



PROGRAMMAZIONE II Stefan Andrei Daniel

RIPASSO PROGRAMMAZIONE I

TORRI DI HANOI

RICORSIONE

SUCCESSIONE DI FIBONACCI (non da esame)

GITHUB

Creazione repository

COMPLESSITÀ'

Complessità temporale

Notazione asintotica

Notazione O

Funzioni notevoli

Notazione Ω

Notazione Θ

Analisi di algoritmi: esempi pratici

Complessità O

RICORRENZE(divide et impera)

Relazioni di ricorrenza

Limitare le sommatorie

RICORRENZE FONDAMENTALI

RICERCA DI UN ELEMENTO IN UN ARRAY

Ricerca dicotomica

ALGORITMI ITERATIVI DI ORDINAMENTO

Ordinamento per scambio

Selection sort

Insertion sort

ALGORITMI RICORSIVI DI ORDINAMENTO

Merge-Sort

Quick-Sort

PROGRAMMAZIONE OOP

Definizioni generali

Differenza tra una classe ed oggetto

Inline e non inline

MODIFICATORI DI ACCESSO

COSTRUTTORE DI COPIA

Static

Static e oggetti

Const

Const con le classi

Parentesi su alcuni argomenti di Programmazione I da ripassare bene

EREDITARIETÀ

Ereditarietà multipla

POLIMORFISMO E BINDING

CLASSI ASTRATTE

Astrazione delle classi

TEMPLATE

Uguaglianza fra istanze

FRIEND

Friend a classe

STRUTTURE DATI

Lista

Liste linkate semplici

Operazioni su liste

Controllo lista vuota

Inserimento in testa di un valore "b"

Inserimento in coda

Inserimento in modo ordinato

[Compilazione con header presenti in cartelle diverse dal main](#)

[Cancellazione di un nodo\(casi\)](#)

[Cancellazione del nodo di testa](#)

[Cancellazione del nodo di coda](#)

[Cancellazione di un nodo con valore specifico](#)

[LISTE DOPPIAMENTE LINKATE \(DL\)](#)

[Operazioni su DL List](#)

[Verifica DL List vuota](#)

[Inserimento in testa di un nodo in una DL List](#)

[Inserimento in coda di un nodo in una DL List](#)

[Inserimento di un nodo in modo ordinato in una DL List](#)

[Cancellazione della testa in una DL List](#)

[Cancellazione della coda di una DL List](#)

[Cancellazione di un nodo specifico in una DL List](#)

[LA PILA](#)

[Differenze fra pila limitata e pila non limitata](#)

[Implementazioni pila dinamica](#)

[Implementazione pila statica](#)

[Gestione degli errori per la dimensione della pila](#)

[Complessità delle operazioni dello Stack](#)

[CODA \(queue\)](#)

[Implementazione dinamica della coda](#)

[Implementazione statica della coda](#)

[ALBERI BINARI DI RICERCA](#)

[ALBERO BILANCIATO](#)

[ALBERO BINARIO DI RICERCA](#)

[Nomenclatura recap](#)

[Inserimento di un valore](#)

[Visita di un BST](#)

[Massimo e minimo di un BST](#)

[Successore e predecessore](#)

[Osservazioni](#)

[Cancellazione](#)

[Caso 1: z è una foglia](#)

[Caso 2: la chiave ha 1 figlio](#)

[Caso 2: la chiave ha 1 figlio \(codice completo\)](#)

[Caso 3: la chiave ha 2 figli](#)

NOTE:

[Esercizio: Simulare le seguenti operazioni](#)

[Algoritmi di visita](#)

[Gestione errori "throw"](#)

[Complessità operazioni BST](#)

[GRAFI](#)

[Rappresentazione dei grafi](#)

[LISTA DI ADIACENZA](#)

[Osservazioni](#)

[Codice liste di adiacenza](#)

[MATRICE DI ADIACENZA](#)

[Matrice di adiacenza codice](#)

[COMPLESSITÀ LISTA vs MATRICE DI ADIACENZA](#)

[Visita in ampiezza di un grafo \(Breadth-First Search\)BFS](#)

[Significato dei colori](#)

[Procedimento](#)

[Visita in profondità](#)

[Procedimento](#)

[Come ci si sposta dall'altra parte del grafo?](#)

Ordinamento topologico

[Pseudo codice BFS e complessità](#)

[Osservazioni sulla complessità](#)

[Pseudo codice DFS e complessità](#)

[Osservazioni sulla complessità DFS](#)

[ESEMPI DOMANDE ESAME](#)

[PROVA PRATICA: Come mi esercito?](#)

RIPASSO PROGRAMMAZIONE I

- La classe template di tipo `<typename T, int>` ha come secondo parametro un **tipo primitivo**. Pertanto, al momento dell'istanza degli oggetti, bisogna passare, come secondo parametro, dei valori ben definiti (o costanti) e non variabili.
- **Allocando staticamente** un array di 100 interi, si riservano 400 byte all'interno dello stack(quindi PILA).
- **int** ha una dimensione di 4byte
- **unsigned char** ha valori da 0 a 255
- **unsigned** serve per eliminare i valori negativi che una variabile può assumere e raddoppiare i suoi possibili valori positivi.
- Dopo aver fatto la **delete**, la memoria può ancora essere utilizzata dal programma ma è inaffidabile. La delete **non cancella la variabile** dal programma e **nemmeno i valori** ma dice al Sistema Operativo che quella **parte di memoria può essere riutilizzata**.

TORRI DI HANOI

Il problema delle Torri di Hanoi consiste nello spostare n dischi ordinati da una sorgente a una destinazione. L'unica regola è che un disco «**minore**»(dimensioni) nell'ordinamento **non può stare sotto** a un disco «**maggiore**»(dimensioni). I dischi non possono essere spostati in blocco. Per risolvere il problema si fa uso di un **“ausiliario”**.



RICORSIONE

Per ricorsione si intende un riutilizzo di un metodo più volte in un determinato tempo. La tecnica usata è quella dell'induzione matematica che ha:

1. **un caso base**
2. **passo induttivo**

Esempio con successione di numeri

caso base: se $n=0$, il successore è 1

passo induttivo: se $n>0$, il successore è $1+\text{succ}(n-1)$.

Es. $\text{succ}(3)=1+\text{succ}(2)=1+1+\text{succ}(1)=1+1+1=\text{succ}(0)=1+1+1+1=4 \rightarrow$ il successore di 3 è 4. Si torna sempre al caso base già definito $\text{succ}(0)$, quindi già noto. Si utilizza sempre la stessa funzione (ricerca del successore del precedente) fino ad arrivare al caso base.

Esempio con somma(a,b) sapendo fare il successore di un numero, quindi $n+1$. La dimostrazione per induzione è così definita:

- $\text{succ}(a) = a+1$ se $b=1$
- $1+\text{somma}(a, b-1)$ se $b>1$

soluzione alternativa: somma(a,b)

- a se $b=0$
- $1+\text{somma}(a,b-1)$ se $b>0$

Esempio con prodotto: $\text{prod}(a,b)$, con $a, b \neq 0$. ($a+a+a+a+\dots$) b volte

- a se $b=1$
- $a+\text{prod}(a,b-1)$ se $b>1$

Esempio con elevamento a potenza $\text{pow}(a,b) \rightarrow (a*a*a*a...)*b$ volte

- a se b=1
- 1 se b=0 (secondo caso base)
- $a * \text{pow}(a,b-1)$

Esempio con fattoriale di un numero n. $\rightarrow n!$

- 1 se n=0
- $n * \text{fatt}(n-1)$ se n>0

SUCCESSIONE DI FIBONACCI (non da esame)

Sequenza di numeri dove ogni numero è il risultato della somma dei due numeri precedenti.

$\text{Fib}(n)=\text{Fib}(n-1)+\text{Fib}(n-2)$. La sua sequenza è 1 1 2 3 5 8 13 21 ...

- caso base 0 e 1: $\text{fib}(0) = 1$ e $\text{Fib}(1) = 1$
-

Si ha una ricorsione doppia visti i due casi base!

Una ricorsione è sempre eliminabile (sia che sia di testa o di coda). La ricorsione di coda è quella ricorsione che è l'ultima funzione che verrà richiamata. Dopo di essa non ci sono altre operazioni. Possono essere sostituite con un'iterazione

vedi fibonacci.cpp

La sequenza di fibonacci di n che tende a infinito del rapporto fra $\text{Fib}(n-2) / \text{Fib}(n-1)$ dà il numero ϕ (phi) che rappresenta la sequenza aurea che vale circa 1.618...

Le funzioni ricorsive provocano un overhead(sovraccarico) perché una funzione viene richiamata più volte e si aggiungono una serie di chiamate/parametri.

Una funzione iterativa è MOLTO più veloce nell'esecuzione rispetto alla funzione ricorsiva. La ricorsiva è potente in caso di scrittura e leggibilità però, nel caso della sequenza di fibonacci, ogni funzione richiama se stessa 2 volte (le 2 somme dei 2 precedenti).

La ricorsione è utilizzabile con numeri bassi, non troppo alti. Ha senso utilizzarla su particolari tipi di algoritmi e particolari strutture dati.

Capitolo 19 Aguilar (approfondimenti)

GITHUB

- Un commit è un aggiornamento di un repository eseguito in un determinato file.
- Bisogna installare GIT.
- "git clone <https://github.com/marcomoltisanti/programmazione2-AA2122.git>" su terminale per aggiungere la repository di git su desktop in maniera sincronizzata non scaricando semplicemente il zip.
- Per aggiornare la repository su desktop su usa il comando "git pull".
- Bisogna scaricare "git bash" da "gitforwindows.com" per avere queste righe di codice disponibili e altri comandi.

Procedimento per aggiungere un file su github (remoto) da locale:

- git status → //vedo lo stato di git fra locale e remoto
- git add static_fun.cpp
- git status
- git commit -m "static inside functions" //m vuol dire "aggiungi un messaggio di commit"
- git push chiede il passphrase

Creazione repository

`git init` crea i file necessari. Una repository ha una **parte locale** e una **parte remota**.

La parte locale contiene dei file (nella cartella chiamata ".git") usati per le varie operazioni (di default).

Quando faccio l'operazione di "`add`" si inserisce **in un registro** che si deve tenere traccia del file aggiunto all'interno del "local".

Dopo le modifiche si avrà il file finito.

Dopo di che si fa il **commit** (il file è stato modificato e si deve tenere in considerazione per la parte "remote")

si fa il `push` (allinea il repository remoto con quello locale). Si va a vedere cosa c'è su "`add`".

Il push va a mettere il file preso in considerazione.

...or create a new repository on the command line

```
echo "# corso-git" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/overclokk/corso-git.git
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/overclokk/corso-git.git
git branch -M main
git push -u origin main
```

→ Come si **sincronizza** il **repository locale** con quello **remoto**?

```
git init //inizializza i vari file ".git"
git add static_fun.cpp //aggiungo il file che mi interessa
git status //mostra situazione del registro
git commit -m "first commit"
git status //dice che non c'è nessun nuovo file perchè ho "freezato" la situazione e non ho il disallineamento
git branch -M main //varie ramificazioni nelle quali posso dividere il mio repository
git remote add origin "link" //la parte remota si trova a questo indirizzo
git push -u origin main
```

ssh-key genera delle chiavi private e pubbliche da aggiungere al proprio repository per permettere di fare il push.

Si crea una nuova cartella dove, appunto, avrà key e key.pub che sono chiave privata e chiave pubblica. Una delle due va copiata nelle proprie impostazioni di github.

La chiave inizia per ssh-rsa (circa)

COMPLESSITÀ'

Il **costo di un algoritmo** è un prezzo da pagare per ottenere qualcosa in cambio. Esso dipende da:

- input;
- tempo;
- spazio.

```
Es. if(x%2==0) { //riga chiamata GUARDIA
// .. //CORPO della funzione
}
```

Si ha un **costo associato alla guardia** e un **costo associato al corpo**. Il costo **preponderante** è il costo del **corpo** del codice.

In altre parole, il costo è assimilabile al costo del corpo, appunto.

Il costo computazionale è la somma del costo di ogni singola istruzione che vanno da 0 a N , ovvero:

$$\sum_{i=0}^N C_i$$

C_i costo del corpo della funzione (blocco di codice) non ricorsiva, quindi **iterativa**.

Il costo dipende da N che sarà il numero di volte che si dovrà eseguire l'istruzione. [**while ($x < N$)**]

Esempio: ordinamento di 2 array, in modo crescente:

1. **0 1 2 3 4 5 6** è già ordinato in modo crescente e risulta il **caso migliore** perchè non esiste un ordinamento migliore che ci impiega meno tempo;
2. **6 5 4 3 2 1 0** non è ordinato ed è il **caso peggiore** perchè ci mette il massimo tempo possibile per ordinarlo in modo crescente

Fra il caso migliore e il caso peggiore si trova il **caso medio**:

1. **0 1 2 3 4 5 6** ← caso migliore
2. **3 1 0 5 6 2 4** ← caso medio
3. **6 5 4 3 2 1 0** ← caso peggiore

Complessità temporale

Se si fa partire un algoritmo di ordinamento su un array (vedi esempi sopra), quanto tempo si impiega?

Si assume che la complessità temporale è **proporzionale al numero di operazioni eseguite** ed è **indipendente dall'hardware considerato** perchè si sta cercando di stabilire il tempo di esecuzione di un codice rispetto all'input fornito.

Serve una relazione fra dimensione dell'input con il tempo di esecuzione.

