Programmazione 2

Stefano Mavilla

Anno Accademico 2023/2024 - Canale F-N

1 Introduzione

1.1 Programmazione Orientata agli Oggetti

La Programmazione Orientata agli Oggetti è un paradigma di programmazione che scompone i problemi in Oggetti.

1.1.1 Oggetto

Un Oggetto è un'entità dove sono combinati gli attributi, che caratterizzano l'oggetto e ne descrivono le proprietà e i metodi, ovvero le funzionalità che l'oggetto mette a disposizione. Quando un oggetto viene creato, viene allocata memoria.

1.1.2 Classe

La Classe è una struttura che incapsula sia gli attributi che i metodi degli oggetti che rappresenta. Esse permettono di creare tipi di dati astratti e forniscono una interfaccia pubblica per interagire con gli oggetti definiti dalla classe. Quando una classe viene creata, non viene allocata memoria.

1.2 L'importanza dell'Astrazione

L'Astrazione è il processo di semplificazione della realtà che vogliamo modellare, rappresentando solo i dettagli caratteristici del sistema che stiamo considerando.

1.3 Incapsulamento e Occultamento

L'Incapsulamento è la proprietà degli oggetti di incorporare al loro interno sia gli attributi che i metodi, ovvero le caratteristiche e le funzionalità dell'oggetto.
L'Occultamento consiste nel nascondere all'esterno i dettagli implementativi dei metodi di un oggetto, rendendoli inaccessibili dall'esterno e consentendo l'interazione sono tramite l'interfaccia pubblica. Questo è permesso tramite gli specificatori d'accesso:
• **Pubblico**: accessibile da qualsiasi parte del programma;
• **Protetto**: accessibile solo all'interno della classe stessa e delle sue sottoclassi;
• **Privato**: accessibile solo all'interno della classe stessa.

1.4 Ereditarietà

L'Ereditarietà è lo strumento che permette la creazione di nuove classi Sottoclassi da una classe generatrice chiamata Superclasse o Classe Madre, ereditandone tutti gli attributi e i metodi. Queste relazioni tra classi si possono rappresentare con un grafo di gerarchia:

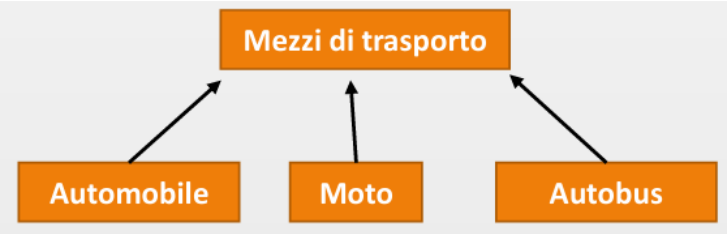


Figure 1: Esempio di Gerarchia tra Classi

La nuova sottoclasse si differenzia dalla classe Madre in due modi:
• Per estensione: aggiungendo nuovi attributi e metodi;
• Per ridefinizione: modificando i metodi ereditati, specificando una implementazione diversa di un metodo (override, overload);

1.5 Polimorfismo

Il polimorfismo indica la possibilità per i metodi di assumere implementazioni diverse all'interno della gerarchia delle classi.
Viene trattato nello specifico nel capitolo 4.

2 Elementi base del C++

2.1 Libreria iostream

Comprende tutti gli oggetti per input e output standard, ovvero:
• *cin*, oggetto che legge da standard **input** (da tastiera);
• *cout*, oggetto che stampa su standard **output** (sullo schermo);
• *cerr*, oggetto che stampa su standard **error** (senza buffer);
• *clog*, oggetto che stampa su standard **error** (con buffer);

2.2 Libreria fstream

Comprende tutte le funzioni per interagire con file, sia in lettura che in scrittura, ovvero:
• *open* e *close*, rispettivamente aprono e chiudono il file;
• *put* e *get*;
• *read*, per leggere da file **binario**;
• *write*, per scrivere su file **binario**;
Inoltre vengono usati gli stream **ifstream** e **ofstream** rispettivamente per leggere da file e scrivere su file.
La gestione dei file è trattata nel Capitolo 6.

2.3 Libreria cstring

Comprende tutte le funzioni del C per manipolare le stringhe, come ad esempio strlen, strcmp, strchr, strcpy, etc...

2.4 Namespace

I Namespace, o Spazio di Nomi, sono una caratteristica specifica del C++. Si tratta di un blocco definito dal programmatore che contiene definizioni di funzioni o variabili e si accede ad esse con "::".
Tipicamente si utilizza solamente il namespace std, scrivendo:

```
using namespace std;
```

2.4.1 Esempio

```
std :: cout << "Hello World" << endl;
```

2.5 Allocazione Dinamica della memoria

La memoria è suddivisa in 4 sezioni:
• L'area del codice, ovvero la memoria occupata dal codice del programma;
• L'area **statica**, che contiene le variabili statiche;
• Lo **Stack**, che contiene i record di attivazione delle funzioni
(variabili locali);
• L'**Heap**, che contiene le variabili allocate dinamicamente.

Per allocare e deallocare dinamicamente variabili in C++ si utilizzano rispettivamente **new** e **delete**.

2.5.1 Allocazione con New

La funzione **new** genera una variabile di un certo tipo assegnandole un blocco di memoria di dimensione opportuna e restituisce un puntatore alla variabile, accessibile se viene deferenziata.
Se non si tratta di un array:
• tipo* puntatore = new tipo;
Se si tratta di un array:
• tipo* puntatote = new tipo[dimensione];

Esempio:

```
char* p = new char[100];
```

2.5.2 Deallocazione con Delete

La funzione **delete** libera la memoria allocata dinamicamente affinché essa possa essere riallocata successiavemente, tuttavia non cancella il puntatore, che quindi può essere riutilizzato. Se non si tratta di un array:
• **delete** puntatore;
Se si tratta di un array:
• **delete** [] puntatore;

Esempio:

```
delete [] p;
```

2.6 Funzioni inline

Le funzioni **inline** servono per aumentare la velocità del programma, in quanto sono trattate come **funzioni macro** e obbligano il compilatore a ricopiare il codice della funzione in ogni punto che la funzione è chiamata.

Esse vengono utilizzate quando una funzione viene spesso richiamata ed il suo codice è breve, in quanto ogni ripetizione della funzione richiede memoria. Si inserisce **inline** prima del nome della funzione.

3 Costruzione di una Classe

3.1 Elementi fondamentali di una Classe

Per costruire correttamente una Classe è buona prassi:
- Assegnare un **nome adeguato** alla classe;
- Inserire gli attributi nella sezione **Private**;
- Inserire i metodi nella sezione **Public**;
- Inserire il **Costruttore** e il **Distruttore**;
- Inserire metodi **Getter** e metodi **Setter**;

3.1.1 Esempio di una Classe

```
class Persona{
    private: //attributi
        string nome;
        string cognome;
        int età;

    public:
        //costruttore
        Persona(string nome, string cognome, int età)
        : nome(nome), cognome(cognome), età(età) {}
        //distruttore
        ~Persona() {}

        //metodi Getter
        string getNome(){return nome;}
        string getCognome(){return cognome;}
        int getEtà(){return età;}

        //metodi Setter
        void setNome(string nome){this->nome = nome;}
        void setCognome(string cognome){this->cognome = cognome;}
        void setEtà(int età){this->età = età;}

        //metodi per l'interfaccia pubblica
        void stampa(){
            cout << nome << " " << cognome << " " << età << endl;
        }
};
```

3.2 Costruttore

Il Costruttore è un metodo che viene eseguito automaticamente alla creazione di un oggetto. Esso ha lo stesso nome della classe e può avere qualsiasi numero di parametri, tuttavia non restituisce alcun valore (nemmeno void).
Se presenta parametri, servono per essere passati all'atto della creazione dell'oggetto.
Se gli attributi sono delle costanti o son passati per riferimento, si usa il vettore di inizializzazione di membri, ovvero **:**
Il compilatore g++ crea automaticamente un costruttore di default (senza parametri) se non è presente, ma non inizializza i membri della classe a valori predefiniti.

Esempi di Oggetti di una Classe:
Prendendo come esempio base la classe Persona si può creare un oggetto staticamente:

```
Persona persona1("Stefano", "Mavilla", 20);
```

Oppure dinamicamente (Importantissimo deallocare la memoria dopo):

```
Persona* persona1 = new Persona("Stefano", "Mavilla", 20);
```

3.3 Distruttore

Il Distruttore è un metodo che viene eseguito automaticamente per distruggere un oggetto e liberare la memoria occupata da quest'ultimo. Inoltre:
• Ha lo stesso nome della classe preceduto dal simbolo ∼;
• Non ritorna alcun valore e non accetta parametri;
• Non può essercene più di uno per classe;
Se manca il distruttore, il compilatore g++ ne crea automaticamente uno vuoto, tuttavia è buona prassi creare un distruttore per ogni classe, specialmente se vengono istanziati **dinamicamente** oggetti di quella classe stessa.

Per richiamare infatti il **distruttore** di una classe bisogna usare **delete**, in base a dove è stato allocato l'oggetto da cancellare.
Infatti, se è stata allocata memoria nel costruttore della classe, bisognerà deallocarla nel distruttore della classe stessa, mentre ad esempio se è stata allocata nel main del programma, allora le **delete** dovranno essere nel main. **Esempio di Distruttore**:

```
~Persona() {}
```

Esempio di deallocazione della memoria:

```
delete persona1;
```

Questa **delete** richiamerà il distruttore sopracitato, che cancellerà l'oggetto precedentemente istanziato.

3.4 Metodi Getter

I Metodi Getter sono metodi utilizzati per accedere agli attributi privati di una classe dall'esterno affinché essi possano essere letti.
Ogni metodo Getter restituisce il tipo dell'attributo a cui vogliamo accedere, non prende alcun parametro e ritorna l'attributo.

Esempio di Metodo Getter:

```
string getNome(){return nome;}
```

3.5 Metodi Setter

I Metodi Setter sono metodi utilizzati per accedere agli attributi privati di una classe dall'esterno affinché essi possano essere modificati.
Ogni metodo Setter restituisce **void** e prende come unico parametro una variabile dello stesso tipo dell'attributo che vogliamo modificare con nome diverso, che sovrascriverà quest'ultimo.

Esempio:

```
void setNome(string n){nome = n;}
```

3.5.1 This

La parola chiave **This**, tipicamente usata nei metodi Setter o nel Costruttore, è un puntatore **implicito** che punta all'oggetto stesso sulla quale è utilizzata.
Si può usare **this** per usare lo stesso nome.

Esempio :

```
void setNome(string nome){this->nome = nome;}
```

4 Sottoclassi ed Ereditarietà

4.1 Definizione ed Esempio

Una classe derivata, o sottoclasse, eredita dalla classe madre sia gli attributi che i metodi.

```
class Operaio : public Persona{
private:
    float stipendio;
public:
    Operaio(string nome,string cognome,int età,float stipendio):
    Persona(nome, cognome, età), stipendio(stipendio) {}
    ~Operaio() {}

    float getStipendio(){return stipendio;}

    void setStipendio(float stipendio){
        this->stipendio = stipendio;
    }
};
```

4.2 Tipi di Ereditarietà

L'ereditarietà è caratterizzata da tre Tipi di Ereditarietà o Specificatori di Accesso, ovvero:
• **public**, il più usato.
• **protected**, mantiene accessibili i membri pubblici e protetti nella classe derivata occultandoli all'esterno.
• **private**, il Tipo di Ereditarietà usato di default se non specificato dal programmatore, privatizza gli elementi pubblici della superclasse nella classe derivata.

Tipo di ereditarietà	Accesso a membro classe base	Accesso a membro classe derivata
public	public	public
	protected	protected
	private	inaccessibile
protected	public	protected
	protected	protected
	private	inaccessibile
private	public	private
	protected	private
	private	inaccessibile

4.3 Costruttore della classe derivata

Il Costruttore nella classe derivata crea l'oggetto solamente dopo che l'oggetto della classe madre è stato creato. Quest'ultimo verrà convertito nell'oggetto della classe derivata.

4.4 Distruttore della classe derivata

Il Distruttore nella classe derivata non viene ereditato dalla classe madre e ne viene generato uno di default quando il corrispondente costruttore crea un oggetto della classe derivata.
Al contrario del costruttore, il distruttore della classe derivata agisce prima di quello della classe madre.

4.5 Binding Statico e Dinamico

Il **binding** è la connessione tra la chiamata di funzione e il codice che la implementa. Può essere **statico** o **dinamico**:
• **Statico**: La connessione avviene nella fase di compilazione ed è il binding impostato di default per C++.
• **Dinamico**: La connessione avviene nella fase di esecuzione tramite il valore di un puntatore di una classe base. Offre più flessibilità rispetto al binding statico ma è meno efficiente.
Per specificare che il binding dev'essere dinamico si usa *virtual* prima di una dichiarazione di un metodo.

4.6 Metodi Virtuali

Un metodo è **virtuale** se viene anteposto *virtual* alla dichiarazione della funzione stessa. Queste possono essere ridefinite in sottoclassi tramite **overriding**. Se un metodo è **virtuale pura**, esso renderà la classe **astratta** e non istanziabile e il metodo **dovrà** essere ridefinito in una sottoclasse.

Esempio di metodo virtuale e virtuale puro:

```
virtual void stampa(){ //virtuale e utilizzabile
    cout << "Esempio." << endl;
}
virtual void stampa() = 0; //virtuale puro
```

4.7 Polimorfismo

Il polimorfismo permette alle **sottoclassi** di assumere implementazioni diverse di metodi già implementate nella classe madre.
Il polimorfismo può essere **statico** o **dinamico**:

4.7.1 Polimorfismo Statico (Overloading)

Si verifica quando due o più metodi nella stessa classe hanno lo stesso nome ma parametri diversi (numero, tipo o ordine). Il compilatore determina quale metodo invocare in base ai parametri passati.

Esempio di Overloading:

```
class Persona{
    public:
        void stampa(string nome){
            cout << nome << endl;
        }

        void stampa(int età){
            cout << età << endl;
        }
};

int main(){
    Persona pers;
    pers.stampa("Stefano"); //chiamo la versione con la stringa.
    pers.stampa(20); //chiamo la versione con l'intero.
}
```

4.7.2 Polimorfismo Dinamico (Overriding)

Si verifica quando una sottoclasse ridefinisce un metodo ereditato dalla classe base. Una funzione dichiarata come **virtuale** nella classe base può essere sovrascritta (tramite **override**) nelle classi **derivate**.

Esempio di Overriding:

```
class Base{
    public:
        virtual void stampa(){
            cout << "Stampo da classe base." << endl;
        }
};

class Derivata : public Base{
    public:
        void stampa() override{
            cout << "Stampo da classe derivata." << endl;
        }
};

int main(){
    Base* ogg = new Derivata();
    ogg->stampa(); //stamperà dalla classe derivata.
}
```

4.8 Friend

La parola chiave **friend** permette di accedere a metodi e attributi di una classe da un metodo esterno o da un'altra classe, indipendentemente dallo **specificatore d'accesso**, rompendo quindi l'**incapsulamento**. **Esempio**:

```
friend class Persona;
```

4.9 Casting

4.9.1 Static Cast

Lo **static cast** è un cast che permette di convertire tipi **compatibili**. Esso avviene in fase di **compilazione**, affinché il compilatore possa controllare che la conversione abbia senso, come ad esempio tra:
• Tipi numerici, ad esempio:

```
int n = 20;
float m = static_cast<float>(n);
```

• Puntatori o riferimenti di tipi in gerarchie di classi, ad esempio:

```
class Base{
    public:
        int n;
        Base(int n) : n(n){}
};
class Derivata : public Base{
    public:
        Derivata(int n) : Base(n) {}
};

Base* b1 = new Derivata(20);
Derivata* d1 = static_cast<Base*>(b1);
```

• Da **void*** a un altro tipo di puntatore.

Lo static cast non produce overhead quindi è molto veloce, tuttavia non possiede alcun controllo in **esecuzione**, quindi può provocare a comportamenti non definiti nel programma.

