizione

Un grafo può essere immaginato come un insieme di punti disposti casualmente nello spazio collegati da ponti agli altri punti.

Un *grafo orientato* \mathbf{D} è un insieme $D = (V, A)$, dove V è l'insieme dei vertici di D e A è l'insieme degli archi orientati di D .

Un *arco orientato* è un arco caratterizzato da una direzione. In particolare, è composto da una *testa* (rappresentata solitamente dalla punta di una freccia), che si dice raggiunge un vertice in entrata, e una *coda*, che lo lascia in uscita. Un *grafo non orientato* D è un insieme di vertici e archi dove la connessione $i - j$ ha lo stesso significato della connessione $j - i$.

Grafo pesato → Si associa l'informazione a un *peso* di ogni arco. Dunque un grafo con archi che hanno dei *pesi* è nominato grafo pesato.

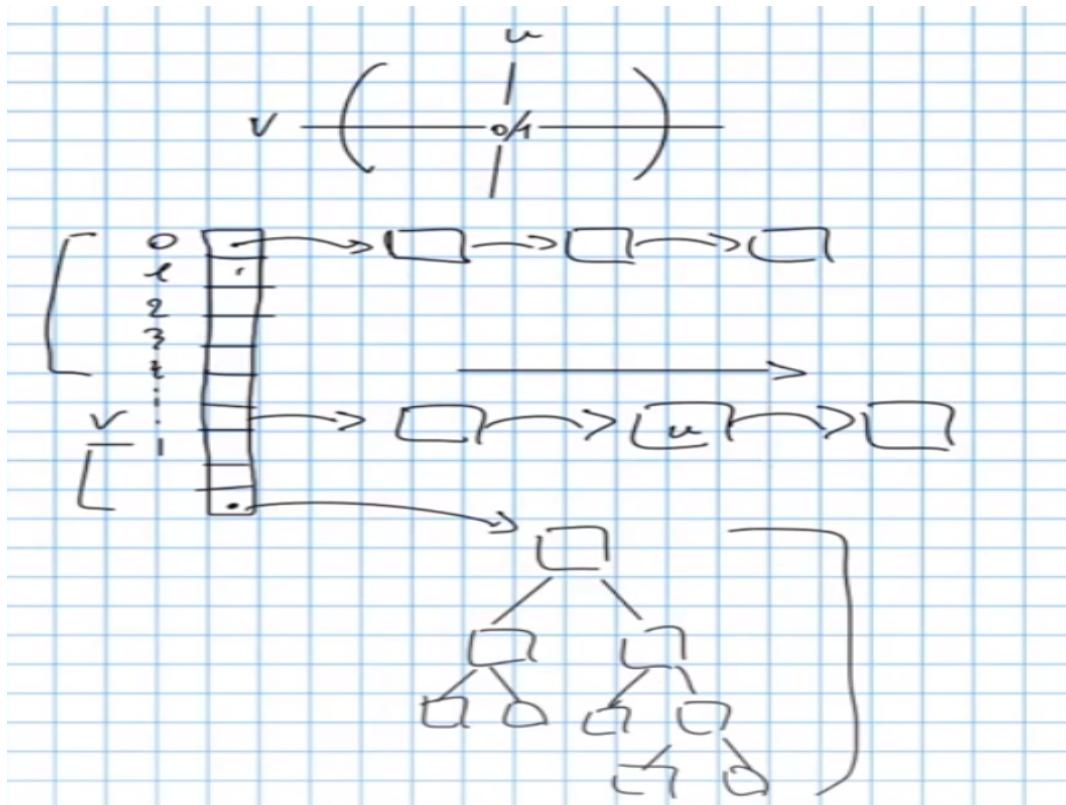
Grafo etichettato → Si dice grafo etichettato un grafo che presenta vertici muniti di una etichetta.

24.2 Rappresentazione

- Matrice di adiacenza;
- Lista di adiacenza (alberi di adiacenza);

Matrice di adiacenza → una matrice che a una certa coppia di coordinate corrisponde 0/1 se nel nodo di indice V c'è un arco che va verso il nodo di indice U (lo zero rappresenta l'assenza di questo arco).

Nella lista di adiacenza invece abbiamo un vettore V lungo quanto i nodi presenti nel grafo, rappresentati con gli indici di numeri naturali in cui ognuno di questi non è altro che un puntatore ad una lista (nella gestione del vettore si ha un accesso immediato alla lista di adiacenza, con un eventuale spreco di spazio → caso di grafo dinamico). Analogamente nel caso di un'albero di adiacenza dove il vettore andrà a puntare non ad una lista, ma alla radice di un ABR (che si spera sia bilanciato).



24.3 Rappresentazione dei grafi etichettati

Il modo più banale per farlo è avere una matrice di puntatori, dove:

- Il `nullptr` rappresenta l'assenza dell'arco (e di conseguenza del dato);
- Un puntatore `non null` che punta ad una struttura dati che contenga il dato (di conseguenza rappresenta la presenza dell'arco).