La prima assunzione porta alla seconda assunzione che ha a che fare con la **dipendenza dalla dimensione dell'input** e con il **comportamento asintotico**.

Quindi **si assume** anche che non si vuole valutare il comportamento su specifici input ma si vuole valutare il comportamento asintotico, ovvero $n \rightarrow \infty$ “ n che tende a infinito”.

Notazione asintotica

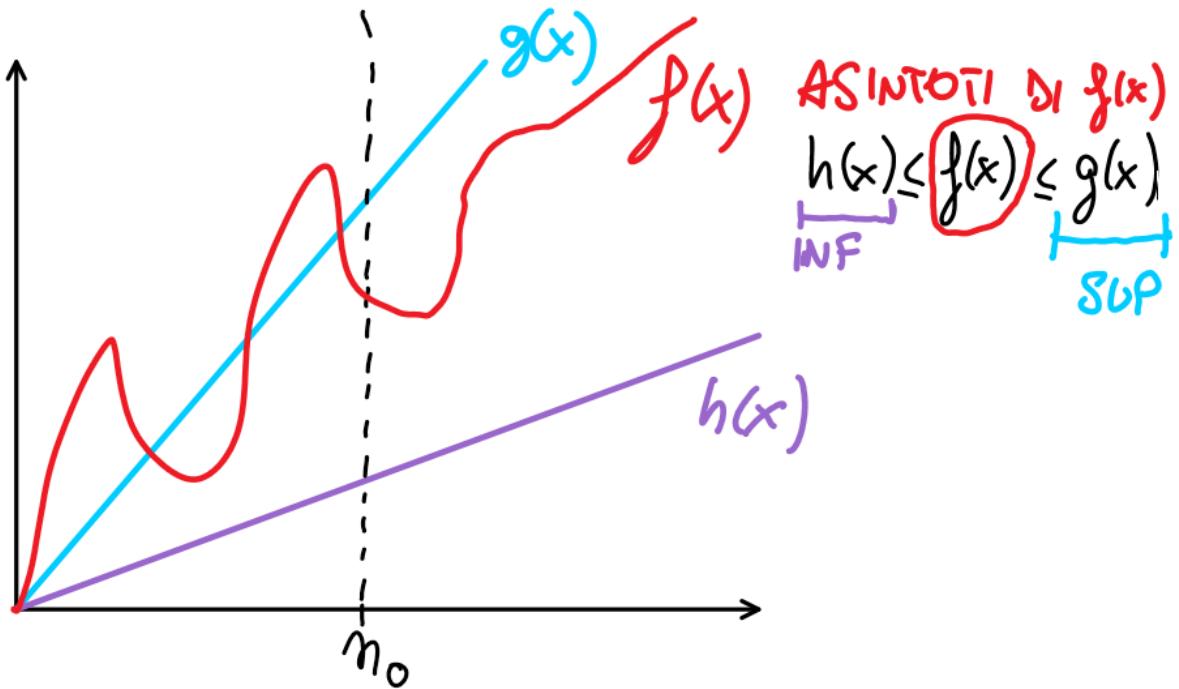
Considerato un algoritmo:

```
for(int i=0; i<N; i++) {  
    swap(i,i+1);  
    swap(i+2,i+1);  
    v[i]=-4;  
}
```

Costo computazionale 3 perchè ci sono 3 istruzioni nel for. Il costo generale dipende dalla dimensione dell'input N e questa è la valutazione della **complessità temporale**.

L'asintoto $g(x)$ **limita SUPERIORMENTE la funzione $f(x)$** perchè sta sopra la funzione $\forall n > n_0$

L'asintoto $h(x)$ **limita INFERIORMENTE la funzione $f(x)$** perchè sta sotto la funzione $\forall n > n_0$



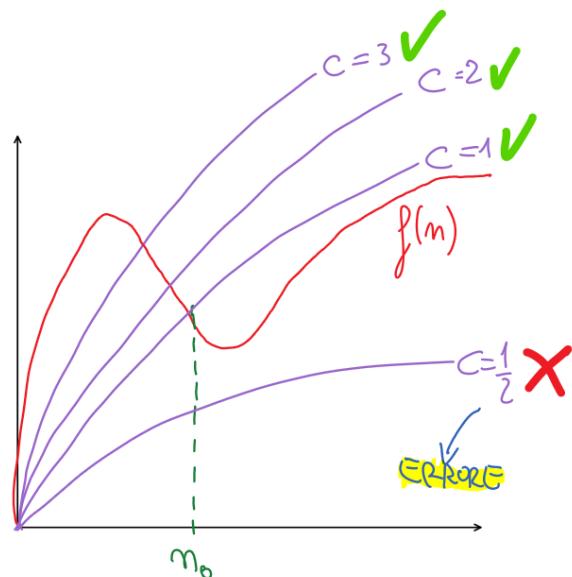
Notazione O

Si applica a delle funzioni $f(n)$ e si dice che $f(n)$ ha un **limite asintotico superiore** dato dalla funzione $g(n)$ se la funzione $g(n)$ **LIMITA SUPERIORMENTE**, in maniera asintotica, la funzione $f(n)$.

La funzione $O(g(n))$ stabilisce una **classe di equivalenza** e identifica **UN POSSIBILE LIMITE SUPERIORE**.

$$O(g(n)) = \{f(n) : \exists c, n_0 / 0 \leq f(n) \leq cg(n)\} \forall n > n_0$$

- Basta che esista almeno un valore di "c" che renda vera l'affermazione.
Infatti per $c = \frac{1}{2}$ non è vera perché **sta sotto g(n)**.
- $f(n) > 0$ **SEMPRE**.
- $f(n) < g(n)$ **SEMPRE oltre il valore** di ascissa n_0 .
- $f(n)$, oltre il valore di n_0 **tende** ad assumere il valore di $g(n)$



Esempio di POSSIBILI limiti superiori:

$f(n) = O(n \log n)$ oppure $O(n^2)$ oppure $O(n^2 \log n)$ oppure $O(n^4)$

Quale limite superiore prendiamo in considerazione?

Prendiamo in considerazione il più "piccolo", cioè $O(n \log n)$.

Per dire che $f(n)$ è $O(n)$, cioè $f(n) = O(n)$, posso usare:

1. $O(1)$, cioè costante;
2. $O(n^2)$, cioè quadrato;
3. $O(\log n)$, cioè logaritmico.

Vengono così identificate tutte le possibili relazioni di equivalenza.

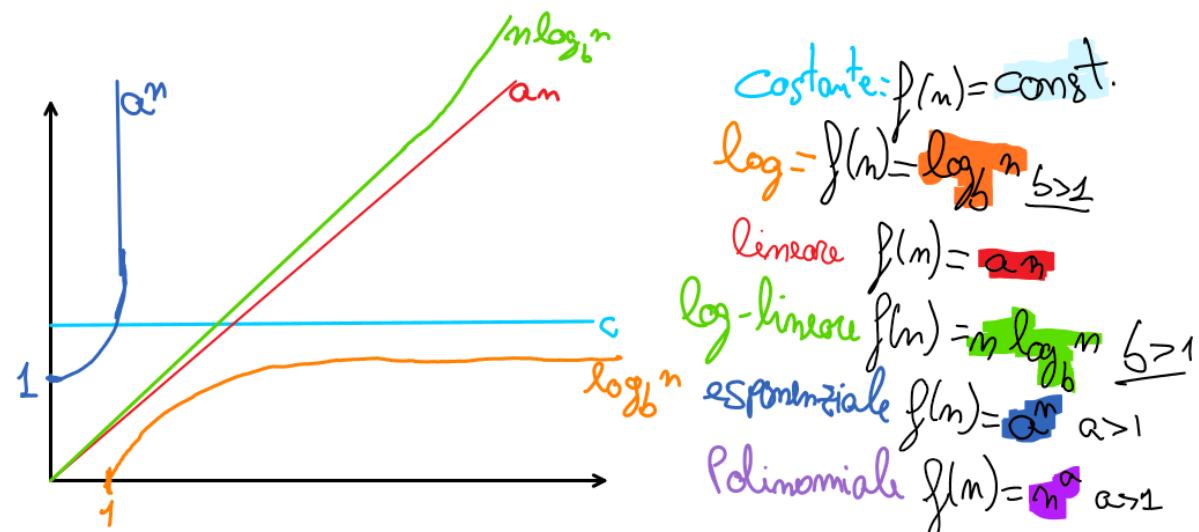
Ritornando al discorso dei "casi migliori/peggiori", in quale di questi casi ci troviamo con la notazione del limite asintotico superiore O ?

Non ci troviamo in nessuno dei 3 casi (migliore, peggiore, medio).

Funzioni notevoli

Le funzioni notevoli servono per definire le classi di equivalenza.

Non si può avere una funzione periodica come asintoto di una funzione.

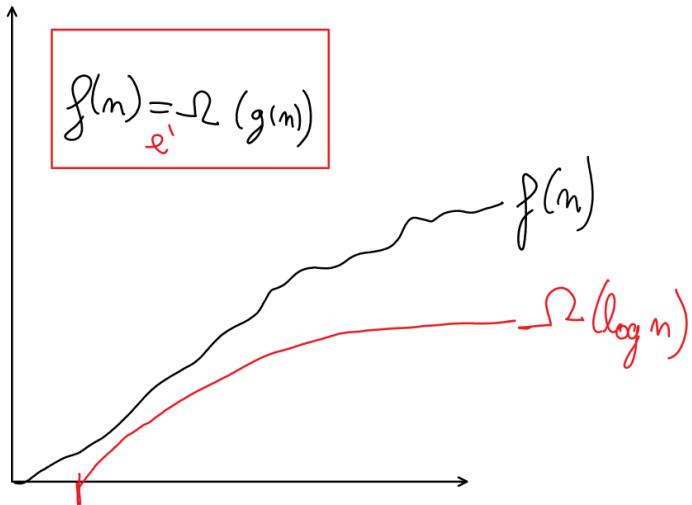


Notazione Ω

Ω definisce un **LIMITE INFERIORE**.

$$\Omega(g(n)) = \{f(n) : \exists c, n_0 / 0 \leq cg(n) \leq f(n)\} \forall n \geq n_0.$$

- $f(n) > 0$ SEMPRE;
- $f(n) > g(n)$ SEMPRE
oltre il valore n_0 ;
- $f(n)$ ha un limite inferiore $\Omega(\log n)$.

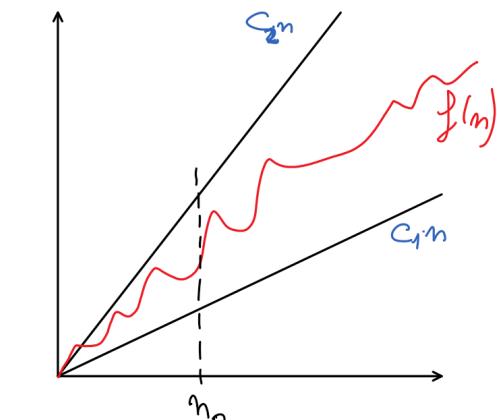


Notazione Θ

La notazione Θ definisce un **LIMITE INFERIORE** e un **LIMITE SUPERIORE** della funzione $f(n)$.

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 / 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\} \forall n \geq n_0$$

- $f(n) > 0$ **SEMPRE**;
- $f(n) > c_1 n$ E $f(n) < c_2 n$;
- $f(n)$ ha un limite inferiore e superiore.



Dal grafico, $c_2 n$ è il **limite superiore** della funzione $f(n)$ mentre $c_1 n$ è il **limite inferiore** della funzione $f(n)$. Pertanto la funzione $f(n)$ è la funzione Θ .

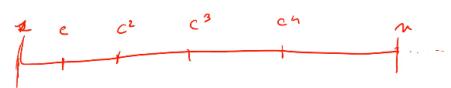
La notazione Θ indica che la funzione che prendo in considerazione $f(n)$ è limitata superiormente da $c_2 n$ e inferiormente da $c_1 n$.

“ O ” stabilisce una **famiglia di funzioni** che sono il **limite superiore** della funzione $f(n)$.

“ Ω ” stabilisce una **famiglia di funzioni** che sono il **limite inferiore** della funzione $f(n)$.

“ Θ ” stabilisce **equivalenza** fra $O(f(n))$ e $\Omega(g(n))$.

Analisi di algoritmi: esempi pratici



↑Esempio di una funzione esponenziale ↑

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

$O(n)$
 $O(\log n)$
 $O(\log n)$
 $O(n \log^2 n)$

Esempio

Ricerca di un elemento:

v[]	2	5	3	0	12	45	55	12	4
	v[0]	v[1]	...		v[n-1]				

```
int ricercaLineare (int vettore[], int dim, int chiave)
{
    for (int i = 0; i < dim; i++)
        if (vettore[i] == chiave) return i;
    return -1;
}
```

Caso migliore: 1
Caso peggiore: n
Caso medio: $(n+1)/2$

$$\sum_{i=1, N} \text{Prob}(el(i)) \cdot i = \sum_{i=1, N} (1/N) \cdot i$$

↳

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

$O(n)$

```
// c è una costante positiva
for(int i = 1; i <=n; i += c) {
    for (int j = 1; j <=n; j += c) {
        //espressioni con costo O(1)
    }
}
```

$O(n^2)$

- La complessità del primo esempio è $O(\log n)$ a

prescindere dal corpo della funzione. Quindi questa funzione, per terminare, impiega un tempo pari a $\log n$;

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

$O(\log n)$

- La complessità del secondo esempio è $O(\log n \log n)$ a prescindere dal corpo della funzione. Quindi questa funzione, per terminare, impiega un tempo pari a $(\log n \log n)$.