4.9.2 Dynamic Cast

Il **dynamic Cast** è un cast che permette di convertire puntatori o riferimenti di tipi in gerarchie di classi. Esso è più sicuro dello **static cast** in quanto il controllo è effettuato in **esecuzione** e, in caso di impossibilità di conversione, esso restituisce **nullptr**. Esempio:

```
class Base{
    public:
        virtual ~Base() {} // necessario il polimorfismo
        virtal void stampa(){
            cout << "Oggetto classe Base" << endl;
        }
};
class Derivata : public Base{
    public:
        void stampa() override {
            cout << "Oggetto classe Derivata" << endl;
        }
};
int main(){
    Base* ogg1 = new Base();
    Derivata* ogg2 = new Derivata();
    Base* array[] = {ogg1, ogg2};

    for(int i = 0; i < 2; i++){
        Base* ogg[] = array[i];
        if(Derivata* ogg2 = dynamic_cast<Derivata*>(ogg){
            ogg2->stampa();
        } else {
            ogg1->stampa();
        }
    }
}
```

Il dynamic cast provoca overhead e quindi è più lento dello static cast e funziona solo su tipi **polimorfici**, tuttavia non può portare a comportamenti imprevisti del programma.

5 Template

I Template sono funzioni e classi **generiche**, implementate per un tipo di dato definito in seguito.

5.1 Template di Funzioni

I Template di Funzioni sono funzioni alle quali possono essere passati parametri di **ogni** tipo, un insieme indeterminato di funzioni sovraccaricate che descrive l'algoritmo specifico di una funzione generica.
Ogni funzione specifica di questo insieme è una **istanza** del template di funzione e viene prodotta automaticamente quando necessario.
Per dichiarare un template di funzioni si usa la parola chiave *template* e subito dopo il nome del dato generico con *typename*.

Esempio di un Template di Funzioni:

```
template <typename T>
void scambia(T &v1, T &v2){
    T v3 = v1;
    v1 = v2;
    v2 = v3;
}
```

5.2 Template di Classi

I Template di Classi permettono di definire classi parametriche in grado di gestire diversi tipi di varaibili. Essi possono rappresentare strutture dati come array, pile, code, liste ordinate, alberi, grafi ed ecc.

Esempio di un Template di Classi:

```
template <typename T>
class Addizione{
    private:
        T v1;
        T v2;

    public:
        T somma(){
            return v1 + v2;
        }
};
```

6 Gestione dei File

6.1 Lettura da File

Su C++ per leggere il contenuto di un file dobbiamo includere la libreria $< fstream >$, che comprende lo stream **ifstream**, per creare variabili stream file input. Si useranno le funzioni **open** e **close** per rispettivamente aprire e chiudere il file. Tramite un esempio verranno illustrati i passaggi essenziali da eseguire per leggere il contenuto di un file.

6.1.1 Esempio

Supponiamo che nella stessa directory del file .cpp sia presente un file "input.txt" contentente "ciao 10". Per leggere il contenuto del file e usarlo sul file .cpp dovremo:

```
// Passo 1: dichiarare una variabile input file stream
ifstream inFile;

//Passo 2: aprire il file con open() assegnandogli il nome
//e l'estensione del file input
inFile.open("input.txt");

//Passo 3: eseguire un controllo (opzionale ma utile)
if(!inFile.is_open()){
    cerr << "Errore: Impossibile aprire il file." << endl;
    exit(EXIT_FAILURE);
}

//Passo 4: Creare variabili con cui memorizzare il
//contenuto del file
string parola;
int n;
//Passo 5: Indicare in che ordine il contenuto del file
//deve essere letto
inFile >> parola >> n;
//Passo 6: Chiudere il file con close().
inFile.close();
```

6.2 Scrittura su File

Sempre tramite un esempio, supponiamo di voler scrivere su un file "output.txt" una stringa. L'apertura del file avverrà tramite la creazione dello stream **ofstream** mentre la chiusura avverrà con **close**.

6.2.1 Esempio

```
string parola;
cout << "Inserisci una parola: " << endl;
cin >> parola;

// Passo 1: dichiarare una variabile output file stream
ofstream outFile("output.txt");

// Passo 2: inserire il contenuto nel file
outFile << parola;

// Passo 3: chiudere il file.
outFile.close();
```

7 Complessità Algoritmica

La Complessità misura l'efficenza di un algoritmo in termini di **tempo** e **spazio**. Si distingue in:

7.1 Complessità Spaziale

La Complessità Spaziale misura il **quantitativo di memoria necessaria** per eseguire l'algoritmo (esclusa la memoria occupata dall'input).

7.2 Complessità Temporale

La Complessità Temporale misura il **quantitativo di tempo impiegato** dall'algoritmo a terminare in funzione della dimensione dell'input.
Il costo in termini di tempo di una funzione è dato dal costo del suo corpo.
Nonostante la grandezza dell'input n sia un numero fisso, vi sono diverse circostanze (come la disposizione degli n dati) che distinguono tre casi per la complessità temporale:
• **Caso Migliore**: Input ideale, costo minimo tra tutte le istanze del problema dato;
• **Caso Medio**: Costo mediato tra tutte le istanze del problema dato;
• **Caso Peggiore**: Costo massimo tra tutte le istanze del problema dato;

7.3 Notazione Asintotica

Per valutare l'efficienza di un algoritmo al variare delle dimensioni dell'input utilizziamo la **Notazione Asintotica**.
Non calcoliamo specificamente il tempo o lo spazio impiegato da un algoritmo, ma soltanto come questi parametri crescono alla variazione dell'input.

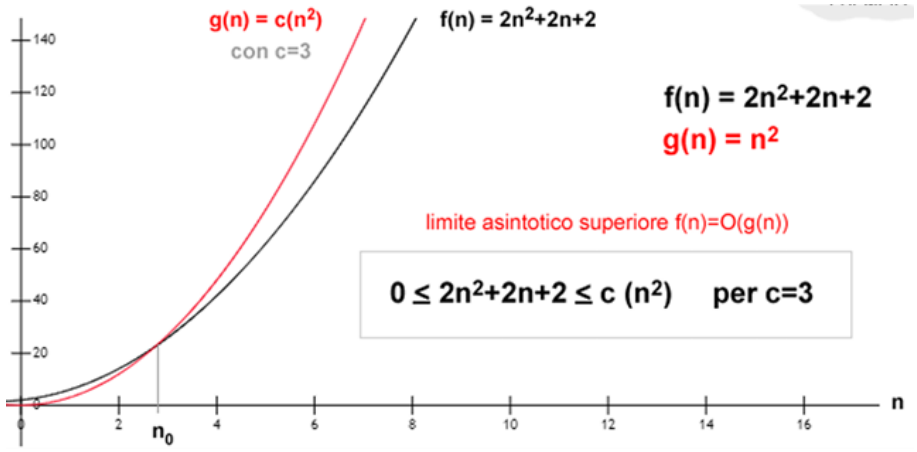
7.4 Notazione O-grande

La Notazione **O**-grande descrive un **limite superiore** alla crescita di una funzione. Una funzione $f(n)$ è compresa nell'insieme delle funzioni $\mathrm{O}(g(n))$ se esistono delle costanti positive c, n_0 tali che:
$0 \geq f(n) \geq c \cdot f(n)\ \forall n > n_0$

Se $f(n) \in O(g(n))$, allora $f(n)$ crescerà al massimo come $g(n)$.

Osservazione: La notazione **O**-grande rappresenta **una** delimitazione asintotica superiore alla complessità dell'algoritmo, e non **la** delimitazione asintotica superiore. Se per una funzione $T(n)$ sono note più delimitazioni asintotiche superiori, allora è da preferire quella più piccola.

7.4.1 Esempio

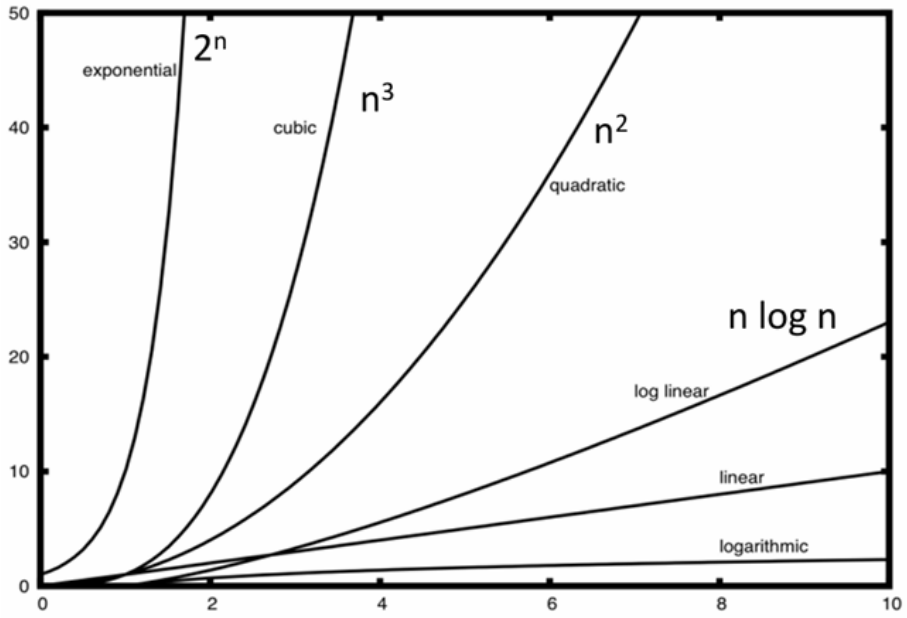


7.5 Ordine della complessità degli algoritmi

Qui sono elencati i vari **ordini di grandezza** della Complessità che un algoritmo può avere:

	Ordine	**Funzione Esempio**
$O(1)$	Costante	$f(n) = c$
O(log n)	Logaritmico	$f(n) = log\ n$
$O(n)$	Lineare	$f(n) = c \cdot n$
$O(n\ log\ n)$	Pseudo-Lineare	$f(n) = n \cdot log\ n$
$O(n^2)$	Quadratico	$f(n) = c \cdot n^2$
$O(n^3)$	Cubico	$f(n) = c \cdot n^3$
$O(n^k)$	Polinomiale	$f(n) = c \cdot n^k$
$O(c^n)$	Esponenziale	$f(n) = c^n$
$O(n!)$	Fattoriale	$f(n) = n!$

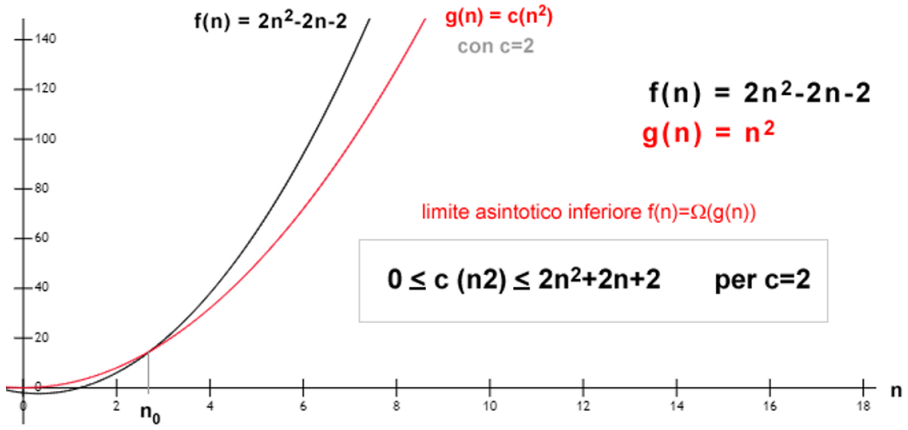
Qui invece come cresce ogni funzione con relativo Ordine di grandezza:



7.6 Notazione Ω-grande

La notazione Ω-grande descrive il **limite inferiore** per la crescita di una funzione. Una funzione $f(n)$ è compresa nell'insieme delle funzioni $\Omega(g(n))$ se esistono delle costanti positive c, n_0 tali che: $f(n) \leq c \cdot g(n) \quad \forall n > n_0$.

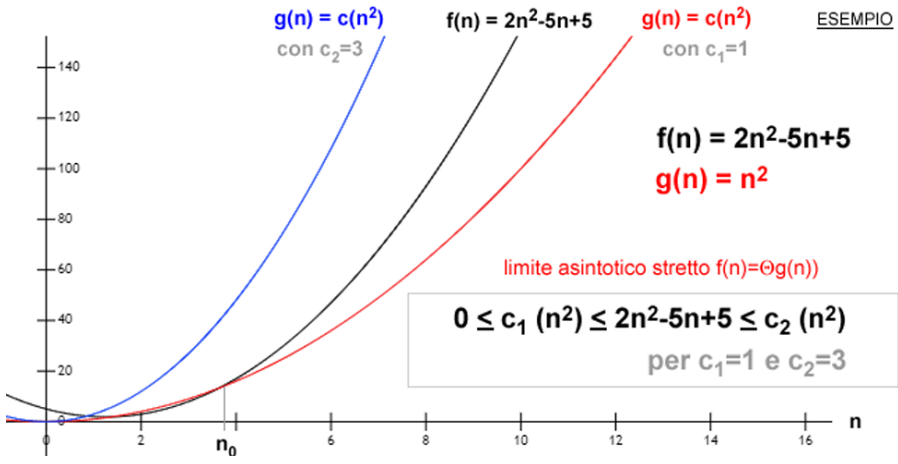
7.6.1 Esempio



7.7 Notazione Θ-grande

La notazione Θ-grande descrive un **limite stretto** (ovvero esattamente proporzionale) per la crescita di una funzione. Una funzione $f(n)$ è compresa nell'insieme delle funzioni $\Theta(g(n))$ se esistono delle costanti **positive** c_1, c_2 e n_0 tali che: $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n > n_0$
oppure: $f(n) = O(g(n)) \quad e \quad g(n) = O(f(n))$.

7.7.1 Esempio



7.8 Delimitazioni alla Complessità di P

Dato un problema **P** e un algoritmo **A** che lo risolve:
• Se **A** impiega tempo $t(n)$ allora diremo che $O(t(n))$ è un **limite superiore**;
• Se nessun algoritmo **A** può risolvere **P** in tempi inferiori a $t(n)$ allora $\Omega(t(n))$ è un **limite inferiore**;
• Se i due limiti coincidono allora l'algoritmo **A** si dirà **ottimale**.

Se un algoritmo si dice **ottimale**, esso avrà complessità $O(t(n))$ e la complessità del problema **P** sarà $\Theta(t(n))$.

7.9 Analisi di Algoritmi

• $O(1)$: complessità di una funzione o blocco di istruzioni ciascuna di costo O(1), che non contengono cicli, ricorsione o chiamate ad altre funzioni non costanti, un esempio è l'istruzione if.

• $O(n)$: complessità di un ciclo quando le sue variabili (es. contatore) sono incrementate/decrementate di una quantità costante.

```
int c; // costante positiva
for(int i = 0; i <= n; i += c){              O(n)
    //espressioni di costo O(1);
}
```

• $O(n^c)$: la complessità di cicli annidati è uguale al numero di volte in cui le istruzioni del ciclo interno vengono eseguite.

```
int c; // costante positiva
for(int i = 1; i <= n; i += c){
    for(int j = i; j <= n; j += c){          O(n^2)
        //espressioni di costo O(1);
    }
}
```

• $O(log\ n)$: complessità di un ciclo quando le sue variabili sono incrementate/decrementate moltiplicandole/dividendole per una costante.

```
int c; // costante positiva
for(int i = 1; i <= n; i *= c){              O(log n)
    //espressioni di costo O(1);
}
```

• $O(log\ (log\ n))$: complessità di un ciclo quando le sue variabili sono incrementate/decrementate esponenzialmente.

```
int c; // costante positiva > 1
for(int i = 2; i <= n; i = pow(i,c)){        O(log log n)
    //espressioni di costo O(1);
}
```

7.9.1 Esempi di Analisi della Complessità

Esempio con codice:

```
void func(int n){
    int count = 0;
    for(int i = n/2; i <= n; i++){              O(n)
        for(int j = 1; i <= n; j = 2*j){        O(log n)
            for(int k = 1; k <= n; k = k*2){    O(log n)
                count++;
            }
        }
    }
}                                               O(n log^2 n)
```

Esempio con due funzioni:

$$T1 = 5n^2 + 2n - 10$$
$$T2 = 5\sqrt{n} + 22$$
$$G = T1 + T2 \quad \longrightarrow \quad O(n^2)$$

8 Algoritmi di Ricerca

8.1 Ricerca Sequenziale/Lineare

Dato un array di n elementi, l'algoritmo ricerca l'elemento "chiave" confrontandolo con gli elementi dell'array. L'algoritmo ha complessità $\mathbf{O(1)}$ se l'elemento cercato è il primo dell'array (caso migliore) e ha complessità $\mathbf{O}(n)$, nel caso medio e peggiore, in quanto deve fare n confronti.