Nel caso di rappresentazione mediante `liste/alberi` di adiacenza:

- Nei vettori di vertici abbiamo i puntatori alle liste / alle radici degli alberi;
- Queste conterranno dei nodi personalizzati con strutture più complesse.

Per l'etichettatura dei nodi aggiungiamo un vettore di appoggio.

24.4 Algoritmi di visita

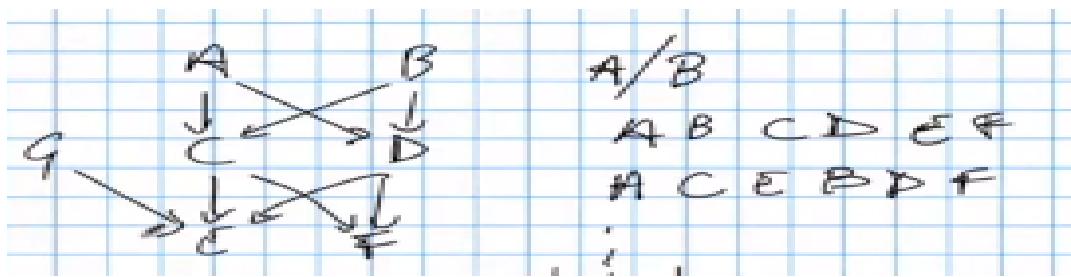
- Algoritmi di visita in ampiezza;
- Algoritmi di visita in profondità
 - Pre-order;
 - Post-order;
- Con iteratori
 - Ampiezza;
 - Profondità;
- Acyclicity Test;
- Iterator for Topological Order (Grafi aciclici);
- SCC (Componenti fortemente connesse);

Nel caso di grafo arbitrario è fondamentale la funzione di colorazione per la terminazione.

Nel caso di grafo aciclico non è importante per la terminazione, ma lo è per la complessità (si rischia di ottenere percorsi esponenziali → situazione che si cerca di evitare).

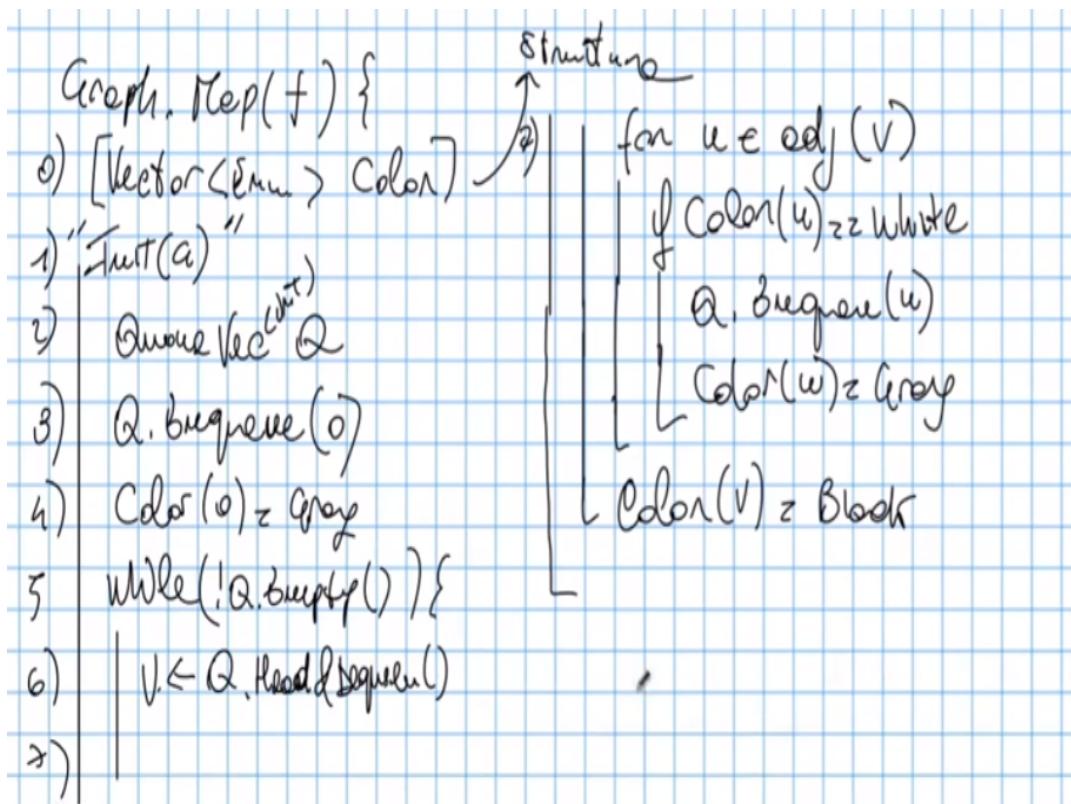
24.4.1 Visita in ampiezza (la classica map)

NB → quando esploriamo un grafo, questo viene esplorato come se fosse una foresta, in quanto andiamo a troncare tutti i cicli.