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

$O(\log \log n)$

Complessità O

- $O(1)$: complessità di una funzione o blocco di istruzioni ciascuna di costo $O(1)$, che **non contengono cicli, ricorsioni o chiamate ad altre funzioni non costanti**.
- $O(n)$: complessità di un **ciclo** quando le sue variabili (es. **contatore**) sono **incrementate/decrementate di una quantità costante**.
- $O(n^c)$: la complessità di **cicli annidati** è uguale al numero di volte in cui le istruzioni del ciclo interno vengono eseguite.

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

RICORRENZE(divide et impera)

Equazioni di ricorrenza sono denotate con "I".

La strategia "dividi e conquista" è così definita:

1. **Dividi**: si divide il problema in sottoproblemi;
2. **Conquista**: si risolvono i sottoproblemi ricorsivamente;
3. **Combina**: si combinano le soluzioni dei sottoproblemi ottenendo la soluzione del problema generale.

Relazioni di ricorrenza

La relazione di ricorrenza determina la prestazione di un algoritmo ricorsivo che si sta prendendo in considerazione, per esempio una sommatoria, che rappresenta, appunto, una funzione ricorsiva.

La sommatoria gode della proprietà di linearità gode della proprietà associativa e distributiva.

Prop. Associativa:

$$\sum_{k=1}^m (c a_k + b_k) = \sum_{k=1}^m c a_k + \sum_{k=1}^m b_k$$

Distributiva:

$$c \sum_{k=1}^m a_k = c \sum_{k=1}^m a_k$$

Sommatorie – Serie aritmetiche

La sommatoria

$$\sum_{k=1}^n k = 1 + 2 + \dots + n$$

è una serie aritmetica il cui valore è

$$\begin{aligned} \sum_{k=1}^n k &= \frac{1}{2} n(n+1) \underset{\text{PREVAUG}}{\approx} \frac{1}{2} n^2 + \frac{1}{2} n \\ &= \Theta(n^2). \end{aligned}$$

Sommatorie – Quadrati e cubi

$$\begin{aligned} \sum_{k=0}^n k^2 &= \frac{n(n+1)(2n+1)}{6} \underset{\text{PREVAUG}}{\approx} \Theta(n^3) \\ \sum_{k=0}^n k^3 &= \frac{n^2(n+1)^2}{4} \underset{\text{PREVAUG}}{\approx} \Theta(n^4) \end{aligned}$$

Sommatorie – Serie geometriche

Per x reale diverso da 1, la sommatoria

$$\sum_{k=0}^n x^k = 1 + x + x^2 + \cdots + x^n$$

è una serie geometrica (o di potenze) il cui valore è

$$\sum_{k=0}^n x^k = \frac{x^{n+1} - 1}{x - 1}.$$

Sommatorie – Serie armoniche

Per n intero positivo, l' n -esimo numero armonico è

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \\ &= \sum_{k=1}^n \frac{1}{k} \\ &= \ln n + O(1). \end{aligned}$$

Serie aritmetiche: la **base cambia** e l'esponente, nella sommatoria, non cambia;

Serie geometriche: **l'esponente cambia** e la base, nella sommatoria, non cambia.

Complessità di:

- Serie aritmetica: somma dei primi n numeri = $\Theta(n^2)$
- Serie armoniche: sommatoria di frazioni di numeratore 1 = $\ln n + O(1)$

Limitare le sommatorie

Data una sommatoria dei primi n numeri $\sum_{k=1}^n k = \frac{n(n+1)}{2}$ la dimostriamo con l'induzione aritmetica:

- Caso **BASE** → $k=1$
 - $n = 1 \rightarrow \sum_{k=1}^1 k = 1 = \frac{1 + (1+1)}{2} = \frac{2}{2}$
- Passo **INDUTTIVO** → Dimostro per $n + 1$ considerando **RISOLTO** il caso per $k = n$
 - $\sum_{k=1}^{n+1} k = \sum_{k=1}^n k + (n+1) =$
 $= \frac{1}{2}n(n+1) + (n+1)$
 $= \frac{1}{2}(n+1)(n+2)$
 - Ottengo, quindi: $\sum_{k=1}^{n+1} k = \frac{1}{2}(n+1)(n+2)$

RICORRENZE FONDAMENTALI

Questa formula si usa per programmi che ciclano sull'input "eliminando" un elemento per volta.

$$\Rightarrow C_N = \underbrace{(C_{N-1} + N)}_{\text{per } N \geq 2, C_1 = 1}$$

$$C_N \text{ è circa } \frac{N^2}{2}$$

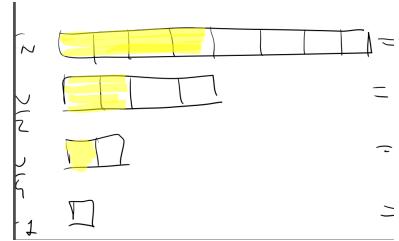
$$\begin{aligned} C_N &= C_{N-1} + N = & N \geq 2, C_1 = 1 \\ &= C_{N-2} + (N-1) + N = \\ &\quad \cdot C_{N-3} + (N-2) + (N-1) + N = \\ &\quad \cdot 2 + 3 + 4 + 5 + \dots + (N-1) + N = \\ &= C_1 + 2 + \dots + (N-2) + (N-1) + N = \\ &= 1 + 2 + \dots + (N-2) + (N-1) + N = \\ &= \frac{N(N+1)}{2} = \sum_{k=1}^N k \end{aligned}$$

Questa formula si usa tipicamente con un programma ricorsivo che dimezza l'input ad ogni passo

$$C_N = C_{\frac{N}{2}} + 1 \quad \text{per } N \geq 2, C_1 = 1$$

C_N è circa $\ln(N)$

$$\begin{aligned} C_N &= \left(C_{\frac{N}{2}} + 1 \right) + 1 = \left(C_{\frac{N}{4}} + 1 \right) + 1 \\ &= C_{\frac{N}{2}} + 1 + 1 + 1 \dots = \\ &= C_4 + 1 + \dots + 1 = C_2 + 1 + \dots + 1 \\ &\approx C_4 + 1 + \dots + 1 = 2 + 1 + \dots + 1 \end{aligned}$$

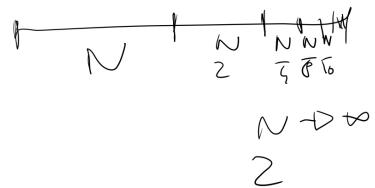


Questa formula si usa tipicamente con un programma ricorsivo che dimezza l'input ed esamina, eventualmente, ogni elemento di esso

$$C_N = C_{\frac{N}{2}} + N \quad \text{per } N \geq 2, C_1 = 0$$

C_N è circa $2N$

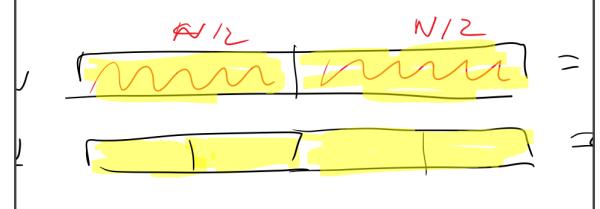
$$\begin{aligned} & \text{Diagram showing an array of length } N \text{ being divided into two halves of length } \frac{N}{2}. \\ & \text{Diagram showing a half-length array of length } \frac{N}{2}. \\ & \text{Diagram showing a quarter-length array of length } \frac{N}{4}. \\ & \text{Diagram showing an eighth-length array of length } \frac{N}{8}. \\ & \text{Diagram showing a 1x1 array.} \\ & \text{Final diagram showing a square symbol indicating the base case.} \end{aligned}$$



Questa formula si usa tipicamente con un programma ricorsivo che esegue una scansione lineare dell'input prima, durante, oppure dopo aver suddiviso l'input in due parti.

$$C_N = 2C_{\frac{N}{2}} + N \quad \text{per } N \geq 2, C_1 = 0$$

C_N è circa $N \log(N)$



Questo formula si usa tipicamente con un programma ricorsivo che dimezza l'input ed esegue una quantità di lavoro addizionale costante.

$$C_N = 2C_{\frac{N}{2}} + 1 \quad \text{per } N \geq 2, C_1 = 1$$

C_N è circa $2N$

RICERCA DI UN ELEMENTO IN UN ARRAY

Dato un array con random elementi (10 carte) devo trovare dei numeri con la ricerca lineare.

10 8 7 6 4 9 2 3 5 1

Trovare 5,3,7

1. Per il 5 si fanno **9 controlli**;
2. Per il 3 si fanno **8 controlli**;
3. Per il 7 si fanno **3 controlli**.

Mescolo le carte...Ottengo:

7 3 8 1 9 6 4 10 2 5

Trovare 5,3,7

1. Per il 5 si fanno **10 controlli**;
2. Per il 3 si fanno **2 controlli**;
3. Per il 7 si fa **1 controllo**.

Mescolo le carte...Ottengo:

10 1 8 4 2 5 6 7 3 9

Trovare 5,3,7

1. Per il 5 si fanno **6 controlli**;
2. Per il 3 si fanno **9 controlli**;
3. Per il 7 si fanno **8 controlli**.

Si fanno mediamente 5 controlli e in **termini di complessità** si tratta di $O(n)$.

Se, invece, prendo le carte e le dispongo in maniera crescente ho **1 2 3 4 5 6 7 8 9 10**, allora esiste un metodo (**RICERCA DICOTOMICA**) che permette di cercare un numero all'interno di un array ordinato.

Ricerca dicotomica

Se dovessi cercare il numero 7, i **passaggi** sono i seguenti:

1. **Divido l'array a metà**. Se la chiave è più grande della metà che considero, allora prendo in considerazione la metà verso destra o viceversa;
2. Prendo di **nuovo l'elemento centrale** dell'array e valuto se la chiave è **più grande o più piccola** dell'elemento centrale;
3. Allora valuto la chiave con gli elementi rimasti e trovo il numero 7.

In questo caso si fanno **solo 3 controlli** rispetto ai **mediamente 5 controlli** di prima fatti con la ricerca lineare.

In caso di ricerca lineare, allora in media si fanno $n/2$ controlli $\rightarrow O(n)$

In caso di ricerca dicotomica (o binaria) , conoscendo la disposizione dell'array, è possibile ridurre il numero di passaggi.

```
//RICERCA DICOTOMICA
#include <iostream>

using namespace std;

//binary Search
//è sottinteso che l'array sia già ordinato
bool ricercaBinaria(int array[], int n, int key){
    bool found = false;
    //verificare ad ogni iterazione se key>=array[n/2]
    // int midpoint = n/2; //floor(n/2)
    int start = 0;
    int end = n;

    while( !found && (start!=end ) ){
        int midpoint = start + ((end-start)/2); // è uguale (start+end)/2
        cout << "start=" << start << ", end=" << end << " midpoint=" << midpoint << endl;

        if(array[midpoint]==key)
            found=true;
        else if (key < array[midpoint]){
            //considerare la metà inferiore
            //aggiornare end
            end=midpoint;
            // midpoint=(end-start)/2;
        }
        else {
            //considerare la metà superiore
            //aggiornare start
            start=midpoint+1;
            // midpoint=(end-start)/2;
        }
    }
}
```

```

        }
    }

    return found;
}

int main(){
    int array[]={1,2,3,4,5,6,7,8,9,10};

    cout << ricercaBinaria(array,10,7);
}

```

ALGORITMI ITERATIVI DI ORDINAMENTO

Ordinamento per scambio

Mescolo di nuovo le 10 carte e le metto sul tavolo...

Prendo la prima carta e la **confronto** con **il successivo**.

→ Faccio lo scambio quando la carta presa in considerazione è **più piccola** della carta nella posizione *i*-esima.

1. Prendo una carta in mano lasciando libera una posizione;
2. se la **carta *i*-esima** è **più piccola della carta che ho in mano**, allora **scambio le posizioni**, quindi avrò in mano una carta più piccola rispetto a quello che ho preso in precedenza;
Quindi avviene uno **swap** fra le due carte e la carta *i*-esima è la mia nuova carta attuale;
3. altrimenti non scambio e confronto gli elementi successivi dell'array fino alla fine.
4. Quando l'array finisce, prendo la carta successiva
5. *Si fanno dai 10 ai 90 scambi circa (nel caso peggiore 90, ovvero quando l'array è ordinato al contrario)*

```

void ordinamento (int array[], int n){
    for(int i=0; i<n; i++){
        for(int j=i+1; j<n; j++){
            if( array[j] < array[i] ){
                swap(array[i],array[j]);
            }
        }
    }
}

```

Selection sort

Mescolo il mazzo...Ottengo → 7 2 4 8 6 3 5 1 9 10.

Stendo le carte sul tavolo. Prendo la prima carta e verifico, per il resto dell'array (in avanti), qual è **l'indice dell'elemento più piccolo**.