Codice dell'algoritmo:

```
int ricercaLineare(int* array, int n, int key){
    for(int i = 0; i < n; i++){
        if(array[i] == key) return i;
    }
    return -1; //elemento non trovato
}
```

8.2 Ricerca Binaria

Dato un array **ordinato** di n elementi, l'algoritmo ricerca l'elemento *chiave* partendo dal centro e successivamente si sposta nel *sottoarray* destro o sinistro, dimezzando ad ogni ciclo il numero di elementi da confrontare.
L'algoritmo ha complessità $\mathbf{O(1)}$ se l'elemento cercato è l'elemento al centro dell'array (caso migliore) e ha complessità $\mathbf{O}(log\ n)$, nel caso medio e peggiore, in quanto la suddivisione del sottoarray può arrivare fino a $log\ n$ passi.

Codice dell'algoritmo:

```
int ricercaBinaria(int* array, int low, int high, int key){
    while(low <= high){
        int mid = low + (high - low)/2; //elemento centrale
        if(array[mid] == n){
            return mid; //valore trovato, restituisce indice
        }
        if(array[mid] < n)  low = mid + 1;
        if(array[mid] > n)  high = mid - 1;
    }
    return -1; //elemento non trovato
}
```

9 Algoritmi di Ordinamento

9.1 Bubble Sort

Il Bubble Sort è un algoritmo di ordinamento iterativo che ordina un array confrontando ogni elemento con il suo successivo e li scambia se sono in ordine sbagliato. In tutti e tre i casi la complessità dell'algoritmo è $\mathbf{O}(n^2)$.

9.1.1 Codice dell'algoritmo

:

```
BubbleSort(int* array, int n){
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n-i-1; j++){
            if(array[j] > array[j+1]){
                int temp = array[j];
                array[j] = array[j+1];
                array[j+1] = temp;
            }
       }
    }
}
```

L'algoritmo può essere tuttavia modificato e migliorato, aggiungendo una **flag** per evitare scambi inutili nell'array.

9.1.2 Codice dell'algoritmo migliorato

:

```
BubbleSort(int* array, int n){
    bool scambio = true;
    while(scambio){
        scambio = false;
        for(int i = 0; i < n - 2; i++){
            if(array[i] > array[i+1]){
                int temp = array[i];
                array[i] = array[i+1];
                array[i+1] = temp;
                scambio = true;
            }
        }
    }
}
```

Se l'array è già ordinato allora la complessità nel caso migliore si riduce a $\mathbf{O(n)}$ ma rimane $\mathbf{O}(n^2)$ nel caso medio e peggiore.

9.2 Selection Sort

Il Selection Sort è un algoritmo di ordinamento iterativo che ordina un array dividendo quest'ultimo in una parte ordinata (inizialmente vuota) e una non ordinata. L'algoritmo cercherà, per ogni iterazione, il minimo nella parte non ordinata e, una volta trovato, lo sposterà nella parte ordinata.
In tutti e tre i casi la complessità dell'algoritmo è $\mathbf{O}(n^2)$.

9.2.1 Codice dell'Algoritmo

```
SelectionSort(int* array, int n){
    for(int i = 0; i < n - 1; i++){
        int min = i;
        for(int j = i + 1; j < n; j++){
            if(array[j] < array[min]){
                min = j;
            }
        }
        int temp = array[i];
        array[i] = array[min];
        array[min] = temp;
    }
}
```

9.3 Insertion Sort

L'Insertion Sort è un algoritmo di ordinamento iterativo che ordina un array dividento quest'ultimo in una parte ordinata e una non ordinata. L'algoritmo sposterà, per ogni iterazione, il primo elemento della parte non ordinata in quella ordinata nella giusta posizione.
Nel caso migliore, ovvero quando l'array è già ordinato, la complessità è **O(n)**, tuttavia nel caso medio e nel caso peggiore è **O**(n^2).

9.3.1 Codice dell'algoritmo

```
InsertionSort(int* array, int n){
    for(int i = 1; i < n; i++){
        int key = array[i];
        int j = i - 1;
        while(j >= 0 && array[j] > key){
            array[j+1] = array[j];
            j--;
        }
        array[j+1] = key;
    }
}
```

L'algoritmo può essere modificato e migliorato, affinché invece di usare la ricerca **sequenziale** (complessità **O(n)**) usi la ricerca **binaria** (complessità **O**($log\ n$)), riducendo la complessità della ricerca della posizione corretta da **O**(n^2) a **O**($nlog\ n$).
Tuttavia ogni elemento deve essere comunque spostato di una posizione e quindi il costo totale nel caso medio e peggiore rimane **O**(n^2).

9.4 Merge Sort

Il **Merge Sort** è un algoritmo di ordinamento **ricorsivo** che implementa il paradigma *divide et impera*.
L'algoritmo divide l'array di n elementi in due sotto-array di lunghezza $n/2$, e continuerà a dividere i due sotto-array **ricorsivamente**, fino ad ottenere sotto-array di un solo elemento, già ordinati per definizione. Successivamente gli elementi vengono uniti ordinatamente tramite la procedura **merge()**.

9.4.1 Codice dell'algoritmo

```
MergeSort(int* array, int min, int max){
    if(min < max){
        int mid = (min + max)/2;
        MergeSort(array, min, mid);
        MergeSort(array, mid + 1, max);
        Merge(array, min, mid, max);
    } else {
        return;
    }
}
```

9.4.2 Codice della procedura Merge

```
Merge(int* array, int min, int mid, int max){
    int n1 = mid - min + 1; //grandezze dei subarray
    int n2 = max - mid;
    int* leftSubArray = new int[n1 + 1];
    int* rightSubArray = new int[n2 + 1];
    for(int i = 0; i < n1; i++){ //riempio il leftSubArray
        leftSubArray[i] = array[min + i];
    }
    for(int j = 0; j < n2; j++){ //riempio il rightSubArray
        rightSubArray[j] = array[mid + j + 1];
    }
    leftSubArray[n1] = INT_MAX; //sentinelle
    rightSubArray[n2] = INT_MAX;
    int i = 0; //indici per riempire l'array di partenza
    int j = 0;
    for(int k = min; k <= max; k++){
        if(leftSubArray[i] <= rightSubArray[j]){
            array[k] = leftSubArray[i];
            i++;
        } else {
            array[k] = rightSubArray[j];
            j++;
        }
    } //libero memoria
    delete[] rightSubArray;
    delete[] leftSubArray;
}
```

La complessità dell'algoritmo è $\mathbf{O}(nLog\ n)$ in ogni caso, in quanto l'algoritmo divide sempre le sequenze a metà impiegando tempo $\mathbf{O}(log\ n)$ e le unisce ordinatamente impiegando tempo $\mathbf{O}(n)$. La complessità spaziale è $\mathbf{O}(n)$.

9.5 Quick Sort

Il **Quick Sort** è un algoritmo di ordinamento **ricorsivo** che implementa il paradigma *divide et impera*.
L'algoritmo divide l'array di n elementi in 3 partizioni:
• Una partizione **centrale** che contiene un solo elemento detto **pivot**;
• Una partizione **sinistra** che contiene tutti gli elementi **minori** del **pivot**;
• Una partizione **destra** che contiene tutti gli elementi **maggiori** del **pivot**.
L'algoritmo si applica ricorsivamente sulle partizioni **sinistra** e **destra** dell'array fino a quando non è ordinato. La scelta del **pivot** è casuale.

9.5.1 Codice della procedura Partition

```
int Partition(int* array, int min, int max){
    int pivot = array[max]; //selezione del pivot
    int i = min - 1; //indice
    for(int j = min; j < max; j++){
        if(array[j] <= pivot){
            i++;
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    } //scambio il pivot con l'elemento ad i+1;
    int temp = array[i+1];
    array[i+1] = array[max];
    array[max] = temp;
    return i+1; //ritorno l'indirizzo del pivot
}
```

9.5.2 Codice dell'algortimo

```
QuickSort(int* array, int min, int max){
    if(min < max){
        int mid = Partition(array, min, max);
        QuickSort(array, min, mid - 1);
        QuickSort(array, mid + 1, max);
    } else {
        return;
    }
}
```

La complessità dell'algoritmo è $\mathbf{O}(nLog\ n)$ per il caso **migliore**, in quanto il partizionamento dei due sottoproblemi è $n/2$, e per il caso **medio**, in quanto anche con una ripartizione spropozionata ad ogni livello di ricorsione, l'algoritmo impiega tempo $\mathbf{O}(nLog\ n)$.

Tuttavia nel caso **peggiore**, ovvero quando una partizione ha 0 elementi e l'altra $n-1$, il partizionamento impiega $\mathbf{O}(n)$ e, se questo caso si verifica ad ogni chiamata ricorsiva, l'algoritmo impiega tempo $\mathbf{O}(n^2)$.

10 Strutture Dati

10.1 Classe Nodo

Il nodo è una struttura **autoreferenziale**. Ogni Struttura Dati possiede nodi e, nelle Pile, Code e Liste Singolarmente Concatenate, il nodo ed è formato dalla parte **data** e dal puntatore **next**.

10.1.1 Codice della Classe Nodo

```
class Nodo{
    public:
        int data;
        Nodo* next;

    private:
        Nodo(int data, Nodo* next): data(data),next(next);
        ~Nodo() {}

        int getData(){return data};
        Nodo* getNext(){return next};

        void setData(int data){this->data = data;}
        void setNext(Nodo* next){this->next = next;}
}
```

10.2 Pila

La Pila è una struttura dati LIFO (Last In First Out), dove l'operazione di inserimento (in testa) si chiama **Push** e quella di cancellazione (dalla testa) **Pop**, entrambe con complessità **O(1)**.

La Pila utilizza un puntatore **Head** (che inizialmente punterà a **NULL**) per mantenere il riferimento al nodo in testa e un intero **size** per contare il numero di nodi della Pila (inizialmente a 0).

Pila Vuota:

$$head \longrightarrow \textbf{NULL}$$

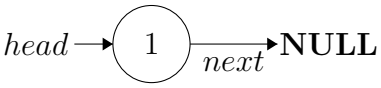
Codice per inizializzare una Classe Pila:

```
class Pila{
    private:
        Nodo* head;
        int size;
    public:
        Pila(): head(nullptr), size(0) {}
        ~Pila() {}
}
```

10.2.1 Push

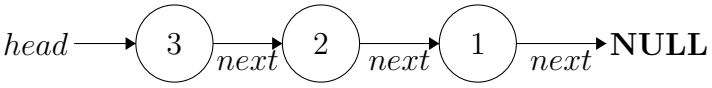
L'operazione di inserimento **Push** nella Pila avviene in due casi diversi:
• Pila vuota: il nodo (1) inserito punterà a **NULL** mentre il puntatore *head* punterà al nodo (1);

Pila con un solo nodo:



• Pila non vuota: il nodo (3) inserito prima punterà al nodo (2) mentre il puntatore *head* punterà il nodo (3), dove prima puntava il nodo (2).

Pila con più nodi:



Codice dell'algoritmo:

```
void push(int n){
    Nodo* newnode = new Nodo(n, nullptr);
    if(head == nullptr){
        head = newnode;
    } else {
        newnode->setNext(head);
        head = newnode;
    }
    size++;
}
```

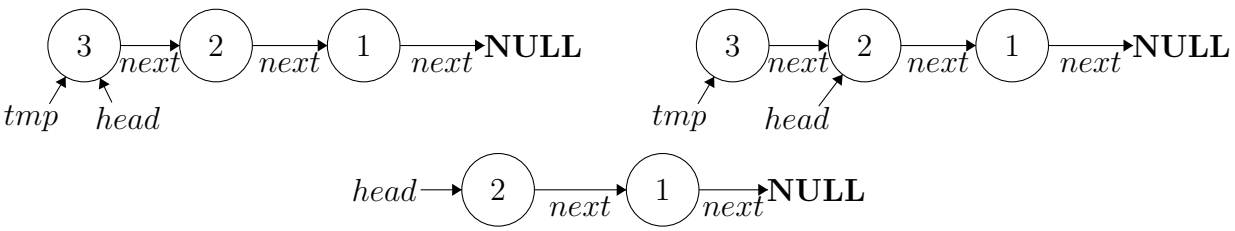
10.2.2 Pop

L'operazione di cancellazione **Pop** utilizza un puntatore a nodo *tmp*, che punterà al nodo puntato da *head*. *Head* punterà al nodo successivo e infine verrà cancellato il nodo puntato da *tmp*. **Pop** può restituire **void** o il contenuto del nodo estratto.

Codice dell'algoritmo:

```
void pop(){
    Nodo* tmp = head;
    head = head->getNext();
    delete tmp;
    size--;
}
```

Rappresentazione Grafica della Pop:



10.3 Coda

La Coda è una struttura dati FIFO (First In First Out), dove l'operazione di inserimento (in coda) si chiama **EnQueue** mentre quella di cancellazione (dalla testa) **DeQueue**, entrambe con complessità **O(1)**.

La Coda utilizza un puntatore **Head** per mantenere il riferimento al nodo in testa, un puntatore **Tail** per mantenere il riferimento allla coda (entrambi i puntatori inizialmente punteranno a **NULL**) e un intero **size** per contare il numero di nodi della Coda (inizialmente a 0).

Coda Vuota:

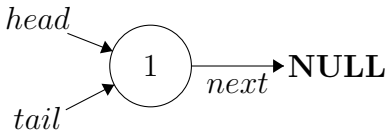
Codice per inizializzare una Classe Coda:

```
class Coda{
    private:
        Nodo* head;
        Nodo* tail;
        int size;
    public:
        Coda(): head(nullptr), tail(nullptr), size(0) {}
        ~Coda() {}
}
```

10.3.1 EnQueue

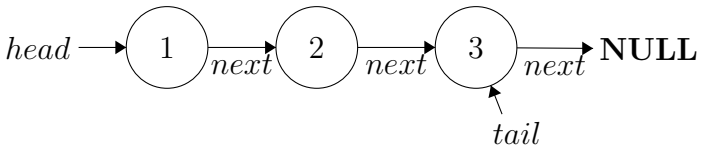
L'operazione di inserimento **EnQueue** nella Coda avviene in due casi diversi:
• Coda vuota: il nodo (1) inserito punterà a **NULL** mentre i puntatori *head* e *tail* punteranno al nodo (1);

Coda con un solo nodo:



• Coda non vuota: il nodo (3) inserito (che punterà a **NULL**), sarà puntato prima dal nodo (2) (puntato da *tail*) e poi da *tail* stesso.

Coda con più nodi:



Codice dell'algoritmo:

```
void EnQueue(int n){
    Nodo* newnode = new Nodo(n, nullptr);
    if(tail == nullptr){
        tail = head = newnode;
    } else {
        tail->setNext(newnode);
        tail = newnode;
    }
    size++;
}
```

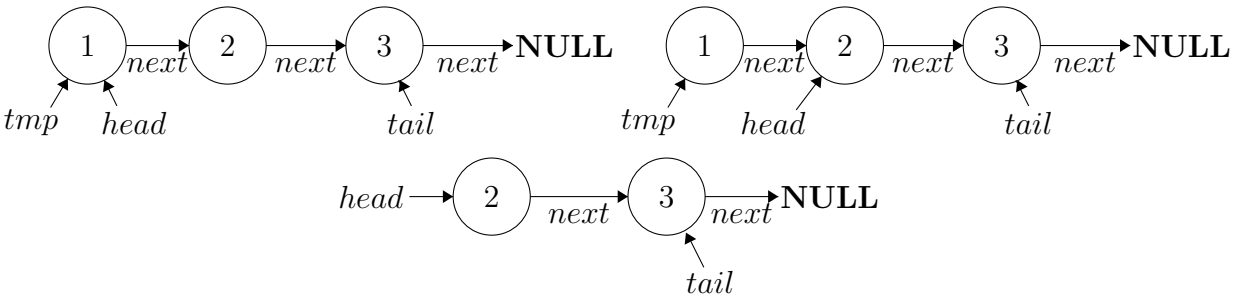
10.3.2 DeQueue

L'operazione di cancellazione **DeQueue** utilizza un puntatore a nodo *tmp*, che punterà al nodo puntato da *head*. *Head* punterà al nodo successivo e infine verrà cancellato il nodo puntato da *tmp*. **DeQueue** può restituire **void** o il contenuto del nodo estratto.