Richiede una funzione accessoria, che può essere:

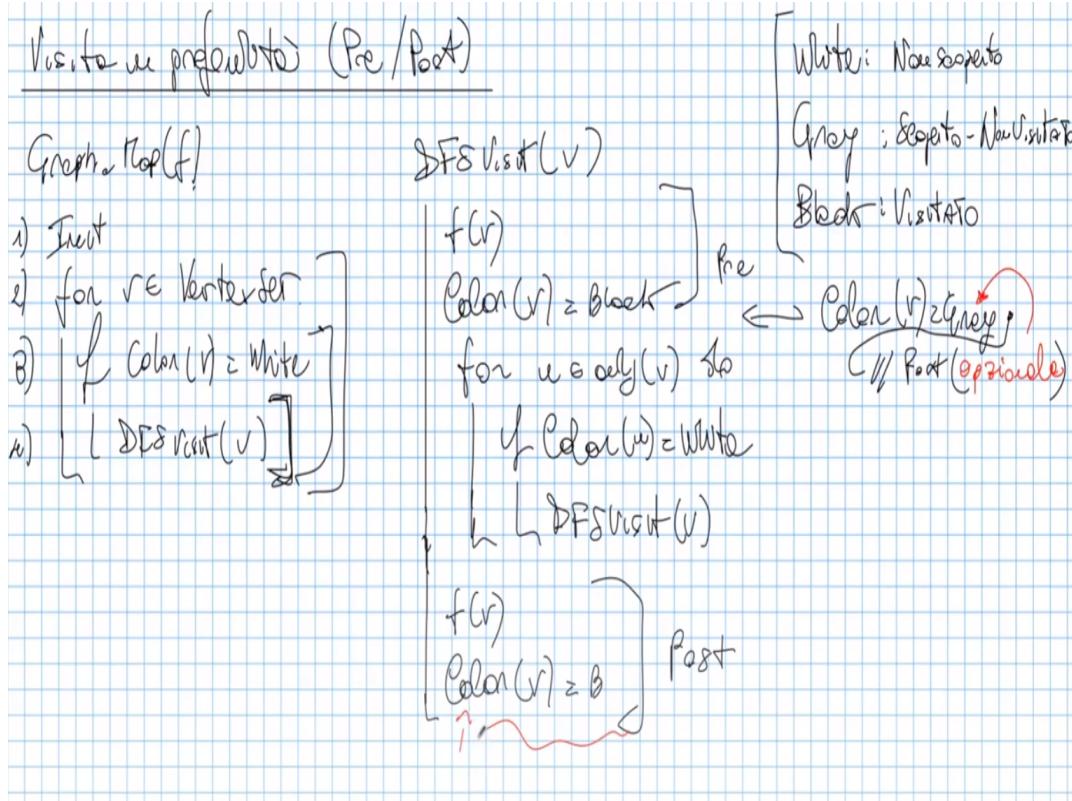
- `Vector<Enum ...> Color` (richiede un'allocazione di memoria ad ogni chiamata);
- Una opportuna variabile inserita all'interno della struttura dati, che andrà a variare durante la visita.



NB → In questo caso viene assunto che il grafo sia SCC (cosa non vera). Dobbiamo infatti scorrere tutto l'insieme dei vertici!

25 Lezione del 28-05

25.1 Visita in profondità (Pre/Post)



Memo:

- White \rightarrow non scoperto;
 - Gray \rightarrow scoperto, non visitato;
 - Black \rightarrow visitato.

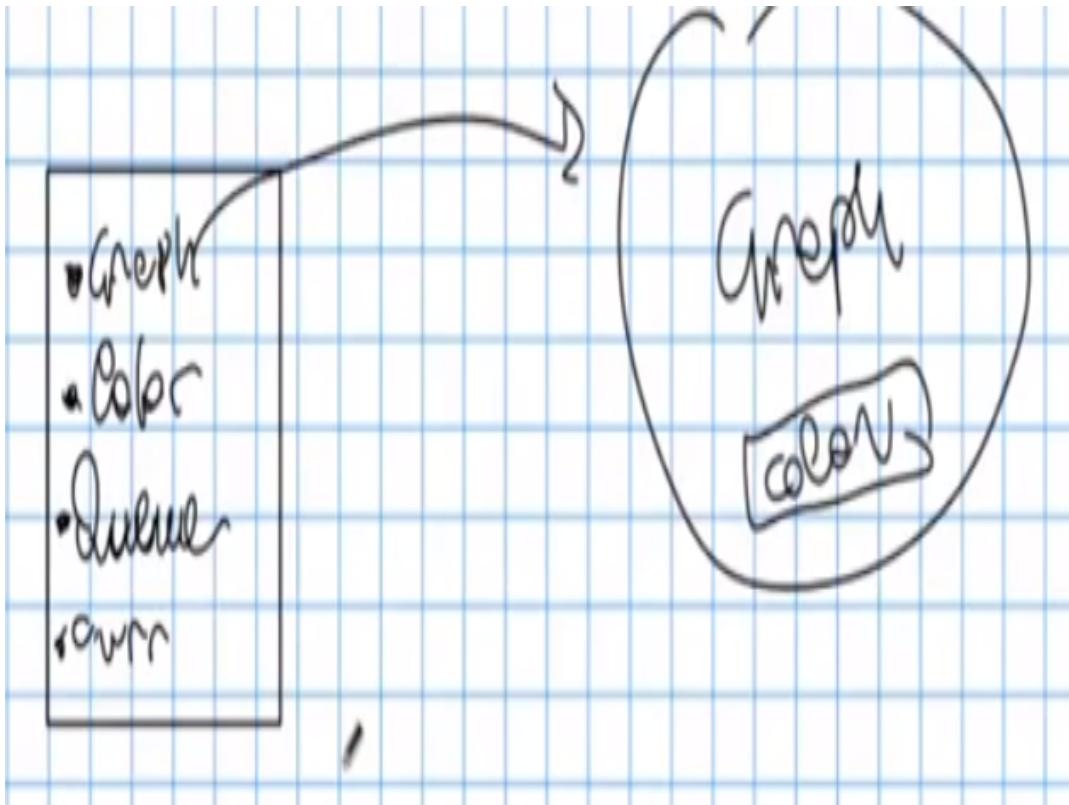
Nel caso si volesse tener traccia di un nodo scoperto ma non visitato (ciclicità) avrebbe senso di usare il grigio (quindi nel caso del post). In generale non è necessario (si può rischiare di andare in loop).