1. prendo il 7. Cerco l'elemento di indice minore negli elementi successivi al 7. Scorro l'array. Non faccio gli scambi, come prima, ma voglio solo vedere qual è l'indice dell'elemento più piccolo.
In questo caso l'elemento più piccolo è a indice 7, quindi dove c'è l'uno (1);
2. faccio lo **scambio** fra il **primo elemento preso in considerazione** e **l'indice dell'elemento più piccolo** appena trovato.
3. Quindi avrò 1 2 4 8 6 3 5 7 9 10
4. Ripeto il procedimento per i successivi numeri in maniera iterativa.

```

void selectionSort (int array[], int n){
    int indexMin;

    for(int i=0; i<n-1; i++){
        indexMin=i;
        for(int j=i+1; j<n; j++){
            if( array[j] < array[indexMin] ){
                indexMin=j;
            }
        }
        swap(array[i],array[indexMin]);
    }
}

```

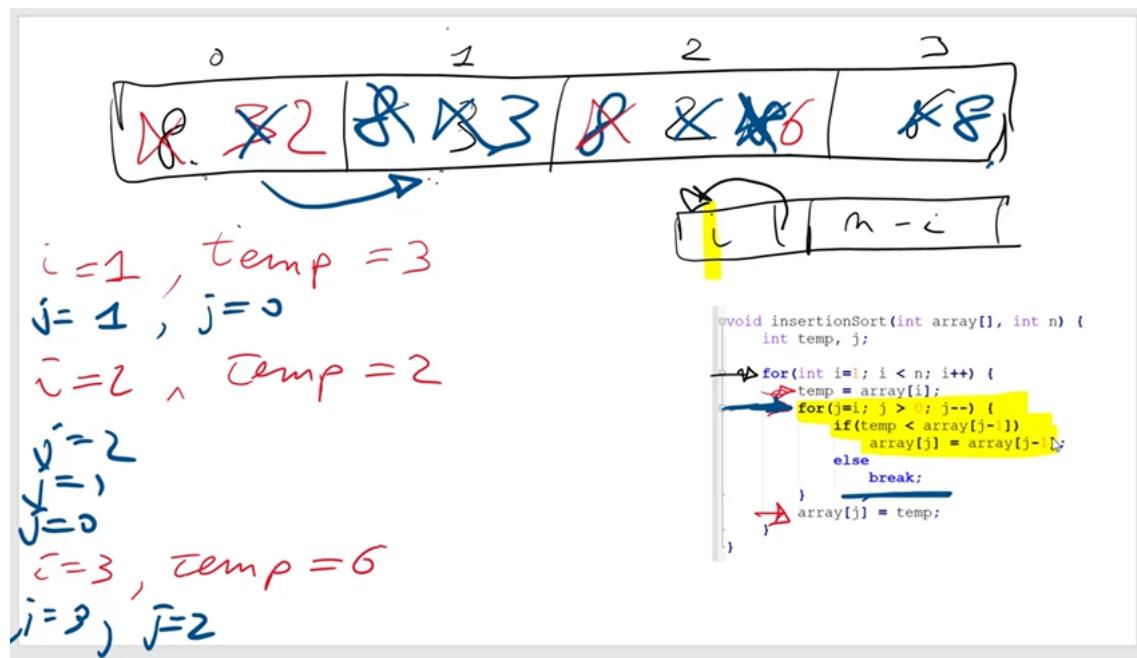
```
    }  
}
```

Insertion sort

Mescolo le carte e le metto sul tavolo (senza distenderle).

1. pescò una carta e la metto in posizione 0.(un vettore di un solo elemento viene considerato ordinato)
2. pescò un'altra carta e la metto in posizione corretta rispetto al sottoarray che ho (di una carta soltanto, quindi un elemento). Quindi ad ogni passaggio, la dimensione del sotto-array aumenta di 1.
3. pescò la prossima carta e ripetò i passaggi, quindi mettendola nella posizione giusta.

```
void insertionSort(int array[], int n){  
    int temp, j;  
  
    for(int i=1; i<n; i++){ //considero il primo elemento già ordinato  
        temp = array[i];  
        for(j=i; j>0; j--){  
            if(temp < array[j-1]){  
                array[j] = array[j-1];  
            }  
            else{  
                break;  
            }  
        }  
        array[j] = temp;  
    }  
}
```



Si può modificare la parte interna del codice (evidenziato in giallo) con una ricerca binaria (**dicotomica**) visto che la parte a sinistra dell'array che si prende in considerazione già è ordinato. [piuttosto che eseguire una ricerca binaria].

→ Implementare la ricerca dicotomica al posto della parte di codice evidenziata in giallo per esercizio

13 algoritmi sopra descritti hanno tutti complessità $O(n^2)$.

(ESAME)

Stima della complessità di un codice. Vedere quale variabile aumenta più esponenzialmente rispetto alle altre. Per esempio:

$$T(n) = n^5 + n^4 + n = O(n^5)$$

$$S(n) = \log n + \frac{n}{2} + \frac{n^2}{4} = O(n^2)$$

$$U(n) = T(n) + S(n) = O(n^5)$$

$$O(n^5) + O(n^2) = O(n^5)$$

ALGORITMI RICORSIVI DI ORDINAMENTO

Si applica la strategia “**Divide et Impera**” seguito da un passo “**Combine**” e consiste nel:

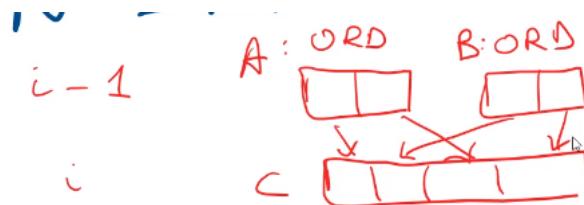
1. **suddividere il problema** in sottoproblemi più piccoli;
2. **risolvere i sottoproblemi** in maniera *indipendente*;
3. **combinare le soluzioni** dei sottoproblemi in maniera che venga mantenuta la soluzione generale che si sta cercando.

In caso dell'ordinamento il problema generale è ordinare gli elementi di un array in modo crescente. In questo caso:

- considero **frazioni sempre più piccoli di array** fino ad arrivare ad un caso base (quando c'è solo un elemento che si considera sempre ordinato) e quindi arrivare ad avere sottoproblemi di dimensioni 1.
1. **spezzare l'array** fino ad ottenere sottoproblemi di dimensione 1
 2. nel **caso base** non si deve fare nulla
 3. **combinare i sottoproblemi**. Questo passo risulta essere sempre il più importante visto che si tratta di “*logica*”. (è *il cuore pulsante della strategia*)

Merge-Sort

Si passa da un problema di dimensione n e si arriva a n sottoproblemi di dimensione 1.



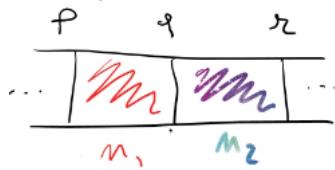
Se al passo $i-1$ ho 2 array A e B ordinati, al passo i avrò un array C dove A e B verranno **smistati** in modo tale da mantenere l'ordinamento.

La procedura prende in input un array e 3 indici $p \ q \ r$, quindi `Merge(A, p, q, r)` dove:

1. **p** = indice **inizio sottoarray** che si sta considerando;
2. **q** = $p \leq q \leq r$, indice “di mezzo”;
3. **r** = **fine sottoarray** che si sta considerando.

$n1=q-p+1$ è la **dimensione del sottoarray di destra**

$n_2 = r - q$ è la dimensione del sottoarray di sinistra



Creo due array di supporto $L[n_1]$ e $R[n_2]$ e li riempio:

- in L metto tutti gli elementi di A che vanno da $[p \rightarrow q]$
- in R metto tutti gli elementi di A che vanno da $[q \rightarrow r]$

Posso mostrare il procedimento così: $L \leftarrow A[\text{da } p \text{ a } q]$ e $R \leftarrow A[\text{da } q \text{ a } r]$

Ad L ed R aggiungo due **elementi di supporto** che sono:

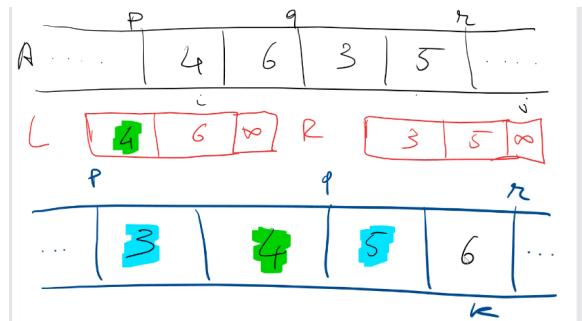
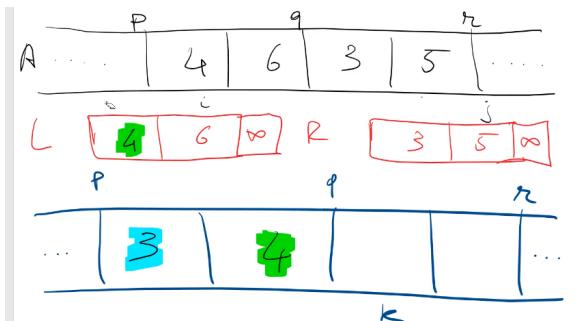
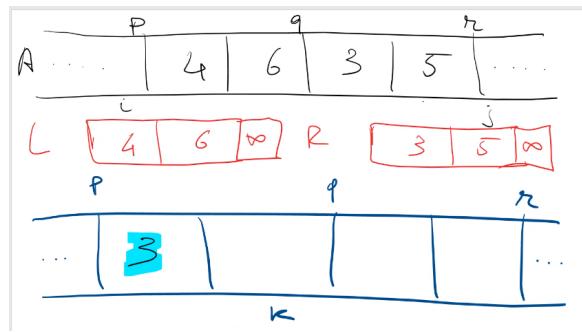
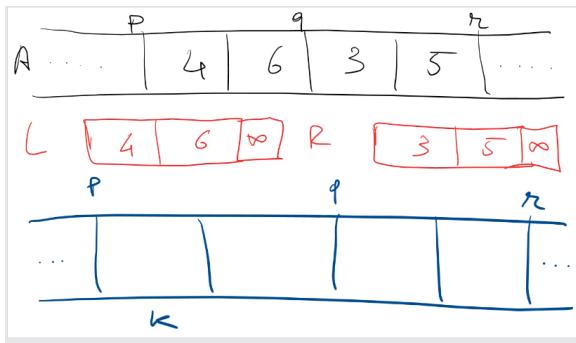
$L[n_1+1] = \infty$ e $R[n_2+1] = \infty$ e permettono di far finire i confronti, quindi servono per evitare un loop infinito.

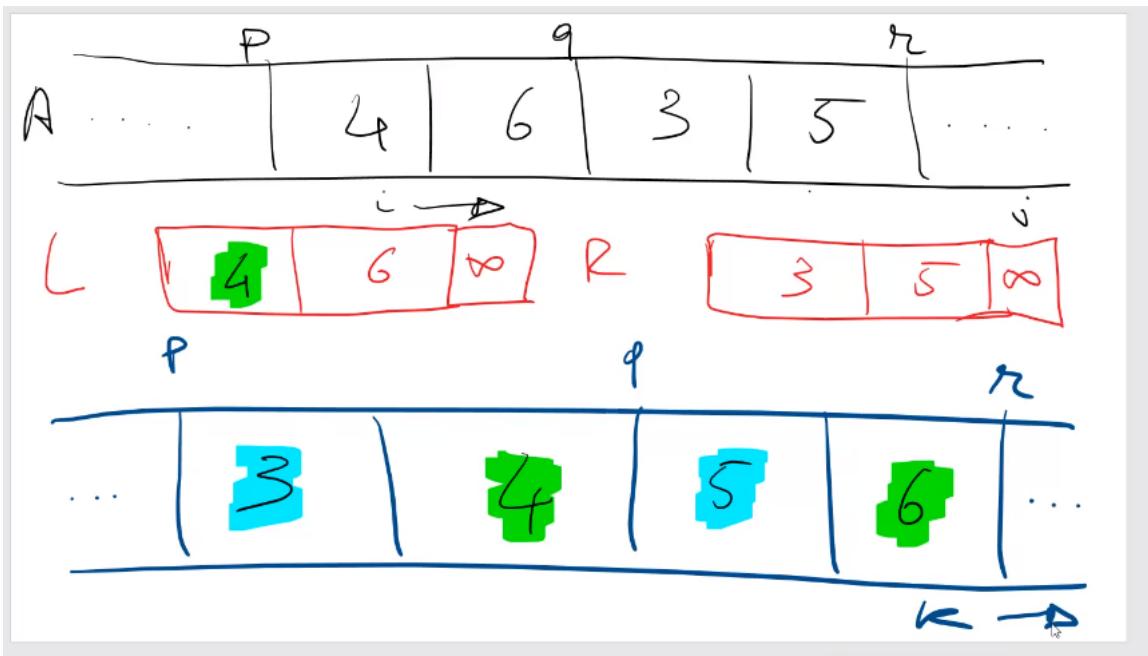
$L \leftarrow L[p \rightarrow q]$
 $R \leftarrow R[q \rightarrow r]$
 $L[m_1+1] \leftarrow \infty$
 $R[m_2+1] \leftarrow \infty$
 for $R = p \dots r$

MERGE (A, p, q, r)
 $m_1 = q - p + 1$
 $m_2 = r - q$
 Crea due array L, R
 $L \leftarrow A[p \rightarrow q]$
 $R \leftarrow A[q \rightarrow r]$
 $L[m_1+1] \leftarrow \infty$
 $R[m_2+1] \leftarrow \infty$
 for $R = p \dots r$, $i = 1, \dots, m_1, j = q, \dots, m_2$

$p = \text{INDICE INIZIALE SOTTOARRAY}$
 $q = \text{INDICE FINALE SOTTOARRAY}$
 $r = \text{INDICE DI } A$
 $p \leq q \leq r$
 for $R = p \dots r$, $i = 1, \dots, m_1, j = q, \dots, m_2$
 if $L[i] \leq R[j]$
 $A[k] = L[i]$
 $i++$
 else
 $A[k] = R[j]$
 $j++$
 $k++$

Creo 3 indici, k i j che scorrono rispettivamente A, L, R
 Quindi avrò $A[k]$ $L[i]$ e $R[j]$ come elementi dei rispettivi array!