Codice dell'algoritmo:

```
void DeQueue(){
    Nodo* tmp = head;
    if(head == tail){
        head = tail = nullptr;
    } else {
        head = head->getNext();
        delete tmp;
    }
    size--;
}
```

Rappresentazione Grafica della DeQueue:

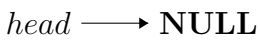


10.4 Lista Singolarmente Concatenata

Una Lista concatenata è una struttura dati i cui nodi sono disposti in ordine lineare. In una lista è possibile inserire nodi sia in testa che in coda, ma è anche possibile **inserire** i nodi in modo **ordinato**.
Stessa cosa riguarda la **cancellazione** di un nodo da una lista.

Una Lista Concatenata utilizza un puntatore **Head** per mantenere il riferimento al nodo in testa (che inizialmente punta a **NULL**) e un intero **size** per numero di nodi della Lista (inizialmente a 0).

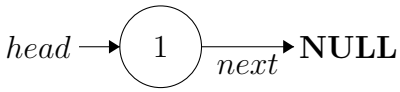
Lista Vuota:



Codice per inizializzare una Classe Lista:

```
Classe Lista{
    private:
        Nodo* head;
        int size;
    public:
        Lista(): head(nullptr), size(0) {}
        ~Lista() {}
}
```

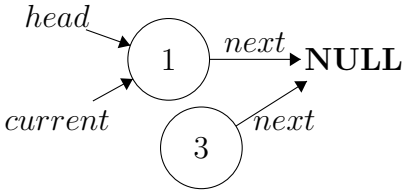
Lista con un solo nodo:



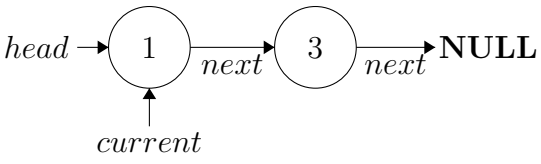
10.4.1 Insert (Inserimento Ordinato)

L'Inserimento Ordinato in una Lista Concatenata, chiamato **Insert**, prende il nodo da inserire e scorre ogni elemento della lista fino a quando non trova la corretta posizione.

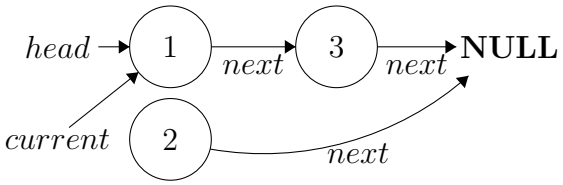
Immaginiamo di inserire un nodo (3). Essendo $3 > 1$ allora dovrà inserito dopo il nodo (1). Allora creiamo un puntatore *current* che punterà inizialmente allo stesso nodo a cui punta *head*, in questo caso il nodo (1).



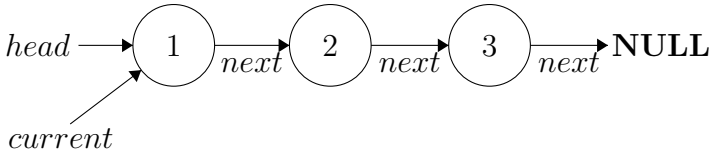
Il puntatore *current*, tramite vari confronti, determinerà la posizione giusta in cui inserire il nodo. In questo caso, essendo presente solo un nodo, al primo confronto il *next* di *current* punterà a **NULL** e quindi inserirà il nodo dopo il nodo (1).



Ora invece inseriamo un nodo (2). Creiamo un puntatore *current* che punterà inizialmente allo stesso nodo a cui punta *head*, in questo caso il nodo (1).



Al primo confronto, il *next* di *current* sarà il nodo (3), quindi il nodo (2) punterà al *next* di *current* e il nodo puntato da *current* (nodo (1)) punterà il nodo (2).



Codice del metodo Insert:

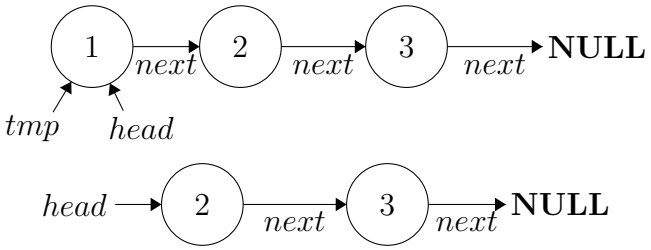
```
void Insert(int n){
    Nodo* newnode = new Nodo(n, head);
    if(head == nullptr || newnode->getData()<= head->getData()){
        newnode->setNext(head);
        head = newnode;
    } else {
        Nodo* current = head;
        //IL WHILE TUTTO IN UN'UNICA RIGA
        while(current->getNext() != nullptr &&
        newnode->getData() > current->getNext()->getData()){
            current = current->getNext();
        }
        newnode->setNext(current->getNext());
        current->setNext(newnode);
    }
    size++;
}
```

Tuttavia la **complessità** di un **inserimento ordinato** è più elevata rispetto ad un inserimento in testa o in coda, nel caso peggiore infatti la complessità è $\Theta(n)$.

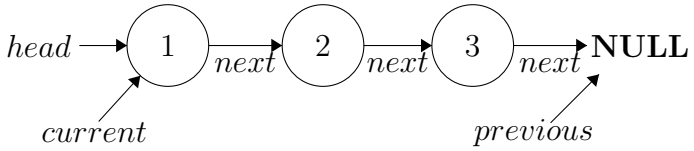
10.4.2 Remove (Cancellazione di un nodo specifico)

Come per l'inserimento, in una Lista Concatenata possiamo scegliere di eliminare il nodo in testa oppure possiamo scegliere di eliminare un nodo specifico. Chiamiamo l'operazione di cancellazione in una lista **Remove**.

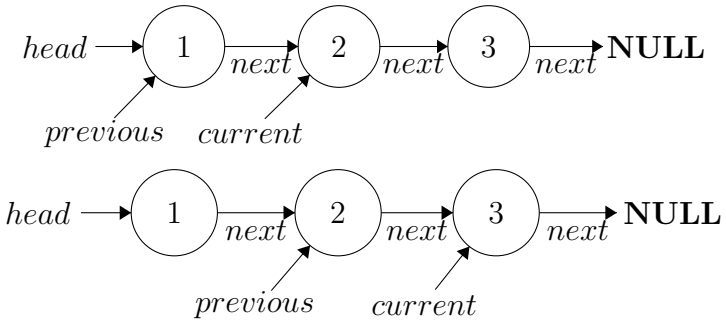
Data la lista precedente, supponiamo di voler eliminare il nodo (1).
Il metodo **Remove** prima di tutto controllerà se la lista ha nodi o meno, poi controllerà se il nodo da eliminare è quello in testa.



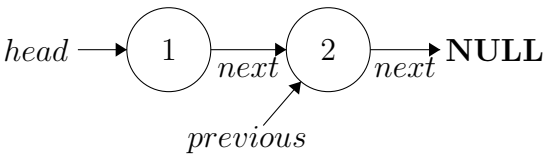
Supponiamo invece di voler eliminare il nodo (3).
Se il nodo da eliminare non si trova in testa, allora il metodo dichiarerà due puntatori a nodo, *current* (che inizialmente punterà al nodo che punta *head*) e *previous* (che inizialmente punterà a **NULL**).



I due puntatori verranno spostati fino a quando *current* non punterà al nodo da eliminare.



A quel punto avverrà la cancellazione.



Codice del metodo:

```
void remove(int n){
    if(isEmpty() == true){
        cerr << "La Lista è vuota." << endl;
        exit(EXIT_FAILURE);
    }
    if(head->getData() == n){
        Nodo* tmp = head;
        head = head->getNext();
        delete tmp;
        size--;
    }

    Nodo* current = head;
    Nodo* prev = nullptr;

    while(current->getNext() != nullptr &&
    current->getData() != n){
        prev = current;
        current = current->getNext();
    }
    if(current != nullptr){
        prev->setNext(current->getNext());
        delete current;
        size--;
    }
    else {
        cerr << "L'elemento da eliminare non è nella lista" << endl;
        exit(EXIT_FAILURE);
    }
}
```

La Cancellazione di un nodo in una Lista Concatenata nel caso migliore, ovvero quando il nodo cercato si trova in testa è **O(1)**, mentre quando il nodo si trova in mezzo alla lista, nel caso peggiore è **O(n)**.

10.5 Lista Doppiamente Concatenata

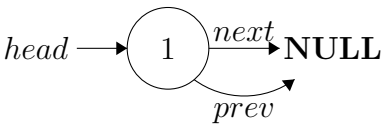
Una Lista **Doppiamente Concatenata**, a differenza della Singolarmente Concatenata, possiede due puntatori per nodo, *next* e *prev*.
Le operazioni **Insert** e **Remove** tuttavia sono molto simili, con l'unica differenza di dover gestire anche il secondo puntatore.

10.5.1 Nodo

Classe Nodo:

```
class Nodo{
    public:
        int data;
        Nodo* next;
        Nodo* prev;

    private:
        Nodo(int data, Nodo* next, Nodo* prev):
        data(data), next(next), prev(prev);
        ~Nodo() {}

        int getData(){return data};
        Nodo* getNext(){return next};
        Nodo* getPrev(){return prev};

        void setData(int data){this->data = data;}
        void setNext(Nodo* next){this->next = next;}
        void setPrev(Nodo* prev){this->prev = prev;}
}
```

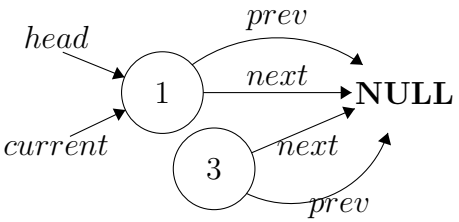
Lista con un solo Nodo:



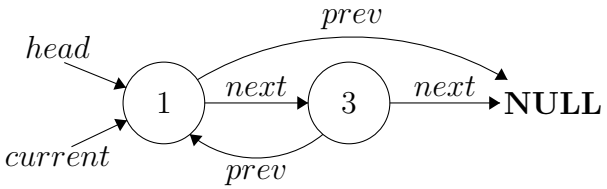
10.5.2 Insert (Inserimento Ordinato)

L'**Inserimento Ordinato** in una Lista Doppiamente Concatenata è simile a quello in una Lista Singolarmente Concatenata, l'unica differenza è la gestione di *prev*.

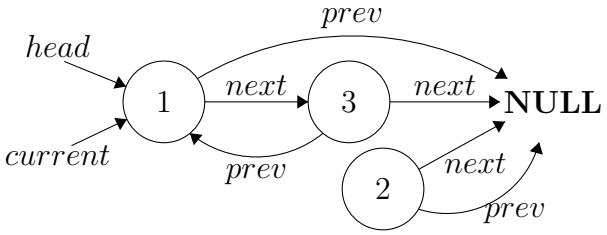
Immaginiamo di inserire nella Lista il nodo (3). Essendo $3 > 1$ allora dovrà inserito dopo il nodo (1). Allora creiamo un puntatore *current* che punterà inizialmente allo stesso nodo a cui punta *head*, in questo caso il nodo (1).



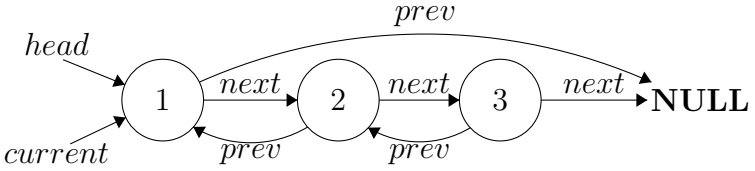
Il puntatore *current*, tramite vari confronti, determinerà la posizione giusta in cui inserire il nodo.
In questo caso, essendo presente solo un nodo, al primo confronto punterà a **NULL** e quindi inserirà il nodo dopo il nodo (1).
Il nodo (1) quindi punterà al nodo (3) con il suo puntatore *next* mentre il nodo (3) punterà al nodo (1) con il suo puntatore *prev*.



Ora invece inseriamo un nodo (2). Creiamo un puntatore *current* che punterà inizialmente allo stesso nodo a cui punta *head*, in questo caso il nodo (1).



Al primo confronto, il *next* di *current* sarà il nodo (3), quindi il nuovo nodo (2) punterà al nodo (3). e il nodo puntato da *current* (nodo (1)) punterà il nuovo nodo (2). Inoltre il *prev* del nuovo nodo (2) punterà al nodo punterà al nodo puntato da *current* e il *next* di *current* punterà al nuovo nodo (2).



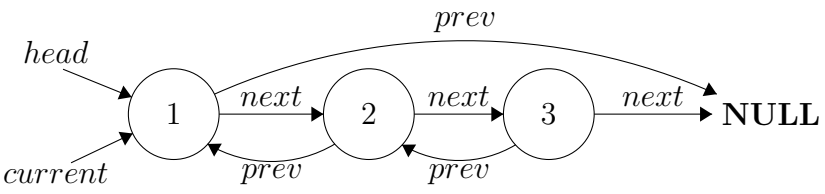
Codice del metodo Insert:

```
void Insert(int n) {
    Nodo* newnode = new Nodo(n);
    if (head == nullptr || newnode->getData() <= head->getData()){
        newnode->setNext(head);
        if (head != nullptr) {
            head->setPrev(newnode);
        }
        head = newnode;
    } else {
        Nodo* current = head;
        while (current->getNext() != nullptr &&
        newnode->getData() > current->getNext()->getData()){
            current = current->getNext();
        }
        newnode->setNext(current->getNext());
        newnode->setPrev(current);
        current->setNext(newnode);

        if (newnode->getNext() != nullptr){
            newnode->getNext()->setPrev(newnode);
        }
    }
    size++;
}
```

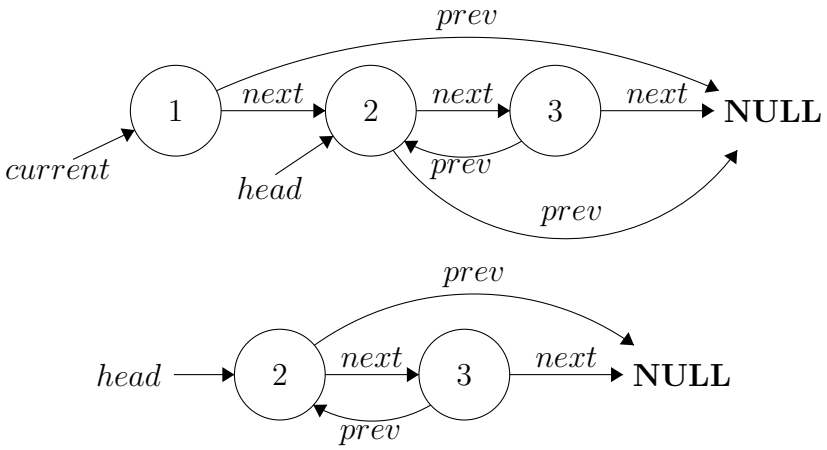
10.5.3 Remove (Cancellazione di un Nodo specifico)

La cancellazione di un nodo da una Lista Doppiamente Concatenata è simile a quella di una lista Singolarmente Concatenata, con la gestione del puntatore *prev* per ogni nodo come unica differenza.

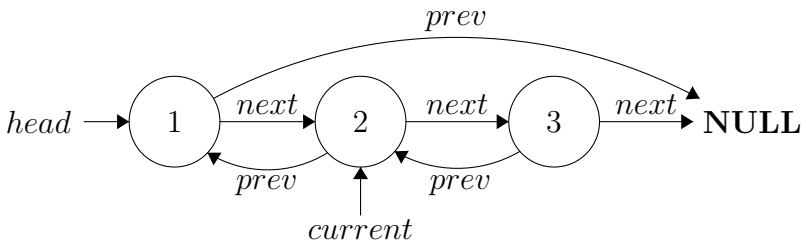
Data la lista precedente, supponiamo di voler eliminare il nodo (1).
Il metodo **Remove** prima di tutto controllerà se la lista ha nodi o meno, poi controllerà se il nodo da eliminare è quello in testa.