La differenza sostanziale tra la post e la pre è il punto in cui viene applicata la funzione (scelte semantiche che non hanno impatto sull'algoritmo).

25.2 Iteratore su Grafo un Ampiezza

- **Costruzione/Distruzione** (il distruttore ovviamente sarà quello di default);
 - Acces operator (**itr***);
 - Terminator check;
 - Successor generator (Forward Iterator).

Avremo una struttura che conterrà i seguenti dati:



Per il costruttore:

```
curr = 0;  
InitColor;  
Color(curr) = black;
```

Per il **Termination check** invece è necessario controllare che la coda sia vuota e che tutti i nodi siano neri. Dobbiamo in qualche modo avere un'informazione sulla posizione all'interno della struttura (il linear container dei nodi) e controllare che tutti i nodi siano neri.

25.3 Operatore **++()**

Si assume che Termination check sia falso e che current sia stato già visitato:

```

Operation  $\text{++}()$ 
| for  $v \in \delta_j(w)$ 
|   if  $\text{Color}(w) = \text{white}$ 
|     [Q,  $\text{Enqueue}(w)$ ]
|
|   if Q,  $\text{empty}$ 
|     for — (T)  $\xrightarrow{\sim} \text{curr} / s!$ 
|       else
|         curr = Q, Head &  $\text{Dequeue}$ 
|         [Color(curr) = black]
|
|       else
|         curr = Q, Head &  $\text{Dequeue}$ 
|         [Color(curr) = black]
|       end
|     end
|   end
| end

```

26 Lezione del 03-06

26.1 Costrutture e Successor (Iteratori)

Nel costruttore avremo 3 strutture (allocate staticamente, localmente):

- curr;
- stack;
- color.

In pseudo-codice questo si tradurrà con (pre-order):

```

InitColor;
curr = FindFirst;
Color[curr] = black; // essendo in pre-order lo scopro e lo visito

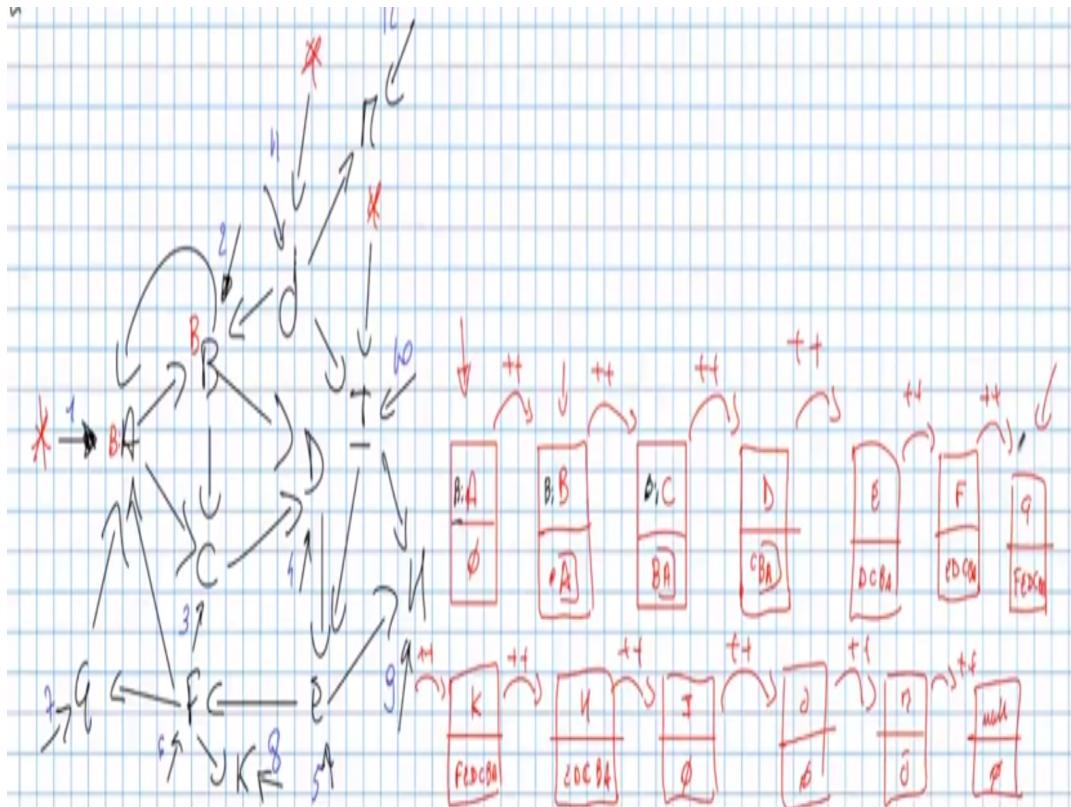
```