La procedura **merge** richiede un tempo pari a n . Richiede $O(n)$ o $\Theta(n)$? Richiede tempo pari a $\Theta(n)$.

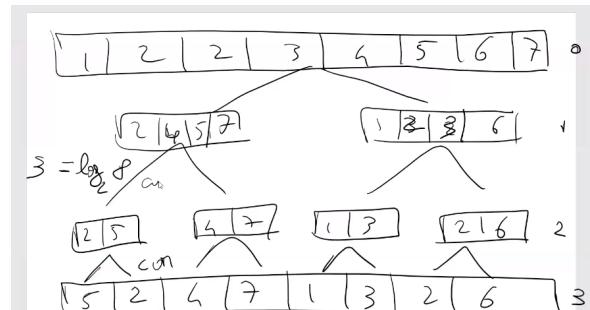
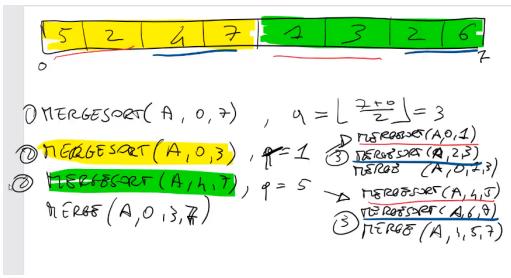
E' possibile fare meno di n passi? no.

Avrò `mergesort(A, p, r)` e `merge(A, p, q, r)` quindi 2 funzioni ricorsive.

```

if (p<r) {
    q=floor(p+r)/2;
    mergesort (A,p,q); //porzione sinistra di array
    mergesort(A,q+1,r); //porzione destra di array
    merge(A,p,q,r); //unione ordinata dei due array
}

```



Il metodo merge ha impatto sulla complessità spaziale. Serve 2 volte la dimensione dell'array in memoria.

```

void merge(int A[], int p, int q, int r){
    int n1=q-p+1;
    int n2=r-q;

    // cout << "INDICI: " << p << " " << q << " " << r << endl;

    int* L=new int[n1+1];
    int* R=new int[n2+1];

    //adesso devo procedere alla copia negli array L ed R
    for(int i=0; i<n1; i++){
        L[i]=A[p+i];
    }

    for(int j=1; j<=n2; j++){

```

```

        R[j-1]=A[q+j];
    }

/*
for(int j=0; i<n2; j++)
    R[j]=A[q+j+1];
*/

L[n1]=INT_MAX; //inserisco il valore sentinella, ovvero "infinito"
R[n2]=INT_MAX; //inserisco il valore sentinella, ovvero "infinito"

// cout << "L= " << endl;
printArray(L,n1+1);
// cout << "R= " << endl;
printArray(R,n2+1);

int i=0, j=0, k=p;

for(k=p; k<=r; k++){
    if(L[i] < R[j]) {
        A[k]=L[i];
        i++;
    }
    else {
        A[k]=R[j];
        j++;
    }
}

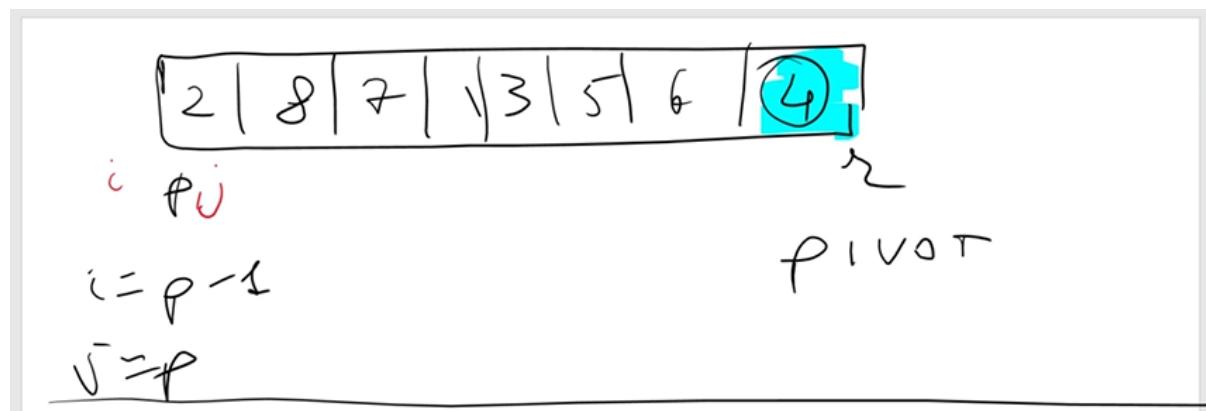
delete[] L;
delete[] R;
// delete L; // elimina solo il primo elemento dell'array.
// delete L[]; //non compila
//delete[] L è corretto
}

void mergesort(int A[], int p, int r){ //quindi avrà 2 funzioni ricorsive
    if(p<r){
        int q = floor((p+r)/2.0);
        mergesort(A,p,q); //metà sinistra dell'array
        mergesort(A,q+1,r); //metà destra dell'array
        merge(A,p,q,r);
    }
}
}

```

Quick-Sort

Questo tipo di ordinamento ordina gli elementi “sul posto” e quindi non ha bisogno di array di appoggio (ausiliari). Per esempio si avrà:



Si prende un elemento **a caso** nell'array, chiamato **PIVOT**. In base a questo, gli elementi vengono spostati a destra (o a sinistra di esso) in base al tipo di ordinamento.

→ Anche in questo caso avrò gli indici di **inizio**(*p*), **fine**(*r*) e l'indice **intermedio** (*q*)

→ Si deve far scorrere **j** da **p** a **r-1**.

- Se l'elemento `A[j] ≤ pivot` allora faccio **avanzare la i** (`i++`) e scambio `A[i]` con `A[j]`, ovvero `swap(A[i], A[j])`. E così via in modo ricorsivo.
- Alla fine** scambio pivot con `A[i+1]`, ovvero `swap(pivot, A[i+1])`.

Alla fine si avrà un array diviso in “partizione con numeri <pivot”, “partizione con numeri >pivot” e al centro il “pivot”.

Per esempio potrei avere, dopo aver eseguito il quick-sort: 2 1 3 7 6

Gli elementi **prima del pivot** sono **più piccoli** di esso. Gli elementi **dopo il pivot** sono **più grandi** di esso.

*Dopodichè bisogna applicare di nuovo l'**algoritmo di ordinamento** sul sottoarray contenente elementi minori del pivot e successivamente applicare lo stesso algoritmo al sottoarray contenente elementi maggiori di pivot.*

→ Alla fine di tutte le ricorsioni si otterrà un **array ordinato di pivot**.

Quindi la funzione sarà `quicksort(A, p, r)`.

```
if p<r{
    q=partition(A,p,r); //mette il pivot in una posizione dove a sx ci sono elementi più piccoli e a destra elementi più grandi
    quicksort(A,p,q-1);
    quicksort(A,q+1,r);
}
```

→ Con un pivot circa uguale alla metà del valore massimo fra elementi, allora la procedura lavora in maniera ottimale.

Però visto che l'elemento che viene preso in considerazione è un **elemento random** dell'array e quindi può essere molto vicino al valore massimo o minimo oppure il massimo/minimo stesso. In questo caso si avrà una complessità “quadratica”.

Si assume che:

- la scelta del pivot è **casuale**, cioè non si sta facendo alcuna considerazione sul suo valore effettivo. (Se ho un array con 100k posizioni di valori numerici e la distribuzione dei numeri sarà più o meno omogenea, ho 1/100k di probabilità di beccare un numero random)
- alla fine** dell'algoritmo si restituisce **i+1** ovvero la **posizione finale del pivot** nell'array, sapendo che a sinistra del pivot i numeri sono più piccolo e a destra del pivot sono più grandi.
- la **complessità** diventa **quadratica** se il **pivot** diventa **il massimo** o **il minimo** oppure, in caso di grandi numeri di elementi, quando si trova circa ai limiti del minimo o massimo.

→ *Implementare il Quick-Sort per esercizio!*

```
template <typename T>
inline int partition(T* array, int start, int end) {
    //Scelgo l'ultimo elemento come pivot
    T pivot = array[end];
    //l'indice j rappresenta l'indice di separazione tra gli elementi
    //più piccoli del pivot (a sinistra) e quelli più grandi (a destra)
    int j = start;
    //l'indice i serve a scorrere tutti gli elementi, compreso il pivot
    for(int i=start; i<=end; i++) {
        //se l'elemento che considero è più piccolo del pivot
        //lo metto nella parte destra, incrementando l'indice j
        n_comparisons++;
        if((array[i] < pivot) || (array[i] == pivot)) {
            swap(array[i], array[j]);
            j++;
        }
    }

    //Decomenta la riga successiva per vedere gli indici di partizionamento
    //std::cout << "Partition from " << start << " to " << end << " - PIVOT IN " << j-1 << std::endl;

    //dato che abbiamo scambiato anche il pivot, l'indice j segnerà la posizione successiva a quella del pivot
    //e quindi restituiamolo j-1
    return j-1;
}

template <typename T>
inline void quicksort(T* array, int start, int end) {
```

```

if(start >= end)
    return;

int q = partition(array, start, end);
//nella procedura partition il pivot viene posizionato correttamente in q, quindi dobbiamo
//effettuare la chiamata ricorsiva sulle partizioni destra e sinistra
//partizione sinistra da start a q-1
quicksort(array,start, q-1);
//partizione destra da q+1 a end
quicksort(array,q+1, end);
}

```

PROGRAMMAZIONE OOP

Principi base della OOP:

1. **Astrazione**: da qualcosa di reale cerco di tirar fuori le caratteristiche descrittive e le uso per descrivere l'entità nella sua **"essenza"**. Si hanno **interfacce comuni** a tutti gli oggetti di quel determinato tipo;
→ *Per esempio*: uno studente ha una matricola, nome, cognome e altri dati personali e ha anche dei **comportamenti**, tipo *andare a lezione, studiare, scrivere, fare esami* ecc.. ←
2. **Incapsulamento (information-hiding)**: si usano **modificatori di accesso** così da gestire le informazioni pubbliche e le informazioni private (che verranno nascoste all'esterno);
3. **Ereditarietà**: se nel processo di astrazione rilevo delle caratteristiche comuni, allora potrebbe avere senso introdurre **un altro livello di astrazione** che permette di alzare le caratteristiche comuni in modo da avere una classe base che abbia queste caratteristiche comuni.
→ *Per esempio* la classe Studente (*con tutte le informazioni personali*). La classe Lavoratore (*con info personali*). Le **informazioni comuni** possono essere nome e cognome. Si ha quindi una classe base (**superclasse**) dalla quale le altre classi (**sottoclassi**) ereditano.
Esso è un meccanismo che permette un livello di astrazione **GERARCHICO** che consente di derivare classi da altre classi in modo da conservare le proprietà della classe base nelle classi derivate.
4. **Polimorfismo**: **funzioni** che hanno lo **stesso nome** ma assumono **comportamenti diversi** a seconda del chiamante.
Per esempio una classe Persona che ha un metodo "print()". Da questa classe derivano due classi, ovvero Studente e Lavoratore ed entrambe ridefiniscono la funzione "print()". Quindi la funzione *print()* è la **funzione polimorfa**.

Definizioni generali

Quando si ha una classe, si hanno dei **DATI**, o **attributi**, e delle **FUNZIONI**, o **metodi**. I dati sono variabili di qualunque tipo. Essi, quando sono all'interno della classe, sono detti **membri della classe** e prendono il nome di attributi e metodi.

struct è una keyword che permette di fare qualcosa di simile alle classi. Identifica una collezione di dati e/o funzioni. Struct **non permette** di fare **astrazione** e **incapsulamento** come nella OOP e per questo motivo non può essere usata per la OOP.

Le classi forniscono la possibilità di modificare i livelli di accesso ai membri della classe stessa.

Nelle **classi** il **default** dei modificatori di accesso è **private**. Nelle **struct** il **default** dei modificatori di accesso è **public**.

Differenza tra una classe ed oggetto

Classe=definizione di un prototipo. Oggetto=istanza della classe che viene allocata in memoria (*quindi occupazione di byte*)

Quando definisco una classe, definisco un prototipo di alcuni metodi e attributi. Si ha il passaggio ad **OGGETTO** quando la **classe viene creata** e/o **istanziata in memoria**.