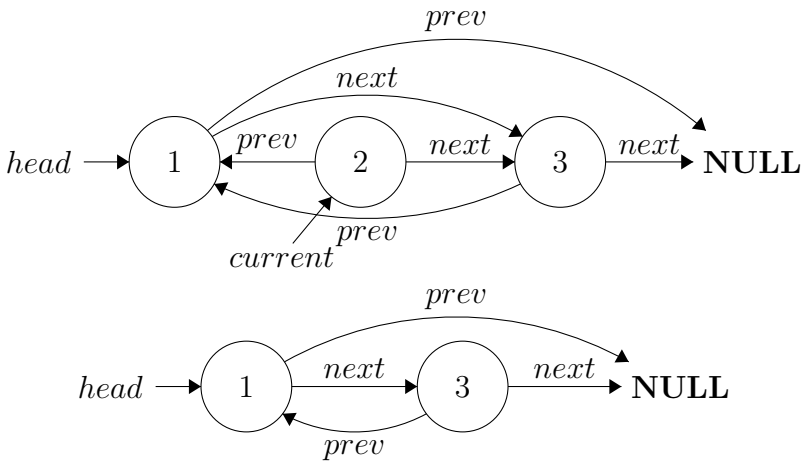
Essendo il nodo (1) in testa, allora il metodo sposterà il puntatore *head* al nodo successivo e il puntatore *prev* del nodo successivo a **NULL**.



Riutilizzando la lista iniziale, supponiamo adesso di voler eliminare il nodo (2). Il metodo dovrà scorrere il puntatore *current* fino a trovare il nodo da eliminare.



Una volta trovato sposterà i riferimenti dei due nodi adiacenti affinché si puntino tra di loro, isolando così il nodo puntato da *current*, che verrà eliminato.

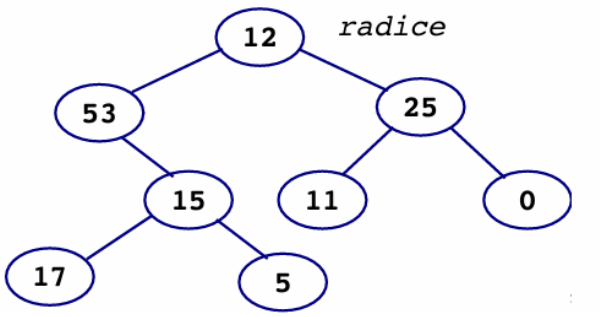


Codice del metodo Remove:

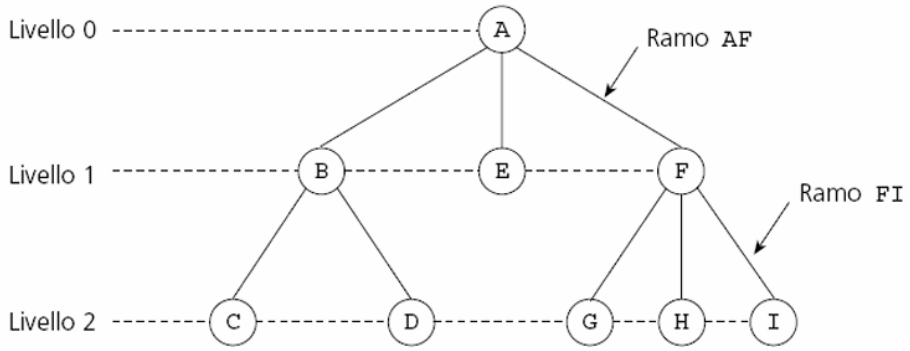
```
void remove(int n){
    if (isEmpty() == true){
        cerr << "La Lista è vuota." << endl;
        exit(EXIT_FAILURE);
    }

    Nodo* current = head;
    if (head->getData() == n){
        head = head->getNext();
        if (head != nullptr){
            head->setPrev(nullptr);
        }
        delete current;
        size--;
    }

    while (current != nullptr && current->getData() != n){
        current = current->getNext();
    }
    if (current == nullptr){
        cerr << "L'elemento non è nella lista." << endl;
        exit(EXIT_FAILURE);
    }

    // COLLEGO IL NODO PRECEDENTE CON IL SUCCESSIVO
    if (current->getPrev() != nullptr){
        current->getPrev()->setNext(current->getNext());
    }
    if (current->getNext() != nullptr){
        current->getNext()->setPrev(current->getPrev());
    }
    delete current;
    size--;
}
```

10.6 Teoria Base sugli Alberi

Prima di introdurre gli Alberi Binari di Ricerca (BST), bisogna enunciare cos'è un **albero** e quali sono le sue proprietà:
• Un **Albero** è un **Grafo** connesso e aciclico;
• Un **nodo** in un albero può avere solo un arco in entrata ma molteplici in uscita;
• Un nodo, se non ha archi in entrata, si dice **radice**;
• Un nodo, se non ha archi in uscita, si dice **foglia**;



• Il nodo da cui un arco parte si dice **padre**, un nodo a cui questo arriva si dice **figlio**;
• Due nodi con lo stesso padre si dicono **fratelli**;
• Da ogni nodo non-foglia di un albero si forma un **sottoalbero**;



Padri: A, B, F
Figli: B, E, F, C, D, G, H, I
Fratelli: {B, E, F}, {C, D}, {G, H, I}
Foglie: C, D, E, G, H, I
Nodi interni: B, F

• Dato un nodo, i nodi che appartengono al suo sottoalbero si dicono suoi **discendenti**;
• Dato un nodo, i nodi che si trovano nel cammino dalla radice ad esso sono i suoi **ascendenti** (per esempio, B ed A sono ascendenti di C);
• Il **livello** di un nodo è la sua **distanza dalla radice**;
• La radice ha livello 0, i suoi figli livello 1, i suoi nipoti livello 2 e così via;
• I fratelli hanno lo stesso livello ma non tutti i nodi dello stesso livello sono fratelli.

Profondità di un Albero:
La **profondità** di un albero è la lunghezza del cammino più lungo dalla radice ad una foglia.
Viene definita ricorsivamente:
• La radice ha profondità 0;
• La profondità di un nodo non radice è 1 + la profondità del nodo genitore.

10.6.1 Profondità di un Nodo

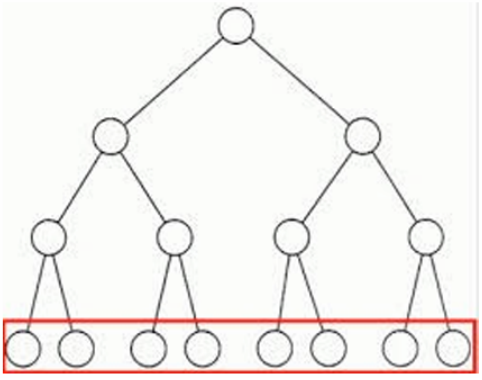
```
int Profondità(Nodo* node){
    int p = 0;
    while(node->getParent() != nullptr){
        p++;
        node = node->getParent();
    }
    return p;
}
```

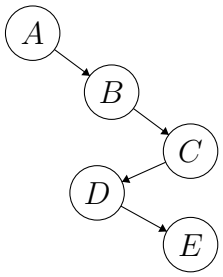
Albero Bilanciato e Albero Binario:
Un albero di profondità h si dice **bilanciato** se:
• Tutte le foglie si trovano sullo **stesso livello**;
• Dato k numero massimo di figli per nodo, ogni nodo interno (inclusa la radice) ha esattamente k figli.

Un albero si dice **binario** se ogni nodo (compresa la radice) non ha più di due figli, ovvero il **sinistro** e il **destro**. Proprietà di un Albero Binario:
• Ad ogni livello n di un albero binario può contenere al più 2^n nodi;
• Il numero totale di nodi di un albero (incluse le foglie) di profondità n è al massimo $2^{(n+1)} - 1$;
• La profondità di un albero bilanciato è $\mathbf{O}(logn)$.

Osservazione: Un albero binario non vuoto si dice **degenere** se ogni nodo diverso da una foglia ha un solo figlio (per convenzione, si assume che l'albero vuoto sia degenere).

Esempi in figura: Albero Binario Bilanciato e Albero Binario **Degenere**.



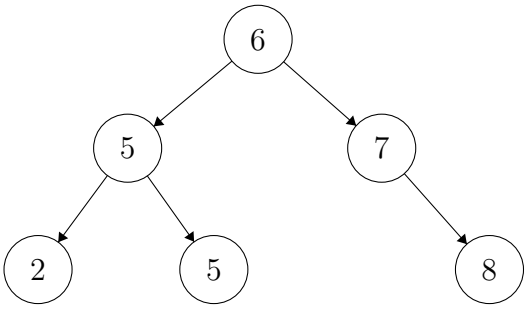


10.7 Albero Binario di Ricerca (BST)

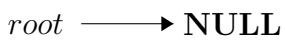
Un Albero Binario di Ricerca, o Binary Search Tree (BST), è una struttura dati **ricorsiva** ad Albero Binario.
Un Albero Binario si dice **BST** se rispetta una proprietà fondamentale:
Dato un nodo generico x:
• se y è un nodo nel sotto-albero **sinistro** di **radice** x, allora
$y.valore < x.valore$;
• se y è un nodo nel sotto-albero **destro** di **radice** x, allora
$y.valore > x.valore$.

Un **BST** utilizza un puntatore **Root** per mantenere il riferimento alla radice dell'albero. Essendo una struttura ricorsiva, per ogni sotto-albero ci sarà un puntatore *root*. Inoltre si utilizza un intero **size** per contare il numero di nodi dell'Albero.

Esempio in figura: Esempio di un Albero Binario di Ricerca.



Albero Vuoto:

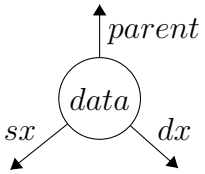


Classe BST:

```
class BST{
    private:
        Nodo* root;
        int size;
    public:
        BST() : root(nullptr), size(0) {}
        ~BST() {}

        Nodo* getRoot(){return root;}
}
```

10.7.1 Nodo

Il nodo di un Albero Binario di Ricerca, oltre al dato, utilizza **tre** puntatori:
• Puntatore al **figlio destro** (che chiameremo *dx*);
• Puntatore al **figlio sinistro** (che chiameremo *sx*);
• Puntatore al **padre** (che chiameremo *parent*).



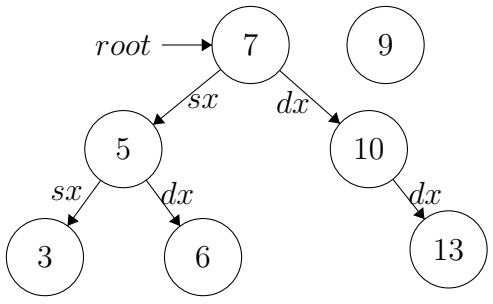
Classe Nodo:

```
class Nodo{
    private:
        int data;
        Nodo* parent;
        Nodo* dx;
        Nodo* sx;
    public:
        Nodo(int data, Nodo* parent, Nodo* dx, Nodo* sx){
            this->data = data;
            this->parent = parent;
            this->dx = dx;
            this->sx = sx;
        }
        ~Nodo() {}

        int getData(){return data;}
        Nodo* getParent(){return parent;}
        Nodo* getDx(){return dx;}
        Nodo* getSx(){return sx;}

        void setData(int data){this->data = data;}
        void setParent(Nodo* parent){this->parent = parent;}
        void setDx(Nodo* dx){this->dx = dx;}
        void setSx(Nodo* sx){this->sx = sx;}
};
```

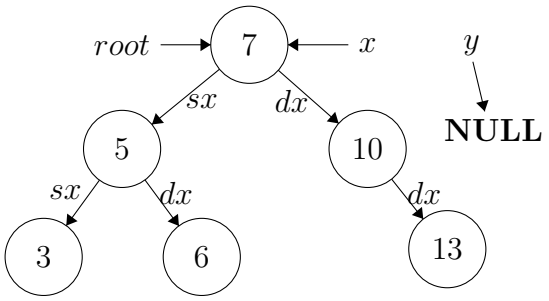
10.7.2 Inserimento (Tree-Insert)

La procedura **Tree-Insert** per l'Inserimento di nodi in un Albero la sopracitata proprietà fondamentale dei BST.
Il Metodo dichiara due puntatori a nodo x e y. Il puntatore x traccerà il cammino per trovare il punto dove allocare il nuovo nodo, mentre il puntatore y inseguirà x, puntando sempre al **padre** di x.

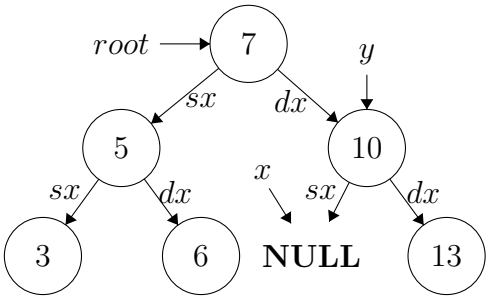
Dato il **BST** in figura, supponiamo di aggiungere il nodo (9).



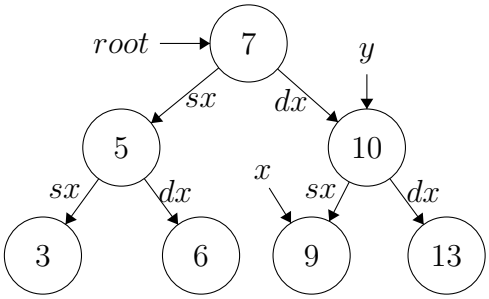
Dichiariamo i due puntatori x, che punterà al nodo puntato da *root*, e y che inizialmente punterà a **NULL**.



I puntatori percorreranno l'albero fino a trovare la giusta posizione per il nodo (9). $9 > 7$ quindi si procede andando a dx, x punterà a al nodo (10) mentre y al nodo (7). Poi $9 < 10$ quindi si andrà a sx. Qui x punterà a **NULL** mentre y al nodo (10).



Qui verrà inserito il nodo (9), che sarà puntato da sx del nodo (10), che a sua volta sarà puntato da *parent* del nodo (9).



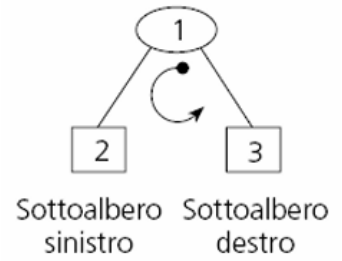
Codice del metodo Tree-Insert:

```
TreeInsert(int n){
    Nodo* x = root;
    Nodo* y = nullptr;

    while(x != nullptr){
        y = x;
        if(n < x->getData()){
            x = x->getSx();
        } else {
            x = x->getDx();
        }
    }

    Nodo* newnode = new Nodo(n, y, nullptr, nullptr);
    if(y == nullptr){
        root = newnode;
    } else if(newnode->getData() < y->getData()){
        y->setSx(newnode);
    } else {
        y->setDx(newnode);
    }
}
```

La complessità della procedura **Tree-Insert** è $\mathbf{O}(h)$, con h l'altezza dell'albero.

10.7.3 Visita Pre-Order (Pre-Ordine)

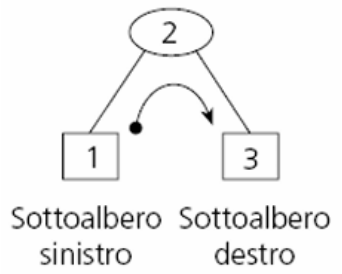
La procedura **Pre-Order** visita prima la **radice**, poi il **sottoalbero sinistro** e infine il **sottoalbero destro**. Complessità: $\Theta(n)$.




```
PreOrder(Nodo* p){
    if(p){
        cout << p->getData() << " "; // visita la radice
        preOrder(p->getSx());  // visita il sottoalbero sinistro
        preOrder(p->getDx());  // visita il sottoalbero destro
    }
}
```

10.7.4 Visita In-Order (In-Ordine)

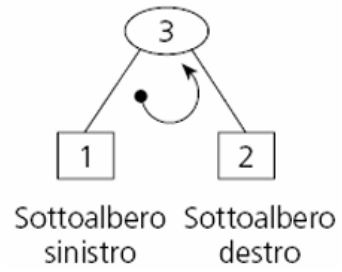
La procedura **In-Order** visita prima il **sottoalbero sinistro**, poi la **radice** e infine il **sottoalbero destro**. Complessità: $\Theta(n)$.




```
InOrder(Nodo *p){
    if(p){
        inOrder(p->getSx()); // visita il sottoalbero sinistro
        cout << p->getData() << " ";  // visita la radice
        inOrder(p->getDx()); // visita il sottoalbero destro
    }
}
```

10.7.5 Visita Post-Order (Post-Ordine)

La procedura **Post-Order** visita prima il **sottoalbero sinistro**. poi il **sottoalbero destro** e infine la **radice**. Complessità: $\Theta(n)$.