(Non consideriamo i casi limite, come il grafo vuoto).

Nel caso di Post-order:

1. Scorriamo gli adiacenti;
2. Al primo bianco faremo una chiamata ricorsiva (non scorriamo tutti gli adiacenti, come andrebbe fatto in ampiezza → prima inserisco tutto in coda e poi vado a prendere la testa).

Nel caso volessimo scorrere il grafo alfabeticamente:



È possibile notare che, arrivati al nodo G (che ha archi uscenti a nodi già visitati) si arriva ad una situazione di "stallo". Quindi si riparte da F (con una `Top()`, e inserisco K). Da qui ripeto l'operazione precedente e si continua con lo scorrimento del grafo.

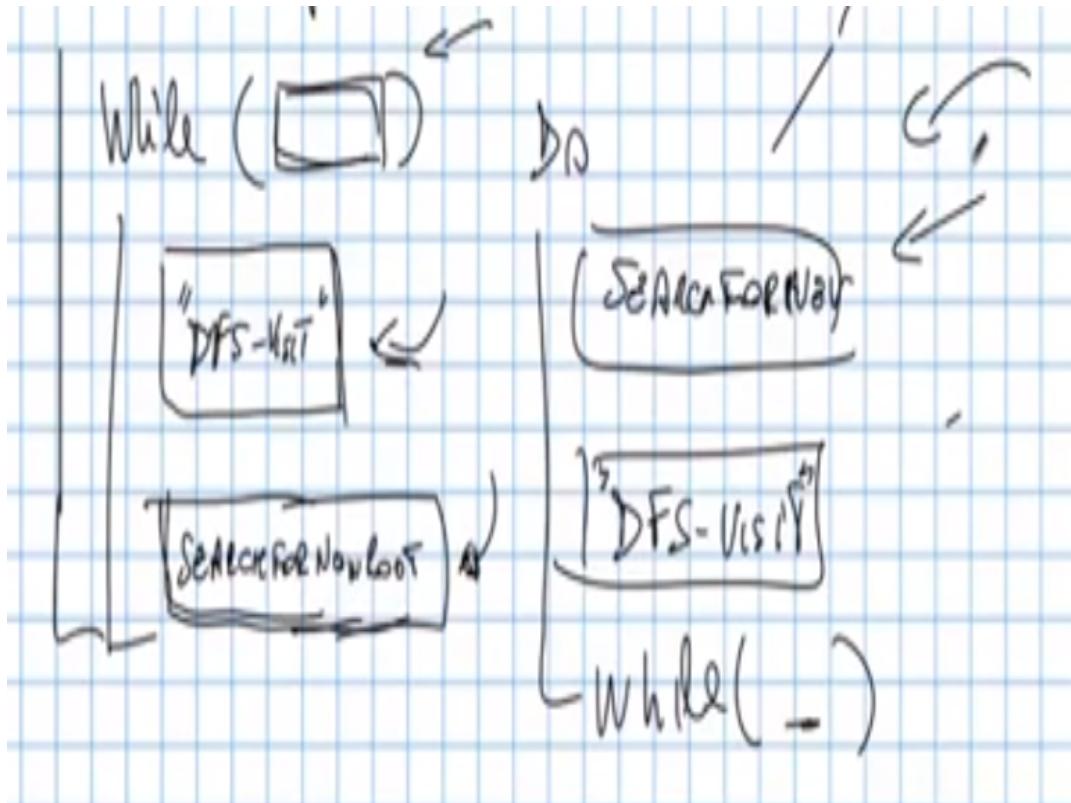
Nel caso si voglia avere un iteratore arbitrario (generico), che non vada a sfruttare l'informazione concreta, e che quindi possa essere realizzata su un concetto di grafo generico, la cosa più semplice da fare è tener traccia del nodo corrente e poi di accedere sempre agli adiacenti per andare a vedere il prossimo (o avere un'informazione su quale sia il prossimo adiacente per mezzo di un operatore, indice numerico o una qualche informazione che permetta di capire quale sia il prossimo nodo).