Il responsabile dell'istanza dell'oggetto è il **metodo Costruttore**:

Ha lo stesso nome della classe. Non ha nessuno tipo di ritorno, nemmeno **void**, ed è responsabile dell'inizializzazione (o non inizializzazione) dei valori degli attributi.

```

class A{
public:
    A(); //costruttore di default
};

```

Ogni classe ha bisogno di un costruttore (non obbligatorio perché il compilatore istanzia la classe usando il metodo “membro a membro” ed è poco affidabile, ma è super raccomandato averlo)

Si hanno i costruttori:

1. **default** (senza parametri);
2. “**completo**” (ovvero che prendono in *input tutti i parametri necessari*)
3. **Copia**

→ **Esercizio:** Definire una classe **Persona** con gli attributi nome, cognome, età, email. Definire inoltre il costruttore di default, un costruttore completo, e un costruttore di copia

```
class Persona {  
    private:  
        string nome;  
        string cognome;  
        int eta=0; //valore di default che posso specificare in caso di chiamata del costruttore di default  
        string email;  
        int* temperatura;  
    public:  
        Persona(){ //costruttore di default  
            temperatura=new int[100];  
        }  
  
        Persona(string n, string c, int e, string mail){ //costruttore "completo"  
            nome=n;  
            cognome=c;  
            eta=e;  
            email=mail;  
            temperatura=new int[100];  
        }  
  
        Persona(const Persona &p){ //costruttore di copia  
            nome=p.nome;  
            cognome=p.cognome;  
            eta=p.eta;  
            email=p.email;  
            temperatura=new int[100]; //vero scopo di questo costruttore  
        }  
        void print(){ //funzione inline. piu veloce a tempo di esecuzione  
            cout << "Mi chiamo " << nome << " " << cognome << ", ho " << eta << " anni e la mia email e' " << email << endl;  
        }  
  
        void print2(); // funzione implementata esternamente, quindi non inline.  
                      //si effettua una call e una return  
  
        ~Persona(){ //distruttore  
            delete[] temperatura; //la delete si fa perché si tratta di allocazione  
                                //dinamica. In caso di statica non era necessario  
                                //visto che il compilatore  
//dealloca automaticamente la memoria statica  
    };  
  
    void Persona::print2(){  
        cout << "Mi chiamo " << nome << " " << cognome << ", ho " << eta << " anni e la mia email e' " << email << endl;  
    }  
}
```

Inline e non inline

La funzione membro `print()` è una funzione **INLINE** quindi, ad ogni istanza di un oggetto, si avrà sempre una “copia” all’interno di ogni istanza dell’oggetto. In caso di **definizione “esterna”**, come nel caso di `print2()` ho una funzione **NON INLINE**.

La **INLINE** è **più veloce** a tempo di esecuzione rispetto alla **NON INLINE**.

→ In funzioni molto complicate conviene farle *non inline*.

Nella **non inline** viene effettuata una **call** seguita da un **return** visto che il pezzo di codice è scritto in una cella di memoria diversa rispetto alla funzione chiamante.

- La `delete` va fatta quando si ha allocazione dinamica visto che, in caso di allocazione statica, il compilatore provvede automaticamente a deallocare la memoria statica e **non ha controllo sullo heap**.

`ifndef PERSONA_H` serve per evitare di **definire l'header** della classe persona **più di una volta**.

"Se l'header non è stato definito, allora procedi alla sua definizione."

- Per chiamare un oggetto Studente, devo istanziare una classe Persona e poi Studente.
- Per distruggere un oggetto Studente, devo prima distruggere Studente e poi Persona.

MODIFICATORI DI ACCESSO

I modificatori di accesso permettono, alle sottoclassi, di derivare dei metodi/attributi dalle superclassi in maniera differente, basandosi sul modificatore di accesso assegnatoli, che possono essere "**private**", "**protected**" e "**public**". Il **funzionamento** di essi è così definito:

class B: **public** A{}; → quello che è:

- *private* in A, in B non ci sarà
- *protected* in A, in B sarà protected
- *public* in A, in B sarà public

class B : **protected** A{}; → quello che è:

- *private* in A, in B non ci sarà
- *protected* in A, in B sarà protected
- *public* in A, in B sarà protected

class B : **private** A{}; → quello che è:

- *private* in A, in B non ci sarà
- *protected* in A, in B sarà private
- *public* in A, in B sarà private

COSTRUTTORE DI COPIA

La sintassi del costruttore di copia è la seguente:

```
Persona(const Persona& p){  
    this->nome=p.nome;  
    this->cognome=p.cognome;  
}
```

Il **costruttore di copia** non è per forza necessario, a meno che non si debba implementare "qualcosa di particolare", come dei puntatori (`int* temperatura`)

Si può anche *non implementare* perchè C++ lo fa *in automatico* "**membro a membro**".

C++ fa una **shallow copy** se non viene definito il costruttore di copia, ovvero si effettua una "**copia poco profonda**". Gli attributi di tipo int, double o comunque primitivi, ma anche gli oggetti che possono essere all'interno degli attributi della classe, vengono copiati.

→ Non vengono copiati i puntatori e per tale motivo bisogna scrivere esplicitamente come tali puntatori verranno copiati!!

```
// ESEMPI RICHIAMO COSTRUTTORE DI COPIA  
Studente s3 = s2; // richiama il costruttore di copia  
Studente s3(s2) //richiama il costruttore di copia
```

Static

Per capirne il funzionamento, vediamo un esempio di utilizzo:

```
void fun(){  
    int a=10;
```

```

}
int main(){
    fun();
    a=45; //genera un errore di scope
}

```

→ La variabile “**a**” è visibile **solo all'interno** della funzione “**fun()**”.

```

void fun(){
    static int a=10; //ha visibilità globale e si può usare fuori dalle funzioni
    ...
    a++;
}

int main(){
    fun();
    a=20; //si può fare
}

```

“**a**” adesso ha **scope globale**, quindi fuori dalla funzione “**fun()**”

→ Differenza fra **int a =10;**(definizione *globale*) e **static int a=10;**

1. L'inizializzazione di “**a**” nella funzione **fun()** **viene fatta solo una volta**. Solitamente le variabili static all'interno delle funzioni sono usate come contatori per vedere quante volte viene usata una funzione.

```

// STATIC -> a=0;a=1;a=2; -> fino a quante volte viene chiamata la funzione
fun(){
    static int a=0;
    a++;
}

//NON STATIC a=0;a=0;a=0; -> fino a quante volte viene chiamata la funzione
fun(){
    int a=0;
    a++;
}

```

Static e oggetti

```

class A{
    int a=0;
}

```

Per ogni istanza di A, “**a**” verrà inizializzata sempre a 0. Sarà una **variabile di istanza**. La sua allocazione avviene ad ogni nuova istanza della classe.

```

class A{
    int a=0; //variabile di istanza
    public:
        static int b; //variabile di classe
};

int A::b=0; //inizializzazione obbligatoria fuori dallo scope della classe

```

L'allocazione di “**b**” avviene solo **una volta** ovvero alla prima istanza della classe. La variabile “**b**” viene **condivisa a tutte le istanze della classe**. Se una istanza modifica “**b**”, tutte le istanze della classe subiranno tale modifica. La variabile di classe deve essere inizializzata fuori dallo scope della classe, quindi **globalmente**.

Di conseguenza, **l'indirizzo della variabile statica** è lo stesso visto che si tratta della **variabile di classe**. Invece l'indirizzo **della variabile di istanza cambierà sempre**.

Const

```

const int a=1002; //a è immutabile
int main(){

```

```
//a=10; //errore
int b=10;
int *c=&b;

// b=10 e c=10;
}
```

```
const int a=1002; //a è immutabile
int main(){
    //a=10; //errore
    int b=10;
    const int *c=&b;
    // b=10 e c=10;
    int d=20;
    c=&d;
}
```

Const con le classi

```
class Persona(){
    string nome;
    string cognome;
public:
    Persona(string n, string c) : nome(n), cognome(c){}
    string getName() const{ //garantisce che gli attributi non verranno modificati
        return this -> nome;
    }
};

int main(){
    Persona p("nome1", "cognome1");
    const string x = p.getName();
    x="marco"; //errore perchè è const
    cout << x << endl;
}
```

Parentesi su alcuni argomenti di Programmazione I da ripassare bene

- **Puntatori costanti** ad un valore variabile;
- **Puntatori a valori costanti**;
- **Puntatori costanti a valori costanti**;
- funzioni → **const void foo()**;
- metodi **void foo() const {}** // il valore di ritorno non si può modificare, ma solo se oggetti → **const class obj** // deve essere inizializzato subito e non si possono usare metodi non const;
- parametri **const**.

Esercizio: scrivere degli esempi per ognuno degli argomenti

```
//puntatore costante ad una variabile
int a=0;
int* const b=&a;
int c=2;
b=&c; //errore

//puntatore a valori costanti
int a=0;
const int* b=&a;
*b=20; //errore

//puntatori costanti a costanti
int a=0;
const int* const b=&a;
int c=34;
*b=10; //errore
b=&c; //errore
//non posso modificare né il valore puntato e nemmeno l'indirizzo del puntatore

//parametri const e tipo di ritorno const quindi la variabile che riceve il return sarà const
const void fun(const int a){
    a=22; //errore
    return a*2;
}
```

```

int main(){
    const int y = fun(b);
    y=12; //errore
}

```

EREDITARIETÀ

→ Abbiamo visto i modificatori di accesso **public**, **private** e **protected**.

```

class A{
    private:
        .
    protected:
        .
    public:
        .
};

```

Esempi di ereditarietà e accesso alle variabili

```

class Base{
    public:
        int n;
};

class Derivata1: protected Base{ //deriva n come protected
    public:
        Base::n; //modiflico la visibilità di n in questa classe e la rendo pubblica
};

class Derivata2 : protected Base{
    //. .
};

int main(){
    Derivata1 d1;
    d1.n = 0; //CORRETTO perchè ho scritto Base::n;
    Derivata2 d2;
    d2.n = 0; //errore perchè n è derivato come protected
}

```

Ereditarietà multipla

```

#include <iostream>

using namespace std;

class A{
    public:
        A(){cout << "costruttore A" << endl;}
        void foo(){cout << "foo di A" << endl;}
};

class B{
    public:
        B(){cout << "costruttore B" << endl;}
        void foo() {cout << "foo di B" << endl;}
};

class C : public A, public B{
    public:
        C(){cout << "costruttore C" << endl;}
};

int main(){
    C c; //OUT. Costruttore A Costruttore B Costruttore C
    c.A::foo(); //usa foo di A
    c.B::foo(); //usa foo di B
}

```

→ *Che succede quando istanzio oggetti di tipo C?*

Se i costruttori sono **privati**, gli oggetti tipo C **non possono venire istanziati**.

In caso di ereditarietà multipla, vengono richiamati i costruttori delle classi basi nell'ordine in cui sono stati dichiarati.

POLIMORFISMO E BINDING

“Binding” è l’associazione tra la chiamata ad una funzione e il codice che la implementa.

1. Quando il codice viene compilato, nella memoria, la lista di istruzioni viene tradotta in codice macchina. Ogni istruzione si troverà ad una determinata cella di memoria. E alcune di queste istruzioni saranno delle funzioni, tipo `foo()` e si troverà ad un determinato indirizzo.
2. Quando si esegue il codice, tutte le istruzioni verranno caricate sullo **Stack**, tra cui `foo()` e quindi il suo indirizzo.

Si distinguono due tipi di binding:

- Binding **Statico** = risolto a tempo di compilazione.
- Binding **Dinamico** = risolto a run-time, ovvero quando il codice si sta eseguendo.

```
class A{
public:
    A(){ cout << "costruttore di A" << endl;}
    void stampa(){
        cout << "sono la classe A" << endl;
    }
};

class B : public A{
public:
    B(){cout << "costruttore di B" << endl;}
    void stampa(){
        cout << "sono la classe B" << endl;
    }
};

int main(){
    A a;
    B b;
    a.stamp();
    b.stamp();
    cout << endl;
    // B c;
    // c.stamp();

    //uso le reference adesso

    A& ra=a;
    A& rb=b;

    ra.stamp();
    rb.stamp(); //out: la stampa di classe A

    //uso i puntatori puntatori ora
    cout << endl;

    A *pa=&a;
    A *pb=&b;

    pa-> stamp();
    pb -> stamp(); //viene chiamata sempre stampa di A perchè non è detto
                    // da nessuna parte che stamp() è riscritta in B
}
```

Nel pezzo di codice sopra riportato, sia che vengono usati le reference o i puntatori, quando si richiama la funzione `stamp()` verrà richiamata sempre la funzione presente nella **classe A** (classe base) perchè il compilatore non sa, in questo caso, che **tale funzione è stata ridefinita in classi derivate**.

→ In questo caso si risolve facendo **Overriding** e viene fatto inserendo nella funzione della **classe base A** la keyword “**virtual**” prima della funzione stessa, quindi avendo la funzione così definita:

```
virtual stampa(){cout << "sono la classe A" << endl;}
```

La funzione virtual viene risolta a **run-time** e quindi vengono richiamati i metodi della classe derivata quando si usano puntatori e references.

*Se definisco una funzione virtual privata in classe Base posso richiamarla usando vari metodi all'interno della classe stessa, come avrei fatto con un **getter**.*

CLASSI ASTRATTE

```
#include <iostream>

using namespace std;

class Base{
public:
    int n=0;
};

class Der1 :virtual public Base{

}; //virtual scritto prima di "public"..
class Der2 : public virtual Base{

}; // ..oppure dopo "public" è UGUALE
class Der3 : public Base{

};

class Multipla : public Der1, Der2, Der3 {
public:
    Multipla(){
        Der1::n=1; //Stesso n di Der2 (con virtual)
        Der2::n=2; //Stesso n di Der1 (con virtual)
        Der3::n=3; //non virtual, "n" è a indipendente

        cout << Der1::n << Der2::n << Der3::n << endl; //senza i virtual stampa 123
                                                //con i virtual stampa 223
    }
};

int main(){
    Multipla m;
}
```

Gli attributi **ereditati** in maniera **virtual saranno condivisi**, quindi se modifico un attributo in classi diverse, l'attributo cambiato è uguale.