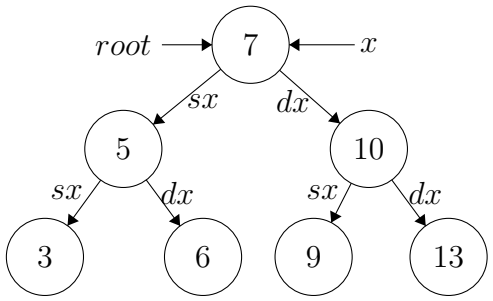



```
PostOrder(Nodo *p){
    if (p){
        postOrder(p ->getSx()); // visita il sottoalbero sinistro
        postOrder(p ->getDx()); // visita il sottoalbero destro
        cout << p ->getData() << " "; // visita la radice
    }
}
```

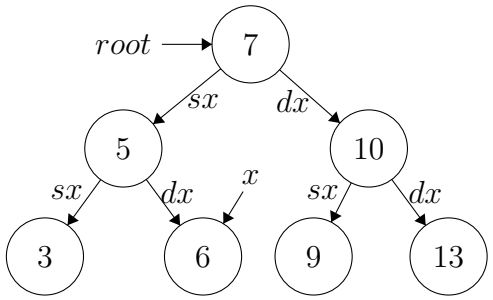
10.7.6 Tree-Search (Iterativa e Ricorsiva)

La procedura **Tree-Search** dichiara un puntatore x che scorre l'albero fino a trovare il nodo cercato e ritorna il puntatore. Si può implementare sia in modo **ricorsivo** che **iterativo**.

Dato il **BST** in figura, supponiamo di cercare il nodo (6).
Il puntatore x inizialmente punterà allo stesso nodo a cui punta *root*.



Essendo $6 < 7$, x scenderà a sx, puntando il nodo (5). Successivamente avremo $6 > 5$, quindi x scenderà a dx e troverà il nodo (6).



Codice della TreeSearch Iterativo:

```
Nodo* IterativeTreeSearch(int n){
    Nodo* x = root;
    while(x != nullptr || n != x->getData()){
        if(n < x->getData()){
            x = x->getSx();
        } else {
            x = x->getDx();
        }
    }
    return x;
}
```

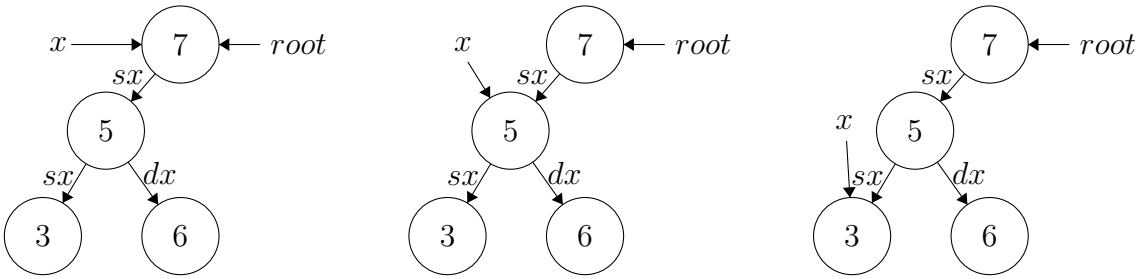
Codice della TreeSearch Ricorsivo:

```
Nodo* TreeSearch(Nodo* x, int n){
    if(root == nullptr || n == root->getData()){
        return root;
    } else if(n < root->getData()){
        x = root;
        TreeSearch(x->getSx(), n);
    } else {
        x = root;
        TreeSearch(x->getDx(), n);
    }
}
```

Complessità: $\mathbf{O}(h)$, con h l'altezza dell'albero.

10.7.7 Ricerca del Minimo e del Massimo

Tree-Minimum:
La procedura **Tree-Minimum** trova l'elemento **minimo** dell'albero scorrendo un puntatore x che, partendo da *root*, seguirà ogni puntatore sx dei figli a sinistra, fino a quando non viene trovato **NULL**.
L'ultimo elemento prima di **NULL** sarà il minimo del BST.



Codice del metodo TreeMinimum:

```
Nodo* TreeMinimum(Nodo* x){
    x = root;
    while(x->getSx() != nullptr){
        x = x->getSx();
    }
    return x;
}
```

Tree-Maximum:
La procedura **Tree-Minimum** trova l'elemento **massimo** dell'albero scorrendo un puntatore x che, partendo da *root*, seguirà ogni puntatore dx dei figli a destra, fino a quando non viene trovato **NULL**.
L'ultimo elemento prima di **NULL** sarà il massimo del BST. Non verrà rappresentata in quanto il procedimento è analogo a quello della **Tree-Minimum**.

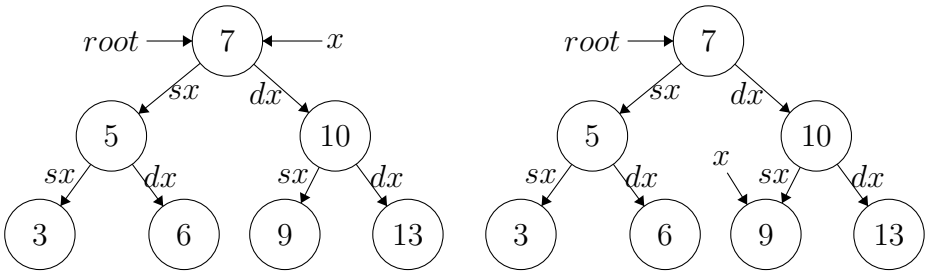
Codice del metodo TreeMaximum:

```
Nodo* TreeMaximum(Nodo* x){
    x = root;
    while(x->getDx() != nullptr){
        x = x->getDx();
    }
    return x;
}
```

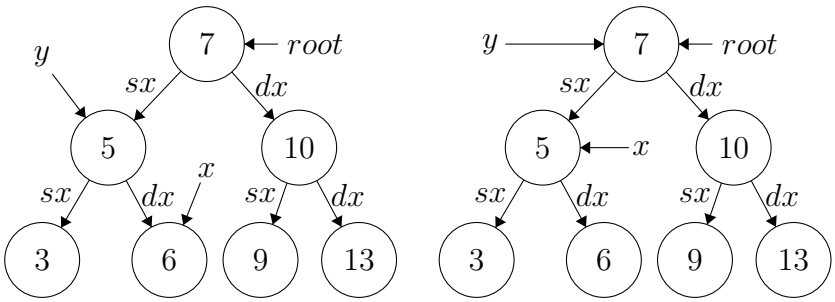
10.7.8 Successore e Predecessore di un Nodo

Tree-Successor:
Dato un nodo x in un BST, la procedura **Tree-Successor** restituisce la **posizione** del successore di x, ovvero il nodo con la più piccola chiave maggiore della chiave di x.
La procedura prevede tre casi:
• Se il sottoalbero **destro** del nodo x non è vuoto, allora il successore di x è proprio il nodo più a **sinistra** del sottoalbero destro;
• Se il sottoalbero **destro** del nodo x è vuoto, allora il successore y (se esiste) è l'antenato più prossimo di x il cui figlio **sinistro** è anche antenato di x.
• Se il successore del nodo non esiste, ovvero il nodo x è il **massimo** del BST, allora la procedura restituisce **NULL**. Complessità: $\mathbf{O}(h)$.

Dato il BST in figura supponiamo di voler trovare il successore del nodo (7). Siamo nel **primo** caso, il sottoalbero del nodo (10) non è vuoto. Questo caso è banale, in quanto la procedura chiamerà la procedura **Tree-Minimum** e l'elemento restituito sarà il nodo (9).



Supponiamo di voler cercare il successore del nodo (6). Siamo nel **secondo** caso, in quanto il nodo (6) è una foglia. Dichiariamo un puntatore y che punterà al padre di x e risaliamo l'albero da x fino a quando incontriamo un nodo che è il figlio **sinistro** di suo padre.



Il successore del nodo (6) è il nodo (7).

Codice della Tree-Successor:

```
Nodo* TreeSuccessor(Nodo* x){
    if(x->getDx() != nullptr){
        return TreeMinimum(x->getDx());
    }

    Nodo* y = x->getParent();
    while(y != nullptr && x == y->getDx()){
        x = y;
        y = y->getParent();
    }
    return y;
}
```

Tree-Predecessor:
Dato un nodo x in un BST, la procedura **Tree-Predecessor** restituisce la **posizione** del predecessore di x, ovvero il nodo con la più piccola chiave minore della chiave di x. La procedura prevede tre casi:
• Se il sottoalbero **sinistro** del nodo x non è vuoto, allora il predecessore di x è proprio il nodo più a **destra** del sottoalbero sinistro;
• Se il sottoalbero **sinistro** del nodo x è vuoto, allora il predecessore y è l'antenato più prossimo di x il cui figlio **destro** è anche antenato di x.
• Se il predecessore del nodo non esiste, ovvero il nodo x è il **minimo** del BST, allora la procedura restituisce **NULL**. Complessità: $\mathbf{O}(h)$.

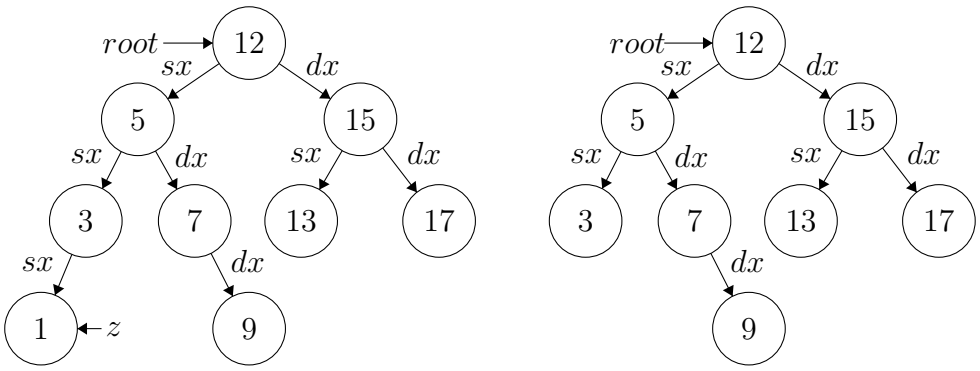
Codice della Tree-Predecessor:

```
Nodo* TreePredecessor(Nodo* x){
    if(x->getSx() != nullptr){
        return TreeMaximum(x->getSx());
    }

    Nodo* y = x->getParent();
    while(y != nullptr && x == y->getSx()){
        x = y;
        y = y->getParent();
    }
    return y;
}
```

10.7.9 Cancellazione (Trapianta e Tree-Delete)

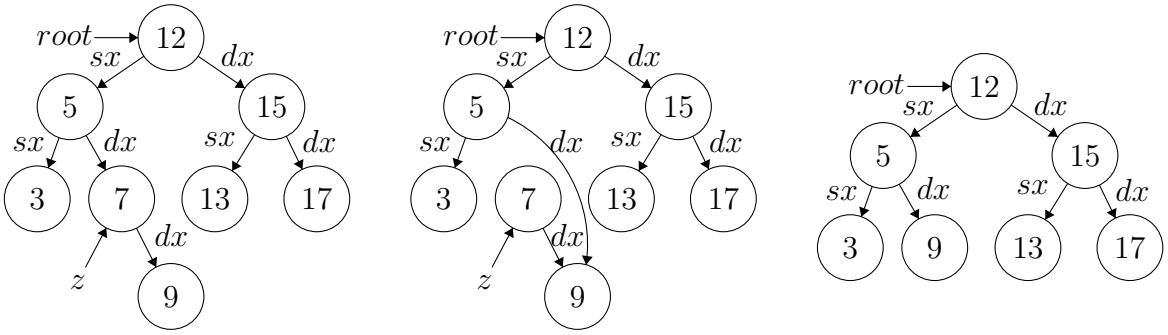
La cancellazione di un nodo z in un BST considera **tre casi** base:
• Se il nodo z non ha figli, cambieremo il riferimento del **padre** di z in modo che punti a **NULL**;

Supponiamo di eliminare il nodo (1). Essendo una foglia sx, il nodo padre punterà a **NULL** e il nodo potrà essere eliminato.



• Se il nodo z ha un solo figlio, eleviamo il figlio per prendere il posto di z e cambiamo il riferimento del **padre** affinché punti il figlio di z invece che z;

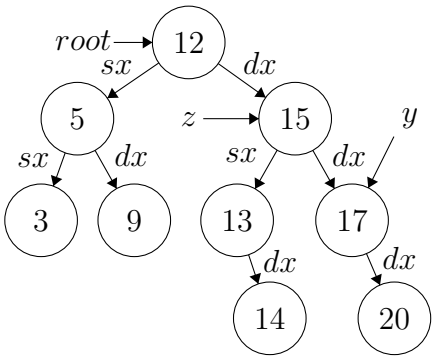
Supponiamo di eliminare il nodo (7). Esso possiede un figlio dx, il nodo (9). Il padre del nodo (7), il nodo (5), dovrà puntare il nodo (9) invece del figlio. Fatto questo, il nodo (9) potrà essere eliminato.



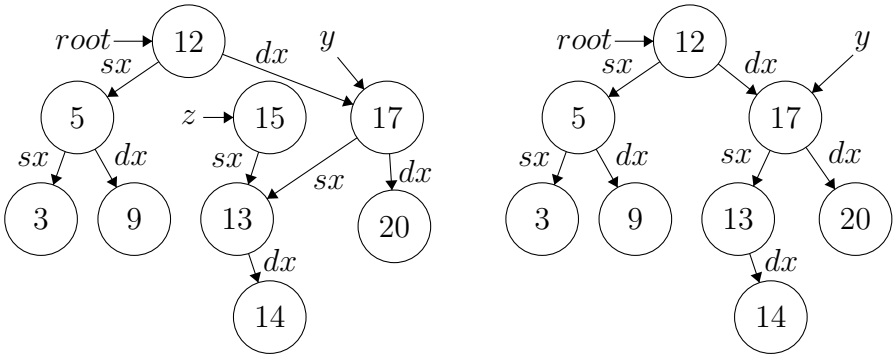
• Se il nodo z ha due figli, troviamo il **successore** (y) di z, che dovrebbe trovarsi nel sottoalbero **destro** di z, e facciamo in modo che y assuma la posizione di z nell'albero. La parte restante del sottoalbero **destro** originale diventa il nuovo sottoalbero **destro** di y e il sottoalbero **sinistro** di z diventa il nuovo sottoalbero **sinistro** di y.

Abbiamo **due** ulteriori sottocasi, in quanto bisogna considerare il caso in cui y è figlio **destro** di z:
• se y è figlio destro di z, il nodo y sarà alzato al livello del nodo da eliminare e sarà puntato dal padre del nodo di z e inoltre y punterà il figlio sx di z;
• se y si trova nel sottoalbero **destro** di z ma non è il suo figlio destro, una volta trovato il successore y lo sostituiamo con z. Se y non ha figli, ci si riconduce al **primo caso**, altrimenti se ha un figlio dx ci si riconduce al **secondo caso**.

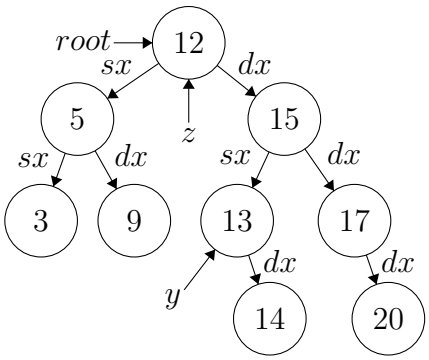
Esaminiamo il **primo sottocaso**. Supponiamo di voler rimuovere il nodo (15). Il suo successore y è il figlio dx di z, il nodo (17).