Nel Successor avremo:

1. Un `while` che controlla che ci sia ancora qualcosa da visitare (corrisponde al `for` della DFS del prof Benerecetti);

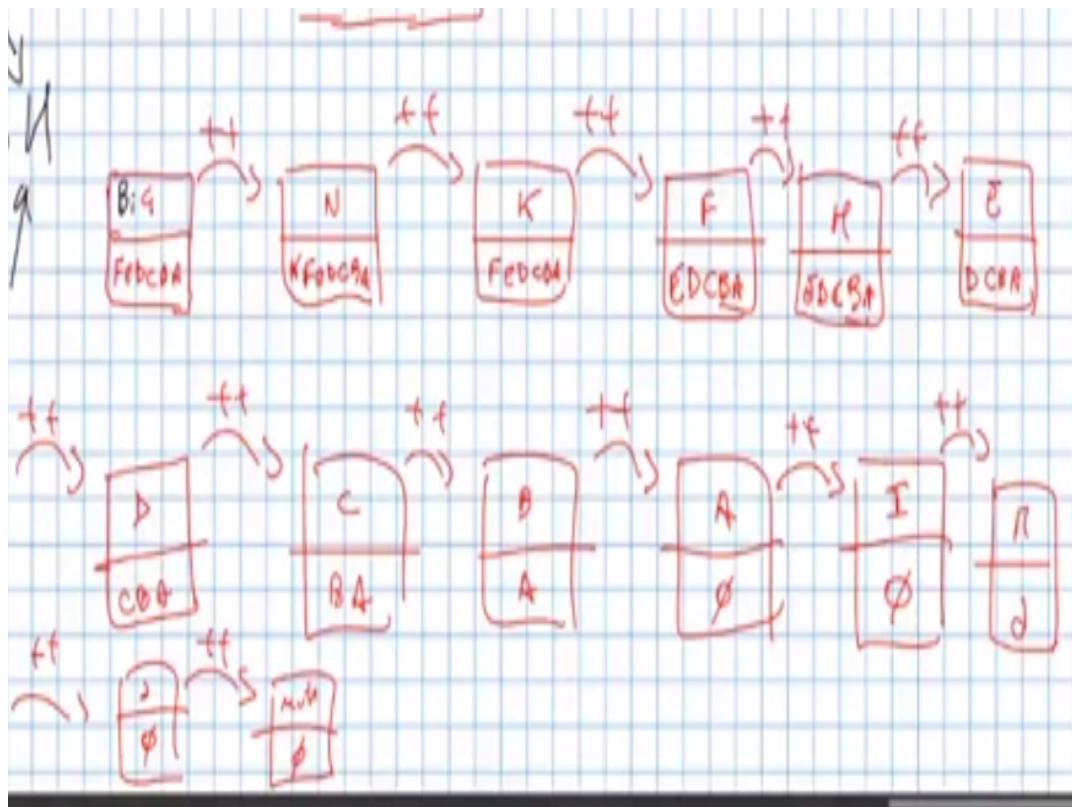
2. Ricerca della prossima radice;

In pseudo-codice:



26.2 Iteratore in Post-Order

Facendo sempre riferimento al grafo precedente:



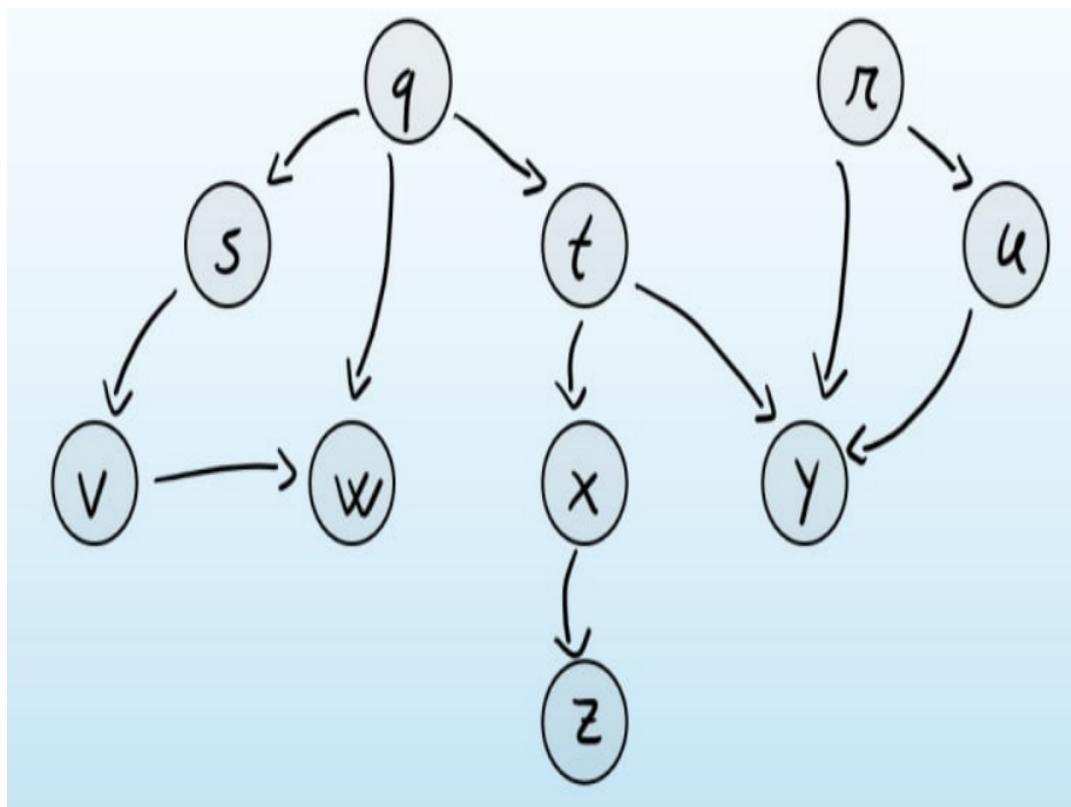
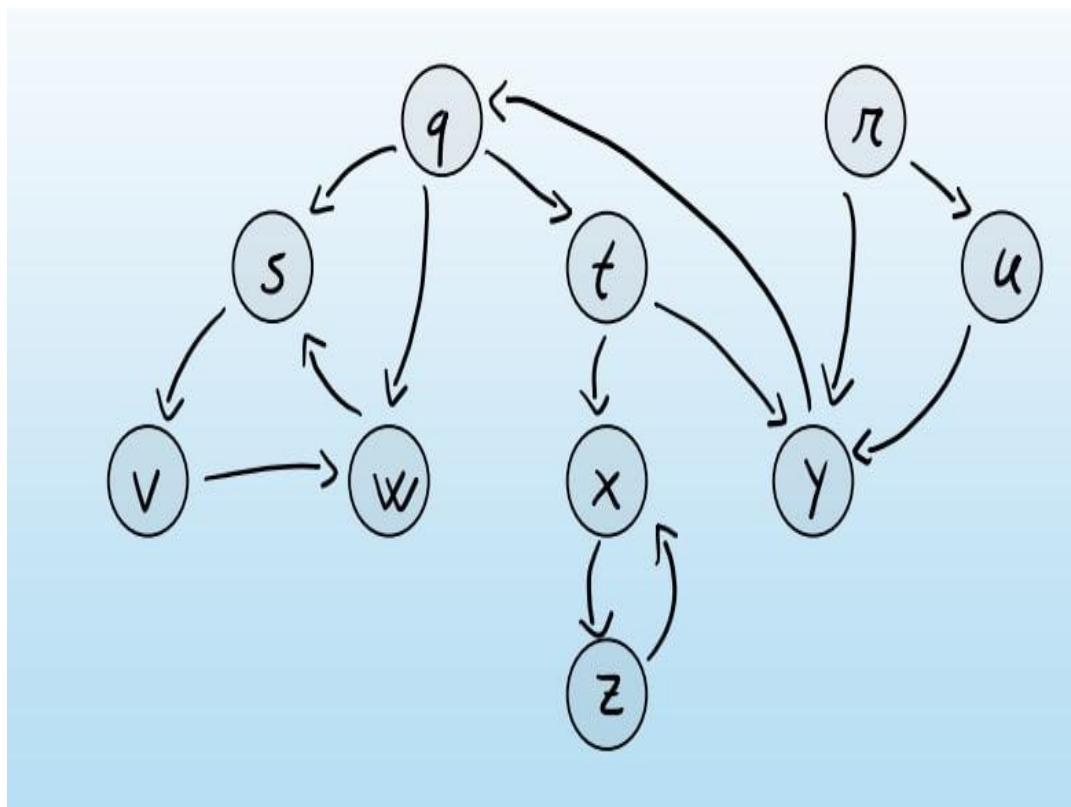
27 Lezione del 04-06

27.1 Acyclicity Test

27.1.1 Definizione

Un grafo che ha almeno un loop è detto ciclico, viceversa uno che non ne contiene è detto aciclico.

Di seguito un esempio di grafo ciclico e aciclico:



27.1.2 Idea dell'algoritmo

Condizione necessaria, ma non sufficiente, affinchè un grafo sia aciclico è quello di avere una foglia.

Per testare che un grafo sia aciclico:

1. Se il grafo non ha nodi, stop → il grafo è aciclico;
2. Se il grafo non ha foglie, stop → il grafo è ciclico;
3. Scegliere una foglia del grafo. Rimuovere la foglia e tutti gli archi interni nella foglia per ottenere un nuovo grafo;
4. Ripetere il punto 1.

27.1.3 Rappresentazione concreta e ragionamenti

Sarà ovviamente una funzione **booleana** che restituisce true (nel caso in cui sia aciclico) o false.