La variabile "n" quindi è **condivisa da Der1 e Der2**.

Astrazione delle classi

```
#include <iostream>

using namespace std;

class A {
public:
    A() { cout << "costruttore di A" << endl; }
    virtual void stampa() = 0;
};

class B : virtual public A{
public:
    B() { cout << "costruttore di B" << endl; }
    void stampa() { cout << "sono la classe B" << endl; }
};

class C : virtual public A{
public:
    C() { cout << "costruttore di C" << endl; }
    void stampa() { cout << "sono la classe C" << endl; }
};

class D : public B, public C {
public:
    D() { cout << "costruttore di D" << endl; }
};

int main() {
    Multipla m;

    D d;
}
```

Le classi astratte si possono **istanziare come puntatori alla classe stessa** per creare **array o liste** che condividono dati all'interno delle classi base dalle quali derivano.

Esempio:

```
A* vett;  
vett= new B(...)[20]; //array di 20 elementi contenenti oggetti B che ereditano da A
```

```
class A { //classe astratta  
public:  
    A(){} //costruttore di A  
    virtual void foo() =0; // funzione virtuale pura  
};
```

TEMPLATE

Il template è un **tipo generico di dato** che può essere usato all'interno di una funzione o all'interno di una classe e può diventare un qualsiasi tipo di dato.

```
template<typename T>  
void swap(T var1, T var2){  
    cout << "swap " << var1 << "and" << var2 << endl;  
    T temp = var1;  
    var1=var2;  
    var2= temp;  
}
```

Si possono definire anche più tipi di dato generici seguendo questa sintassi:

```
template <class T1, class T2> // "class" oppure "typename" è IDENTICO
```

```
template <typename T> //creazione di classe template  
class A{  
public:  
    T attributo;  
}
```

Uguaglianza fra istanze

Esercizio: scrivere una classe template che prenda due tipi di dato generici e li utilizzi come tipi di dato per due attributi della classe stessa.

scrivere metodi set e get e un metodo di stampa, oltre al costruttore.

```
#include <iostream>  
  
using namespace std;  
  
template<typename T1, typename T2>  
  
class A{  
private:  
    T1 dato1;  
    T2 dato2;  
public:  
    A(T1 n, T2 m) : dato1(n),dato2(m){  
        // cout << "A constructor" << endl;  
    }  
  
    void stampa(){  
        cout << "Dato1 = " << dato1 << " Dato2 = " << dato2 << endl;  
    }  
  
    T1 getDato1(){return dato1;}  
    T2 getDato2(){return dato2;}  
  
    void setDato1(T1 n){dato1=n;}  
    void setDato2(T2 n){dato2=n;}  
};  
  
int main(){  
    A<int,double> a(5,10.2);  
    A<string,double> b("ciao",54.4);  
  
    // cout << a.getDato1() << " " << a.getDato2() << endl;  
    cout << "Dati primo oggetto: ";
```

```

a.stampa();
cout << "Dati secondo oggetto: ";
b.stampa();

cout << endl << "Uso i setter su primo oggetto" << endl;
a.setdato1(10); a.setdato2(5.5);
a.stampa();

cout << endl << "Uso i setter su secondo oggetto" << endl;
b.setdato1("pippo"); b.setdato2(79.12);
b.stampa();
// cout << a.getdato1() << " " << a.getdato2()<< endl;
}

```

FRIEND

La keyword `friend` fornisce, ad una funzione o ad una classe, **accesso ai membri private e protected** della classe in cui essa appare.

Le funzioni friend **non sono considerate metodi** quindi, quando sono richiamate mediante un'implementazione esterna oppure dalla funzione main, non serve l'utilizzo del *risolutore di scope* `A::foo()`.

```

class A{
private:
    int n;
public:
    friend void foo(A& a);
};

void foo(A& a){ //non serve void A::foo(A& a)
    a.n=3; //modifico la variabile privata di A grazie alla friendship
}

```

Friend a classe

La “friendship” va applicata **esclusivamente in un senso** e non vale il viceversa. Per esempio:

- se class A è friend di class X allora A può accedere ai private di X ma è soltanto così e non viceversa. Quindi X non può accedere ai private di A. Per esempio:

```

class Y{};

class A{
    int data;
    class B{}; //scope di B è limitato all'interno della classe A.
               //Un oggetto B fuori da A non può essere creato
    enum {a = 100};
    friend class X;
    friend Y;
};

class X : A::B { //X eredita da B in maniera pubblica, lo posso fare solo perché è friend
    A::B mx;
    class Y{
        A::B my;
    };
    int v[A::a];
}

```

enumeration (`enum`)= collezione di variabili alle quali assegno dei valori. Definisce una serie di campi appartenente alla classe di dichiarazione.

```

class A{
    enum {
        a=100;
        b=15;
    }
};

```

Posso accedere ad `A::a` oppure a `A::b` come se fossero delle variabili.

Inoltre `enum` è **costante** e non si possono modificare le variabili al suo interno, ovvero aggiungere, rimuovere o modificare valori delle variabili.

STRUTTURE DATI

Le **strutture dati** sono delle **entità astratte** che servono per **immagazzinare dati**.

Si possono fare delle operazioni di **inserimento**, **ricerca** e **cancellazione**.

Un esempio di strutture dati sono gli **array** che permettono di immagazzinare **elementi dello stesso tipo** in una stessa variabile ed esiste in tutti i linguaggi di programmazione (con diverse proprietà).

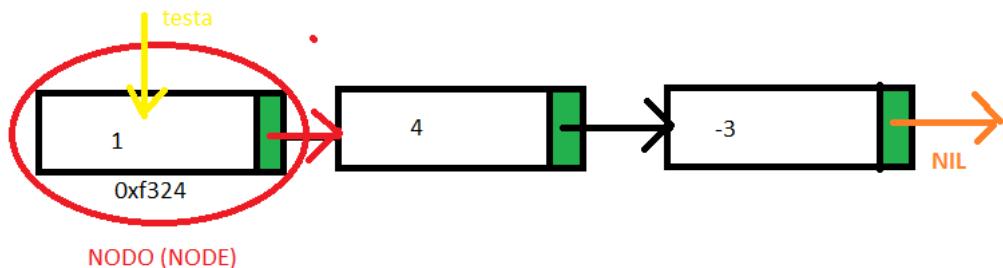
- **Limitazione** array? La **dimensione massima prefissata**;
- **Vantaggi** array? **Accessibilità** ad ogni elemento in **tempo costante** con l'operatore di indicizzazione (parentesi quadre)
→ $\text{Array}[2]$. {complessità $O(1)$ }

La prima struttura dati che vedremo è la "**lista**".

Lista

Una **lista** è una **sequenza di elementi collegati l'uno al proprio successivo** ("si danno la mano").

→ **Differenza rispetto agli array?** Una lista **non è preallocata** (statica o dinamica) ma viene popolata man mano che gli elementi vengono inseriti al suo interno, quindi la sua **dimensione è variabile** (dimensione non prefissata → non nota a priori).



Una lista è formata da **NODI** che sono composti da:

- un **valore**
- un **puntatore all'elemento successivo(link)**

→ Come si accede alla lista? Come si prendo gli elementi se volessi "scorrere"?

Si usa la parola "**testa**" (HEAD) ed è un **puntatore al primo elemento(nodo) della lista**.

Ogni nodo ha un proprio indirizzo. All'ultimo nodo avrà un **nullpointer** perchè indica la fine della lista, e viene indicato con **NIL**.

Ogni elemento è collegato al successivo. Le liste che hanno questa caratteristica vengono chiamate **LISTE LINKATE SEMPLICI**.

Liste linkate semplici

Per definire una lista serve:

1. La **testa Head** che è un puntatore di tipo → `Node<T>* head;` ;

Per definire il nodo, invece, servono:

- un **tipo di dato** che rappresenterà il **valore** → `T val;;`
- un **puntatore al nodo successivo** → `Node<T> *next;` .

→ Come si implementano?

→ Scrivere una classe template(chiamata "List") che implementi la lista.

→ Scrivere una classe template (chiamata "Node") che implementi il nodo

Seguire questi dati:

1. List::: —>Node <T>;

2. Node:: —>val: T; —> next: Node<T>*;

```
#ifndef LNODE_H //L sta per Linked
#define LNODE_H

#include <iostream>

using namespace std;

template <typename T>
class Node {
private:
    T val;
    Node<T>* next;
public:
    Node(T val) : val(val){
        next=nullptr;
    }
};

#endif
```

```
#ifndef LIST_H
#define LIST_H

#include <iostream>

using namespace std;

template<typename T>
class List{
private:
    Node<T>* head;
public:
    List(){
        head=nullptr; //inizializzo head per evitare di avere puntatori "spuri"
    }
};

#endif

/*Nel costruttore,"head" va inizializzato a nullptr sennò è un puntatore "spurio".
Per questo, nella stampa, se non avviene questo assegnamento iniziale,
una lista vuota avrà head come puntatore a qualcosa di non definito.*/
```

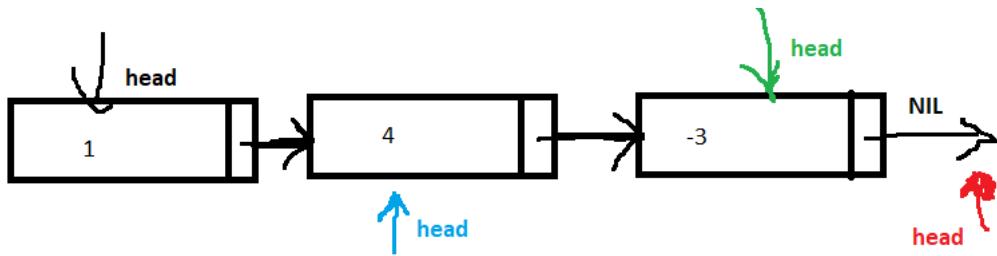
Operazioni su liste

Le **operazioni** che si possono fare sulle liste sono varie. Alcune di esse sono:

1. Inserimento (all'inizio la lista è vuota);
2. Accesso;
3. Ricerca;
4. Cancellazione;
5. Ordinamento;
6. Copia;
7. Controllo se la lista è vuota;

Di queste operazioni vedremo il **controllo della lista vuota**, **inserimento**, **cancellazione** e **ricerca**.

Controllo lista vuota



Vedendo il disegno, posso fare alcune considerazioni riguardo ad esso:

- Lista con **head** è *non vuota*;
- Lista con **head** è *non vuota*;
- Lista con **head** è *non vuota*;
- Lista con **head** è **vuota** perchè head è nullptr.

Come controllo? Per vedere se la lista è vuota, basta verificare che **Head sia nullptr**:

```
bool isEmpty(){ //controllo se la lista è vuota oppure no
    return head == nullptr; //se l'uguaglianza è true, bool di return è true
}
```

Inserimento in testa di un valore “b”



- In caso di **lista vuota**, quindi head che punta a nullptr (head=nullptr) devo creare un nodo e impostare la testa a questo nodo creato. Creo il nodo e assegno la testa a questo nodo.

```
void insert(T val){
    if(this->isEmpty()){
        head = new Node<T>(val); //creo nuovo nodo e assegno il valore "val"
    }
}
```

- In caso di **lista non vuota** posso inserire un elemento in vari modi. Allora si parla di **inserimento dell'elemento all'inizio** della lista, **alla fine** della lista oppure **inserimento dell'elemento maniera ordinata**.

In caso di inserimento all'inizio della lista si deve aggiornare Head e farlo puntare al nodo che contiene il nuovo valore considerato "b" (prima puntava al nodo che aveva "a").

I **passaggi da seguire** sono i seguenti:

1. **creare nuovo nodo** contenente il valore "b" che inizialmente punterà a NIL (di default)
2. Il **successivo del nuovo nodo** deve essere **la testa della vecchia lista**;
3. **aggiornare la testa**;

INSERIMENTO DI (b)



In pseudo codice scriveremmo:

1. `temp ← Nodo(val)` //creazione nuovo nodo
2. `temp.next ← head` //assegnamento di head al nuovo nodo temp
3. `head ← temp` //la testa punta al nuovo nodo

```
void insertHead(T val){  
    if(this->isEmpty()){  
        this->insert(val);  
        return;  
    }  
    Node<T>* temp = new Node<T>(val);  
    temp->next = head;  
    this->head=temp;  
}
```

Compilando il codice (main) si vede che non si può accedere a “`next`” perchè è stato dichiarato privato.

| Allora come si **accede** a `next`?

Usando il “`friend`”:

```
template<typename U>  
friend class List;
```

Con questa definizione, all'interno della classe “Node”, sto comunicando al compilatore che **in futuro** ci sarà una classe template `List` che **avrà accedere ai campi privati di Node**. (quindi non deve essere necessariamente dichiarata prima).

Si deve definire come classe template e un parametro diverso da T (`T` è usato già per Node)

| Perchè va dichiarata **template**?