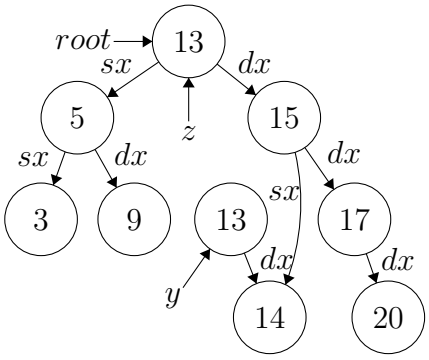
Alzo y alla stessa altezza di z. Il padre di z punterà ad y e quest'ultimo punterà al figlio sx di z (che a sua volta punterà a y invece che a z). Il nodo z, rimasto solo, verrà eliminato.

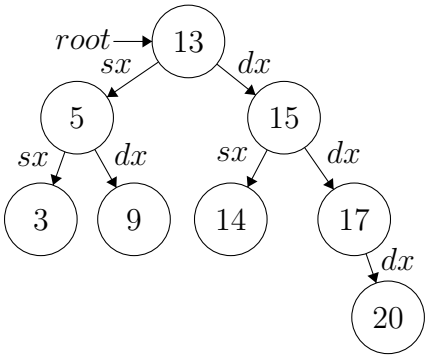


Esaminiamo adesso il **secondo sottocaso**. Supponiamo di voler eliminare il nodo (12). Il suo successore y, si troverà nel sottoalbero **destro** di z, ma non è il suo figlio dx. Allora sarà il figlio più a sx del sottoalbero **destro** di z, ovvero il nodo (13).



Sostituiamo y con il nodo z.
Avendo y un figlio dx, esso diventerà il figlio sx del nodo (15), rifacendomi quindi al **secondo caso**, dove un nodo ha un solo figlio. Infatti se y non avesse avuto figli, ci saremmmo rifatti al **primo caso**.
Infine elimino il nodo y.





Per eliminare nodi da un BST viene usata, oltre alla procedura **Tree-Delete**, anche una procedura ausiliaria chiamata **Transplant**, che permette di
spostare sottoalberi in un albero binario di ricerca.
Quando **Transplant** sostituisce il sottoalbero con radice nel nodo u con il sottoalbero con radice nel nodo v, il padre del nodo u diventa il padre del nodo v, e il padre di u alla fine ha v come figlio.

Codice della procedura Transplant:

```
Transplant(Nodo* u, Nodo* v){
    if(u->getParent() == nullptr){
        root = v;
    } else if(u == u->getParent()->getSx()){
        u->getParent()->setSx(v);
    } else {
        u->getParent()->setDx(v);
    }
    if(v != nullptr){
        v->setParent(u->getParent());
    }
}
```

Codice della procedura Tree-Delete:

```
TreeDelete(Nodo* z){
    if(z->getSx() == nullptr){
        Transplant(z, z->getDx());
    } else if(z->getDx() == nullptr){
        Transplant(z, z->getSx());
    } else {
        Nodo* y = TreeMinimum(z->getDx());
        if(y->getParent() != z){
            Transplant(y, y->getDx());
            y->setDx(z->getDx());
            y->getDx()->setParent(y);
        }
        Transplant(z, y);
        y->setSx(z->getSx());
        y->getDx()->setParent(y);
    }
}
```

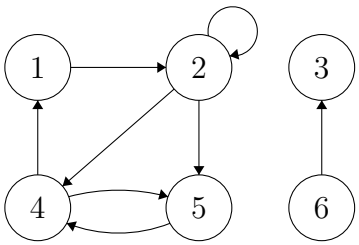
La procedura **Transplant** richiede tempo **costante**, quindi ha complessità $\mathbf{O(1)}$. La procedura **Tree-Delete** ha invece complessità $\mathbf{O}(h)$ in quanto presenta la presenta la procedura **Tree-Minimum**.

11 Grafi

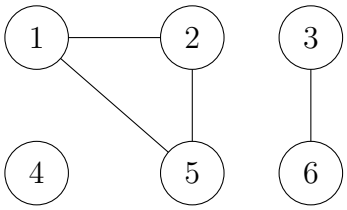
11.1 Teoria Base sui Grafi

Si dice **Grafo** un insieme di nodi legati "a due a due" da **archi**, che possono essere orientati o meno.
Un Grafo si indica con $G = (V, E)$, con V l'insieme dei **nodi** ed E l'insieme degli **archi** (u, v).

Grafi Orientati:
Dato un grafo orientato $G = (V, E)$:
• L'arco (u, v) è **uscente** da u ed **entrante** in v;
• Se l'arco (u, v) è in E, v è **adiacente** a u.
• Un nodo può essere collegato a sé stesso.



Grafi Non Orientati:
Dato un grafo non orientato $G = (V, E)$:
• Dato un arco (u, v) di E, u e v sono **incidenti**, ovvero sia entranti che uscenti;
• La relazione di **adiacenza** è **simmetrica**;
• Un nodo **non** può essere collegato a sé stesso.



Grado di un Nodo:
• Se $G = (V, E)$ è orientato, il grado di un nodo sarà il numero di archi entranti sommato al numero di archi uscenti;
• Se $G = (V, E)$ non è orientato, il grado di un nodo sarà il numero di archi incidenti.

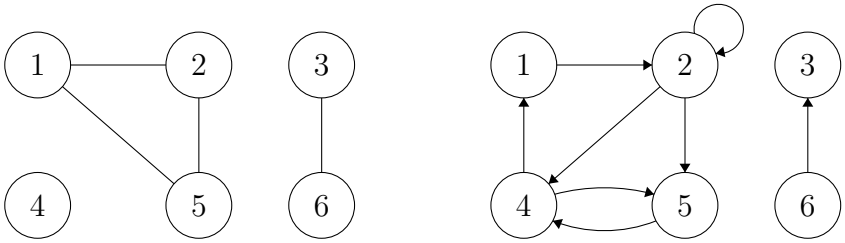
Definizione di Cammino, Ciclo e Proprietà varie:
Un cammino (di lunghezza k) da u a v è una sequenza di nodi da v_0 a v_k tale che $u = v_0$ e $v = v_k$:
• Il cammino contiene i vertici $v_o, \cdots, v_k$ e gli archi $(v_0, v_1), \cdots, (v_{k-1}, v_k)$;
• Un nodo v è raggiungibile da u se esiste un cammino da u a v;
• Un cammino si dice **semplice** se tutti i vertici in esso contenuti sono distinti (alternativamente si chiama anche **percorso**);
• Un **sottocammino** è una sequenza di vertici di un cammino, ad esempio: $v_i, \cdots, v_j$ per $0 \leq i \leq j \leq k$.

Un **Ciclo** è un cammino $v_0, \cdots, v_k$ dove $v_0 = v_k$:
• Un ciclo si dice **semplice** se tutti i suoi nodi sono distinti (alternativamente si chiama **Circuito**);
• Un grafo senza cicli si dice **aciclico**.

Connessione tra Grafi:
Un Grafo non orientato si dice **connesso** se ogni coppia di vertici è unita da un cammino.
• Una **Componente Connessa** è una classe di equivalenza determinata dalla relazione "è raggiungibile da".
• Un grafo non orientato è connesso se ha 1 sola componente connessa.

Un Grafo orientato si dice **fortemente connesso** se per ogni coppia di nodi (u, v) esiste un cammino che unisce u a v e v a u.
• Una componente **fortemente connessa** è una classe di equivalenza determinata dalla relazione "sono mutualmente raggiungibili".

Esempi in figura: Grafo non Orientato e Grafo Orientato.



Il grafo non orientato possiede 3 componenti connesse: {1,2,5}, {3,6}, {4}.
Il grafo orientato possiede 3 componenti fortemente connesse: {1,2,4,5}, {3}, {6}.

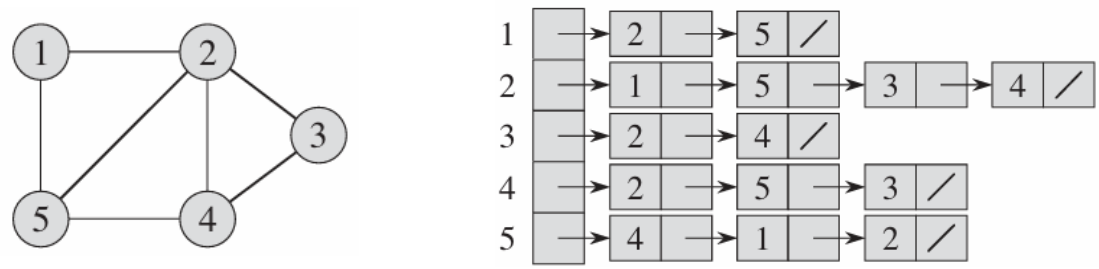
Sottografo:
Un grafo $G' = (V', E')$ si dice **sottografo** di $G = (V, E)$ se V' sottoinsieme di V ed E' sottoinsieme di E.
Grafo Completo:
Un grafo $G = (V, E)$ non orientato si dice **completo** se ogni coppia di vertici è adiacente.
Grafo Sparso:
Un grafo $G = (V, E)$ è detto **sparso** se contiene pochi archi, ovvero
se $|E| < |V|^2$.
Grafo Denso:
Un grafo $G = (V, E)$ è detto **denso** se $|E|$ è prossimo a $|V|^2$.

Rappresentazione di un Grafo:
Un Grafo può essere rappresentato principalmente in due modi:
• Liste di Adiacenza;
• Matrici di Adiacenza;

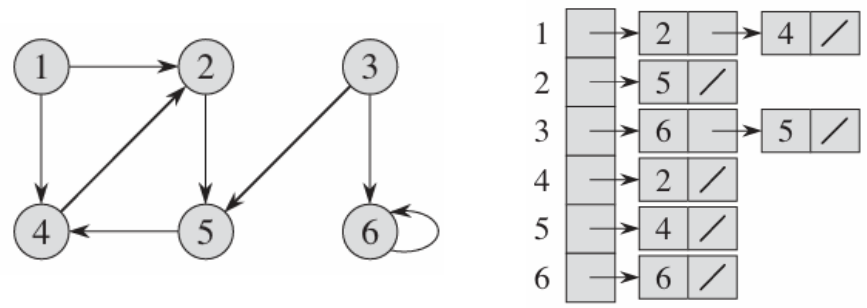
11.2 Liste di Adiacenza

Un grafo $G = (V, E)$ rappresentato con Liste di Adiacenza consiste in un array Adj di $|V|$ liste, una per ogni vertice di V.
Per ogni $u \in V$, la Lista di Adiacenza contiene tutti i vertici v tali che esista un arco $(u, v) \in E$, ovvero $Adj[u]$ include tutti i vertici adiacenti a u in G (oppure i puntatori a questi vertici).
Questa rappresentazione è la più usata, specialmente per i grafi **sparsi**.
Complessità Spaziale: $\mathbf{O}(|V| + |E|)$.

La somma delle lunghezze di tutte le liste, in un grafo **non orientato** è $2|E|$, perché se (u, v) è un arco non orientato, allora u appare nella lista di adiacenza di v e viceversa.



La somma delle lunghezze di tutte le liste, in un grafo **orientato** è $|E|$, perché se (u, v) è un arco orientato, esso è rappresentato inserendo v in $Adj[u]$.



Implementazione di un Grafo con Lista di Adiacenza:

```
class Grafo{
    private:
        Lista* Adj; //array di liste;
        int n; //numero di nodi del grafo
        enum coloriGrafo{white, gray, black};
        coloriGrafo* colArr; //array di colori
        int* dist; //array con le distanze tra i nodi
        int* fine; //solo DFS, tempi di fine dell'algoritmo
        int* pred; //predecessore del nodo visitato

    public:
        Grafo(int n) : n(n) {
            Adj = new Lista[n];
            colArr = new coloriGrafo[n];
            dist = new int[n];
            fine = new int[n];
            pred = new int[n];
        }
        ~Grafo(){
            delete[] Adj;
            delete[] dist;
            delete[] fine;
            delete[] pred;
        }

        int getN() const {return n;}
        Lista& getAdj(int v) const {return Adj[v];}

        Graph& AddEdge(int u, int v){
            Adj[u].InsertTail(v);
            Adj[v].InsertTail(u);
            return *this;
        }
}
```

Il metodo **AddEdge** permette di creare un arco non orientato tra due nodi (dati in input) e restituisce il riferimento al grafo stesso.

Per stampare il grafo si può implementare un metodo come quello riportato, che stampa ogni lista dell'array (quindi è necessario creare un metodo di stampa nella lista) oppure si può utilizzare l'overloading sull'operatore «.

```
void printGrafo(){
    for(int i = 0; i < n; i++){
        cout << "Nodo " << i << " connesso a: ";
        Adj[i].printList();
        cout << endl;
    }
}
```

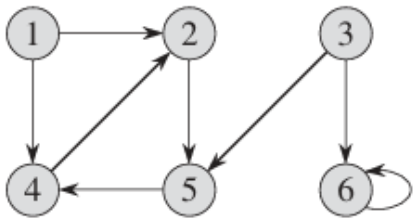
Le liste di adiacenza possono essere facilmente modificate per rappresentare i **grafi pesati**, ovvero i grafi per i quali ogni arco ha un peso associato, tramite una **funzione peso** $w : E \to R$.

11.3 Matrici di Adiacenza

Un grafo $G = (V, E)$ rappresentato con Matrici di Adiacenza consiste in una matrice $A = (a_{ij})$ di dimensioni $|V| \times |V|$, con ogni vertice del grafo numerato $(1, 2, \cdots, |V|)$ arbitrariamente, tale che:

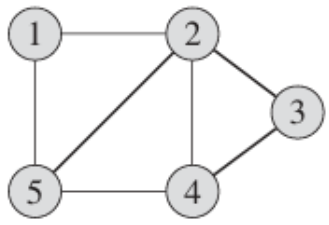
$$a_{ij} = \begin{cases} 1 & se\ (i,j) \in E \\ 0 & negli\ altri\ casi \end{cases}$$

Esempio n.1: Grafo Orientato con la sua Matrice di Adiacenza.



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Esempio n.2: Grafo non Orientato con la sua Matrice di Adiacenza.



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Complessità Spaziale: $\mathbf{O}(|V|^2)$.

Implementazione di un Grafo con Matrice di Adiacenza:

```
class GrafoMatrice{
    private:
        int** Adj;
        int n;
        enum coloriGrafo{white, gray, black};
        coloriGrafo* colArr; // array di colori
        int* dist;
        int* pred;

    public:
        GrafoMatrice(int n) : n(n){
            Adj = new int*[n];
            for(int i = 0; i < n; i++){
                Adj[i] = new int[n];
                for(int j = 0; j < n; j++){
                    Adj[i][j] = 0;
                }
            }
        }

        ~GrafoMatrice(){
            for(int i = 0; i < n; i++){
                delete[] Adj[i];
            }
            delete[] Adj;
        }

        int getN() const {return n;}
        int getAdj(int u, int v) const {return Adj[u][v];}

        GrafoMatrice& AddEdge(int u, int v){
            Adj[u][v] = 1;
            Adj[v][u] = 1;
            return *this;
        }
};
```

Il metodo **AddEdge** prende come argomenti due nodi del grafo e modifica il valore ad 1 nella matrice se i due vertici sono collegati. Esso restituirà il riferimento al grafo stesso.