Per implementare un test di ciclicità è fondamentale la tripla di colori (W,G,B).

Nel momento in cui viene trovato un nodo grigio restituiamo **false** (una volta terminata la visita dell'intero grafo), altrimenti **true**.

NB → Non conviene andare in ampiezza (molto meno efficiente), ma è preferibile usare una visita in profondità.

27.2 Topological Ordering

Abbiamo quindi un grafo e una funzione **TopologicalOrdering()** senza parametri:

`G.TopologicalOrdering()`

che restituisce un opportuno iteratore che debba avere:

- Accesso;
- Operatore **++**;
- **IsTerminated**.

Nel momento in cui poi l'iteratore andrà a dereferenziare una certa posizione (ad es. la prima) indicherà un nodo del grafo che non ha archi entranti, e con l'operatore **++** dovrà creare una sequenza di elementi che una volta dereferenziati, creerà la permutazione dei nodi.

MEMO:

Ordine topologico di un grafo → Ordinamento dei suoi vertici che soddisfa la seguente condizione:

Per ogni arco (u,v) del grafo u precede v nell'ordinamento

- A ciascun vertice u si assegna un intero $p(u)$, in modo tale che, se esiste l'arco (u,v) , allora $p(u) < p(v)$;
- Di conseguenza se esiste un cammino da u a w allora
 - $p(u) < p(w) \rightarrow$ ogni nodo è seguito, nell'ordinamento da tutti i suoi discendenti.

Quindi se esiste un arco tra 2 vertici il vertice che ha l'arco uscente si troverà prima di quello con l'arco entrante.

NB → È sempre possibile determinare un ordinamento topologico di un DAG (digrafo aciclico).

27.2.1 Algoritmo con Grado entrante

Questo algoritmo non usa il colore → sfrutta il grado entrante (implicitamente quindi è come se stessimo usando il colore). In pseudo-codice

```
ComputeInDegree; // Calcolo dei gradi del vertice
// SI inizializzano tutti a 0 e poi i vertici degli adiacenti vengono incrementati
InitQueue; // Inserisce in coda tutti i vertici con grado entrante = 0
while (Q != 0){
  v IN Q.HeadNDeque
  /* Nel caso in cui si voglia fare una map/fold che esegua l'ordinamento topologico
     Inseriamo la funzione */
  f(v);
  for (u in adj(v) do{
    D[u] --;
    if D[u] = 0 then Q.Enqueue(u);
  };
}
```

27.2.2 DFS

```
InitColor;
for v IN VertxSet
  if Color(v) = white then
    PI <- DFSVisit(G,v, PI);
  DFSVisit(G,v, PI);
  Color(v) = Gray
  for u IN adj(v) do
    if Color(u) = white then
      PI <- DFSVisit(G,v, PI);
    Color(v) = black;
```

```

PI.push(v);
// Può essere fatto anche in coda
// Varia solo il modo in cui deve essere letto il dato

```

27.2.3 Costruttore

Necessito di:

- Curr;
- Degree;
- Q (coda).

```

ComputeInDegree; // Calcolo il numero di gradi entranti
// Risparmio il calcolo dell'InDegree (che deve restare immutabile nel tempo)
for v IN Vertx do
  if Degre[v] = 0 then
    Q.Enqueue(v)
curr = Q.HeadNDequeue;

```

27.2.4 SuccessorOperator (++)

Dobbiamo aggiornare il Degree a partire da curr (che è ormai visitato, come si vede nell'ultima riga del costruttore)

```

for u IN adj(v) do
  Degree(u)--;
  if Degree(u) = 0 then
    Q.Enqueue(u);
// prima di farlo bisogna controllare che Q is Empty e quindi mettere curr a null ptr
if Q = Empty then
  curr = nullptr;
else
// Altrimenti fare questo assegnamento
  curr = Q.HeadNDequeue;

```

27.3 Iteratori ONLINE/OFFLINE

OFFLINE → nel momento in cui viene creato l'iteratore creiamo anche l'intera struttura che verrà poi sfruttata nella visita. Ritorna utile durante la visita dell'intero grafo (spendo più tempo nella costruzione per avere una maggiore velocità dopo).

ONLINE → questa operazione viene invece fatta al momento. Ritorna utile nel diluire il tempo di calcolo a cavallo di tutte le operazioni di Successor (presenta un tempo di costruzione minore, ed è preferita nel caso in cui non debba o non sia sicuro di dover visitare l'intero grafo).

27.4 MEMO

Esempio di applicazione dell'ordinamento topologico inverso → Calcolo delle componenti fortemente connesse.

Per avere il trasposto (InDegree e OutDegree) senza doverlo calcolare ogni volta (esplicitamente dichiarato nel grafo) si sfruttano gli indici (si usano dei flag che permettano di contenere il dato, che verrà usato da una funzione **trasponi**). Un caso di struttura che permette ciò è quella delle liste ortogonali.

Con la lista di adiacenza questo non è possibile.