Per stampare il grafo si può implementare un metodo come quello riportato qui sotto oppure si può utilizzare l'overloading sull'operatore «.

```
void printGrafo(){
    for(int i = 0; i < n; i++){
        cout << endl;
        for(int j = 0; j < n; j++){
            cout << Adj[i][j] << " ";
        }
    }
    cout << endl;
}
```

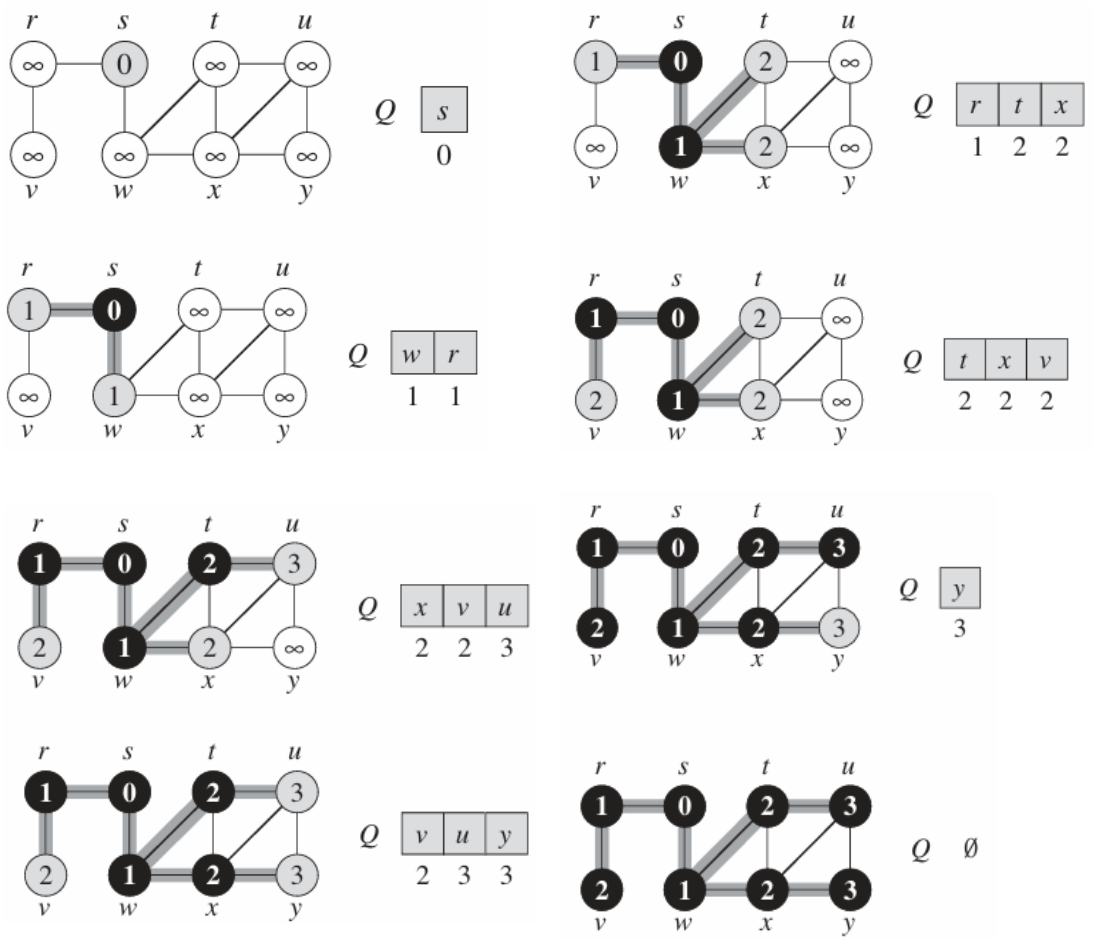
Osservazioni:
• Se il grafo G non è orientato, la sua matrice di adiacenza A corrisponderà alla sua matrice Trasposta A^T;
• Si può modificare la matrice di adiacenza per rappresentare i grafi pesati, seppur non sempre conviente data l'elevata complessità spaziale;
• Conviene usare la rappresentazione con Matrici di Adiacenza se il grafo è **denso**;

11.4 Visita in Ampiezza (BFS)

La **Visita in Ampiezza** o Breadth First Search è un algoritmo di ricerca nei Grafi. Dato un grafo $G = (V, E)$ (orientato o meno) e un vertice s detto **sorgente**, l'algoritmo visita sistematicamente tutti gli archi di G per scoprire tutti i vertici che sono raggiungibili da s. L'algoritmo inoltre:
• Calcola la distanza (ovvero il numero minimo di archi) da s a ciascun vertice raggiungibile;
• Crea un albero BF con radice s che contiene tutti i vertici raggiungibili, dove per ogni vertice v raggiungibile da s, il cammino semplice nell'albero BF che va da s a v corrisponde ad un "cammino minimo" da s a v in G.

La visita in ampiezza si chiama così poiché espande la frontiera fra i vertici scoperti a distanza k da s e quelli da scoprire a distanza $k + 1$.

Per tenere traccia del lavoro svolto, l'algoritmo **colora i nodi** del grafo di bianco, grigio o nero. Inizialmente tutti i nodi sono **bianchi**, ma verranno colorati di **grigio** o **nero** se vengono visitati. Proprietà:
• Se un nodo u è **nero**, i nodi adiacenti ad esso potranno essere solo neri o grigi.
• Se un nodo u è **grigio**, potrebbe avere accanto anche nodi bianchi, ovvero nodi non ancora scoperti.



L'algoritmo **BFS** presuppone che il grado G dato in input sia rappresentato tramite liste di adiacenza.
L'algoritmo utilizza inoltre una coda Q per gestire l'insieme dei nodi grigi, effettuando una **Enqueue** quando scopre un nuovo nodo e una **dequeue** quando il nodo diventa nero.

Complessità della procedura: $\mathbf{O}(V + E)$, in quanto le visite sulle liste di adiacenza impiegano $\mathbf{O}(E)$ mentre le operazioni totali sulla coda impiegano $\mathbf{O}(V)$.

Codice dell'algoritmo (con Liste di Adiacenza):

```
BFS(int s){
    //coloro ogni nodo del grafo di bianco
    for(int i = 0; i < n; i++){
        colArr[i] = white;
        dist[i] = INT_MAX;
        pred[i] = -1;
    }

    //coloro la sorgente del grafo
    colArr[s] = gray;
    dist[s] = 0;
    Coda Queue; // creo la coda dove verranno inseriti i nodi grigi
    Queue.EnQueue(s); // inserisco la sorgente nella coda

    while(Queue.isEmpty() == false){
        //estraggo il nodo in testa memorizzando il suo dato
        int u = Queue.DeQueueHead();
        // e creo un nodo p per visionare i nodi con cui è connesso p
        const Nodo* p = Adj[u].getHead();

        while(p != nullptr){
            int v = p->getData(); //copio il dato di p in v
            if(colArr[v] == white){  // se il colore di v è bianco
                colArr[v] = gray; // allora lo coloro di grigio
                // aumento la distanza tra il nodo tra v e u di 1
                dist[v] = dist[u] + 1;
                pred[v] = u; // imposto il predecessore
                // inserisco il nodo grigio nella coda
                Queue.EnQueue(v);
            }
            p = p->getNext(); // scorro la lista
        }
        // coloro il nodo di nero, quindi il nodo è visitato
        colArr[u] = black;
    }
}
```

11.5 Print Path

La procedura **Print Path** stampa i vertici di un **cammino minimo** da s a v, supponendo che sia già stata eseguita la **BFS** o la **DFS** per calcolare l'albero dei cammini minimi.

Codice dell'algoritmo:

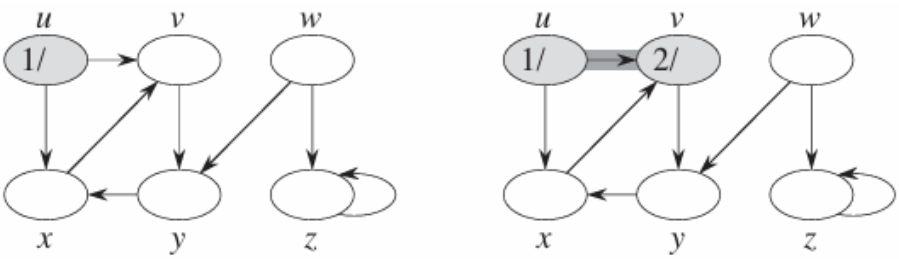
```
void printPath(int s, int v){
    if(v == s){
        cout << s << " ";
    } else if(pred[v] == -1){
        cout << "Non c'è alcun cammino tra " << s << " a " << v;
    } else {
        printPath(s, pred[v]);
        cout << v << " ";
    }
    cout << endl;
}
```

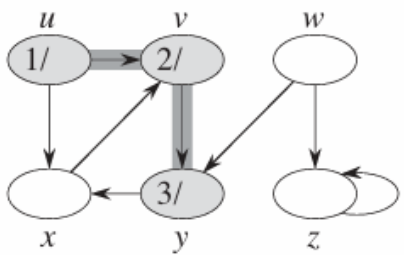
Complessità: **O(|V|)**, in quanto ogni chiamata ricorsiva riguarda un cammino che ha un vertice in meno.

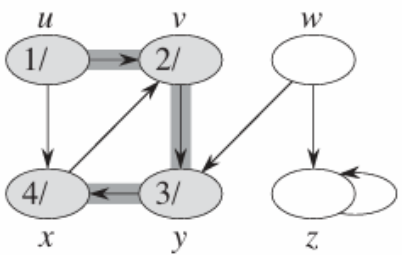
11.6 Visita in Profondità (DFS)

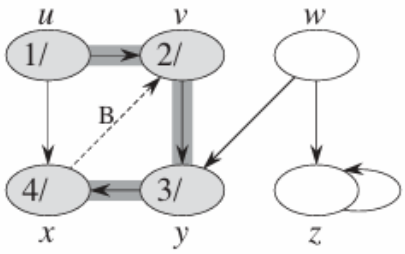
La **Visita in Profondità** o Depth First Search è un algoritmo di ricerca nei Grafi. Dato un grafo $G = (V, E)$ (orientato o meno) la DFS parte dall'ultimo vertice v scoperto che ha ancora **archi** non ispezionati che escono da esso. Quando tutti gli archi di v sono stati ispezionati, la DFS torna indietro per ispezionare gli archi che escono dal vertice dal quale v era stato scoperto. Questo procedimento verrà ripetuto finché non verranno scoperti dal vertice originale.
Inoltre se restano vertici non scoperti, allora uno di essi verrà scelto come nuovo **vertice sorgente** e la visita ripartirà da quest'ultimo. Anche questo procedimento si ripete fino a quando non verranno scoperti tutti i vertici del grafo. Inoltre:
• Se v è scoperto scorrendo la lista di adiacenza di u allora $pred[v] = u$;
• Il **sottografo dei predecessori** una DFS può essere formato da vari **alberi DF**, quindi non è altro che una **foresta DF**.

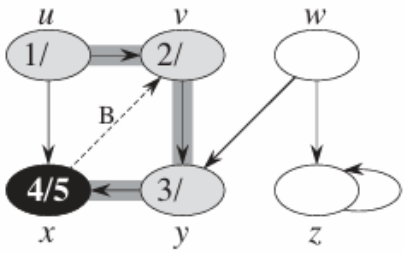
Come la BFS, anche la DFS per tenere traccia del lavoro svolto, colora i nodi del grafo di bianco, grigio o nero. Inizialmente tutti i nodi sono **bianchi**, ma verranno colorati di **grigio** durante la visita o di **nero** quando la sua lista di adiacenza è stata completamente visitata.
Questa tecnica garantisce che ogni vertice vada a finire in un solo albero DF, in modo che ogni albero sia disgiunto dagli altri.

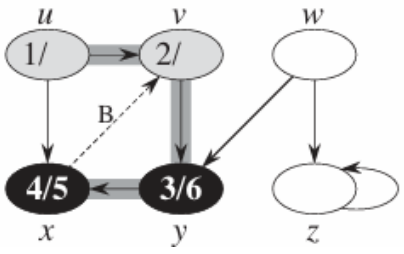


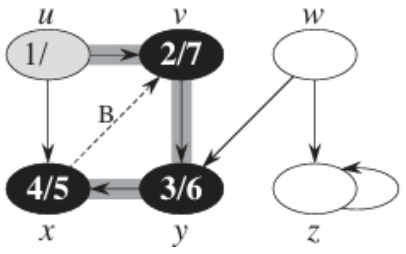

u
v
w
1/
2/
3/
x
y
z

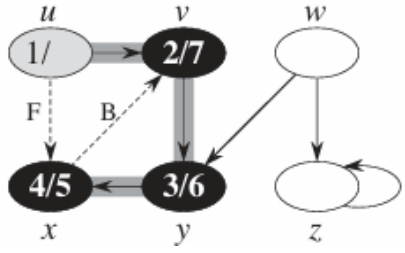

u
v
w
1/
2/
4/
3/
x
y
z

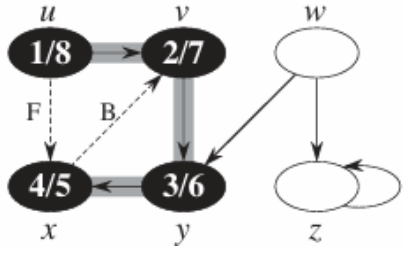

u
v
w
1/
2/
B
4/
3/
x
y
z

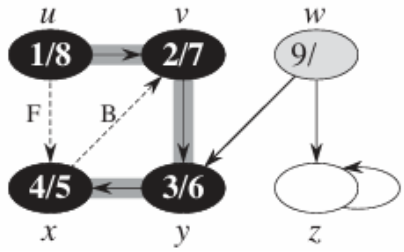

u
v
w
1/
2/
B
4/5
3/
x
y
z

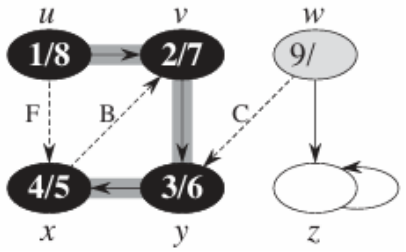

u
v
w
1/
2/
B
4/5
3/6
x
y
z

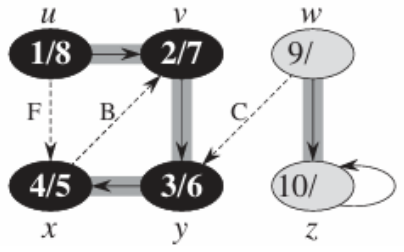

u
v
w
1/
2/7
B
4/5
3/6
x
y
z

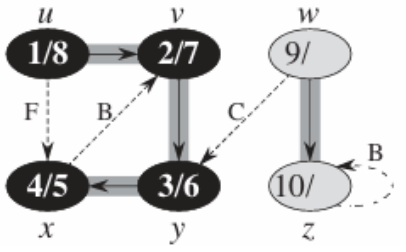

u
v
w
1/
2/7
F
B
4/5
3/6
x
y
z

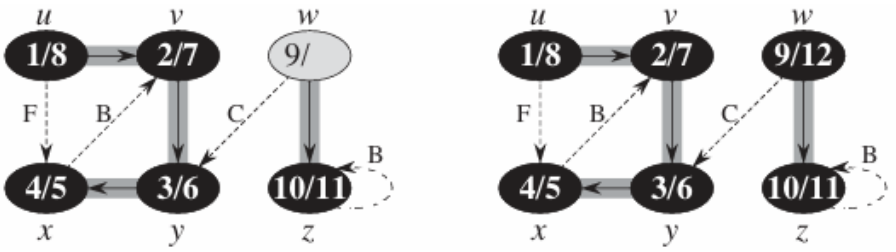

u
v
w
1/8
2/7
F
B
4/5
3/6
x
y
z


u
v
w
1/8
2/7
9/
F
B
4/5
3/6
x
y
z


u
v
w
1/8
2/7
9/
F
B
C
4/5
3/6
x
y
z


u
v
w
1/8
2/7
9/
F
B
C
4/5
3/6
10/
x
y
z


u
v
w
1/8
2/7
9/
F
B
C
4/5
3/6
10/
B
x
y
z



Complessità della procedura: $\Theta(V + E)$, in quanto la procedura **DFS** impiega $\Theta(V)$ mentre la procedura **DFS-Visit** impiega $\Theta(E)$.

Codice dell'algoritmo DFS (con Liste di Adiacenza):

```
DFS(){
    for(int i = 0; i < n; i++){
        colArr[i] = white;
        pred[i] = -1;
    }
    time = 0;
    for(int i = 0; i < n; i++){
        if(colArr[i] == white){
            DFS-Visit(i);
        }
    }
}
```

Codice dell'algoritmo DFS-Visit (con Liste di Adiacenza):

```
DFS_Visit(int u){
    colArr[u] = gray;
    dist[u] = time++;

    const Nodo* p = Adj[u].getHead();
    while(p != nullptr){
        int v = p->getData();
        if(colArr[v] == white){
            pred[v] = u;
            DFS-Visit(v);
        }
    }
    colArr[u] = black;
    fine[u] = time++;
}
```