Perchè devo specificare al Node che la classe `List` che “arriverà in futuro” sarà di tipo template (questa è proprio la firma/sintassi di friend).

| Come **stampo** la lista e il nodo? Mediante l'overload di `<<`

Per la classe `Node` avrà:

```
friend ostream& operator << (ostream& out, const Node<T> &node){  
    out << "node val " << node.val << "- next= " << node.next; //node.next indirizzo del next  
    return out;  
}
```

Per la classe `List` avrà:

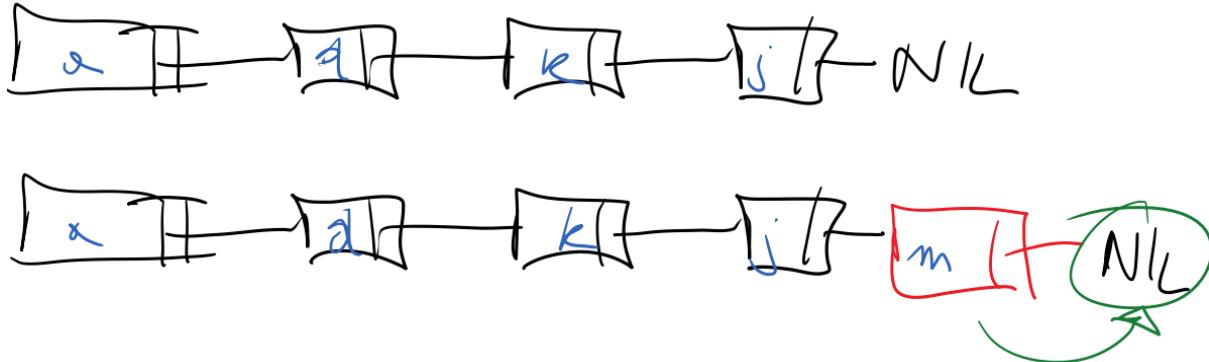
```
friend ostream& operator<< (ostream& out, const List<T> &list){  
    out << "List head=" << list.head << endl;  
    Node<T> *ptr = list.head;  
    while (ptr != nullptr){  
        out << "\t" << *ptr << endl;  
        ptr=ptr->getNext(); //avanzamento di ptr  
    }  
}
```

```

        return out;
    }
}

```

Inserimento in coda



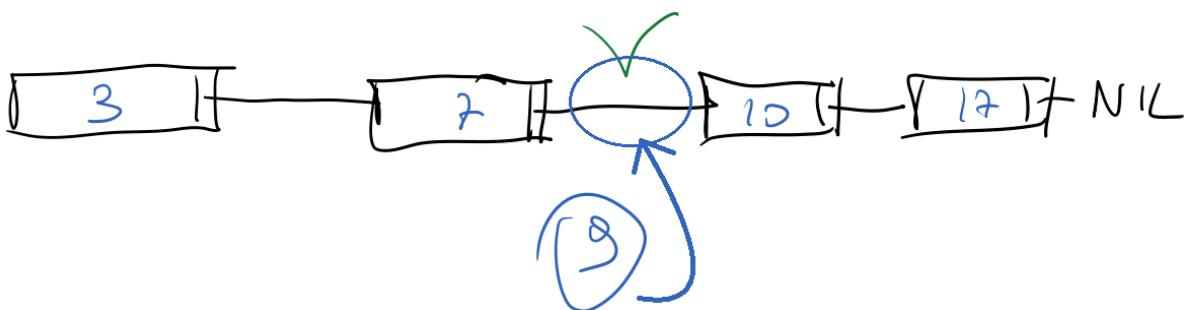
- se la **lista è vuota** posso usare l'inserimento in testa `insertHead(T val)`
- se la **lista non è vuota** devo scorrere la lista fino all'ultimo elemento e inserire l'ultimo elemento, ovvero creare un nuovo nodo e far puntare il vecchio next al nuovo nodo.
- **il puntatore finale** deve essere a `null`, quindi la funzione verrebbe:

```

void insertTail(T val){
    if(this->isEmpty()) { //lista vuota? inserimento in testa
        this->insertHead(val);
        return;
    } //lista non vuota?
    Node<T>* ptr = head; //creo un puntatore e gli assegno head
    while(ptr->getNext() != nullptr){
        ptr = ptr->getNext();
    } //faccio scorrere il puntatore fino alla fine della lista
    Node<T> *temp = new Node<T>(val); //creo il nuovo nodo con valore "val"
    ptr->next=temp; //il next di ptr(punta all'ultimo nodo) è proprio temp
    //temp avrà, di default, il next a null
}

```

Inserimento in modo ordinato



Presumendo che la **lista sia ordinata**, e presumendo che una **lista vuota sia considerata ordinata**, allora:

1. Per prima cosa devo **scorrere** la lista "3 7 10 17" facendo il controllo `(nuovo_valore > next && nuovo_valore ≤ next)` per trovare la posizione corretta. *Nel nostro caso è tra 7 e 10;*
2. Arrivati al controllo che ci interessa, si valuterà *(9 > dell'elemento corrente che sto considerando? Sì. E' minore/uguale del successivo? Sì.)* Allora qui, sulla posizione corrente devo inserire il 9, quindi posso svolgere le mie operazioni. Mi trovo con **il puntatore su 7**. Ho un **nodo successivo che è 10** e un **nuovo nodo che è il 9**;
3. Devo **"staccare"** i collegamenti, sostituire i collegamenti con il nuovo nodo;
4. 9 deve essere successivo di 7 e il suo successivo deve essere il 10.

Procedo in questo modo:

- Il **successivo del nuovo elemento** "9" è il successivo di 7, così garantisco che quello che segue la lista non lo perdo;
- Poi posso scrivere che il **successivo dell'elemento corrente** 7 è il nuovo elemento 9;
- **Non posso fare al contrario** perchè sennò perderei i successivi di 10.

```

void insertInOrder(T val){//inserimento ordinato
    // if(isEmpty() || head->val >= val)
    if(this->isEmpty()){ //se è vuota, inserisco in testa
        this->insertHead(val);
        return;
    }
    if(val <= head->val){ //valore da inserire val <= del valore di testa?
        //allora inserisco in testa
        this->insertHead(val);
        return;
    }

    Node<T>* ptr = head; //puntatore usato per scorrere
    while(ptr->getNext() && (val >= ptr->val && val )) { //scorro la lista finchè il successivo non è nullptr e finchè
        if(val <= ptr->next->val) //val <= del successivo di ptr
            break;
        //altrimenti incremento il ptr e vedo il prossimo nodo
        ptr = ptr->getNext(); //avanzo il puntatore
    }
    if(! (ptr->next) ){
        this->insertTail(val);
        return;
    }

    Node<T>* toInsert= new Node<T>(val); //nuovo nodo da inserire
    toInsert->next = ptr->next; //
    ptr->next = toInsert;

}

```

Compilazione con header presenti in cartelle diverse dal main

`g++ -I ../"percorsoCartella" main.cpp -o main.exe`

i file di #include si possono trovare nella directory corrente, path di sistema(`<iostream>`) oppure nella cartella con percorso **"percorsoCartella"**.

Esercizio: modificare il codice dell'inserimento ordinato `insertInOrder(T val)` in modo che sia possibile inserire sia in ordine ascendente (quello che abbiamo fatto già) che in ordine discendente.

Suggerimento: scrivere due nuovi metodi e selezionare l'inserimento appropriato attraverso un parametro.

Cancellazione di un nodo(casi)

La cancellazione di un nodo può avvenire in diversi modi:

- **I caso:** lista **con un solo nodo**. Il risultato deve essere la **lista vuota**.
- **II caso:** lista con **2 nodi** ($6 \rightarrow 4 \rightarrow \text{NIL}$). `removeHead()` deve essere una lista che ha solo il $4 \rightarrow \text{NIL}$
- **III caso:** lista con **2 nodi** ($6 \rightarrow 4 \rightarrow \text{NIL}$) `removeTail()` deve diventare $6 \rightarrow \text{NIL}$
- **III.2 caso:** lista con **3 nodi** ($6 \rightarrow 4 \rightarrow 3 \rightarrow \text{NIL}$) `removeTail()` deve diventare $6 \rightarrow 4 \rightarrow \text{NIL}$
- **IV caso:** data una lista ($6 \rightarrow 4 \rightarrow 3 \rightarrow \text{NIL}$) `delete(4)` (che elimina un nodo specifico) deve risultare $6 \rightarrow 3 \rightarrow \text{NIL}$

Cancellazione del nodo di testa

Per cancellare il nodo di testa:

1. Devo creare un **puntatore temporaneo** (`temp`) alla testa;
2. **Aggiornare la testa** e facendola puntare all'elemento successivo;
3. **Eliminare il puntatore temporaneo** per fare in modo da perdere il "riferimento"

→ Viene creato un puntatore temporaneo perchè, se aggiornassi la testa assegnandole il next, poi non potrei più eliminare il nodo al quale puntava inizialmente perchè perderei il riferimento ad esso.

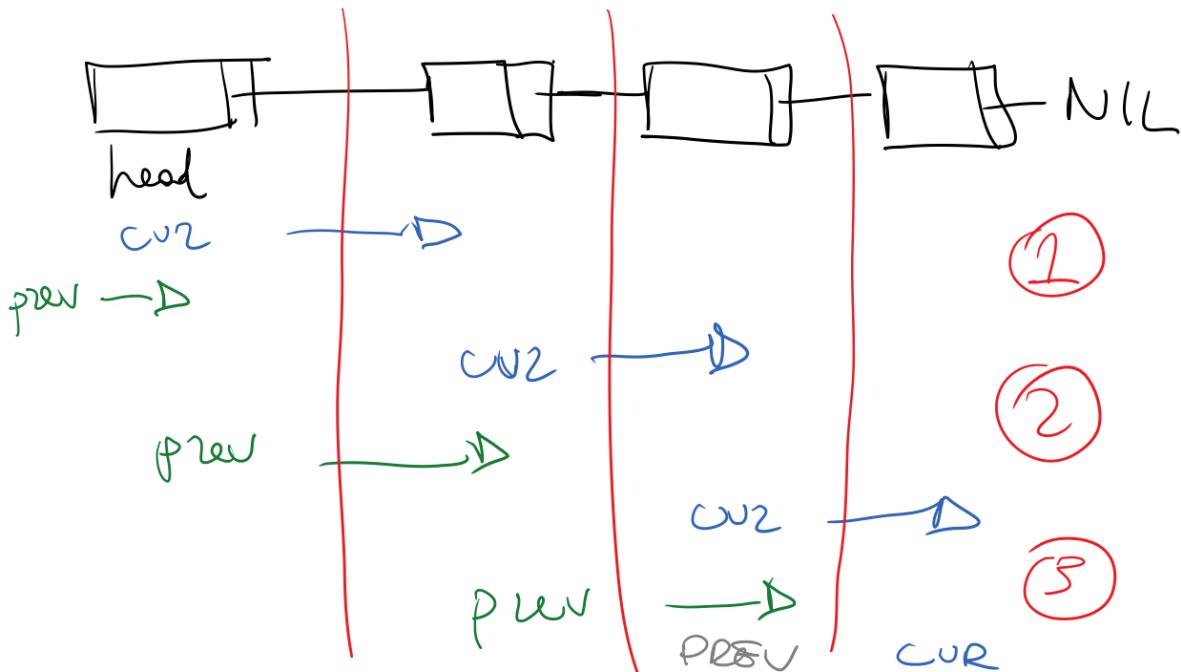
```
void removeHead(){
    if(this->isEmpty()){
        cout << "empty list" << endl;
        return;
    } //altrimenti
    Node<T>* temp=head;
    head=head->next; //faccio scorrere la testa avanti

    delete temp;
}
```

Cancellazione del nodo di coda

In caso di cancellazione del nodo di coda, si devono **creare due nuovi puntatori**:

1. Puntatore **prev**, che sarà il precedente del nodo corrente (inizialmente nullptr);
2. Puntatore **cur**, che punterà all'elemento corrente.
3. Si **scorre la lista**, finchè la condizione (`cur.next!=nullptr`) , cioè il successivo del puntatore corrente è `nullptr` e **aggiornando** il puntatore **cur** e **prev**.
4. Arrivati a fine iterazione, aggiorno il `prev.next` a `nullptr` e così rimuovo la coda.



```
void removeTail(){
    if(this->isEmpty()){
        cout << "empty list" << endl;
        return;
    }

    Node<T>* cur = head; //current pointer
    Node<T>* prev = nullptr; //previous pointer
    while(cur->next->next){
        cur->next; //scorro la lista finchè arrivo alla fine
        prev=cur;
        cur=cur->next;
    }
    prev->next = nullptr; //il successivo del prev è null perchè rimuovo il nodo
}
```

Cancellazione di un nodo con valore specifico

Per la cancellazione di un nodo specifico, si devono usare, anche qui, i puntatori **prev** e **cur**:

1. Per prima cosa posso controllare se il nodo che si deve rimuovere è proprio head. Se è così, allora posso usare il metodo `removeHead();`
2. Altrimenti posso scorrere la lista mediante **prev** e **cur** seguendo le condizione:
`cur.next !=nullptr && cur.val!=val` e aggiornare `prev=cur` e `cur=cur.next;`
3. Se l'elemento da cancellare **non è presente in lista**, allora, a fine iterazione, avrò che cur sarà arrivato alla coda della lista.
Di conseguenza posso controllare se `cur.val!=val`, in tal caso posso stampare che l'elemento non è stato trovato in lista.
4. Altrimenti, dopo aver finito il while, posso **aggiornare** il next di **prev** `prev.next = cur.next;`
5. Di conseguenza **posso eliminare** il puntatore **cur**.

```
void remove(T val){
    if(this->isEmpty()){
        cout << "empty list" << endl;
        return;
    }

    if(head->val == val){ //il nodo da rimuovere è proprio head?
        this->removeHead();
        return;
    }

    Node<T>* cur = head; //current pointer
    Node<T>* prev = nullptr; //previous pointer

    while((cur->next) && (cur->val!=val)){
        prev=cur;
        cur=cur->next;
    }

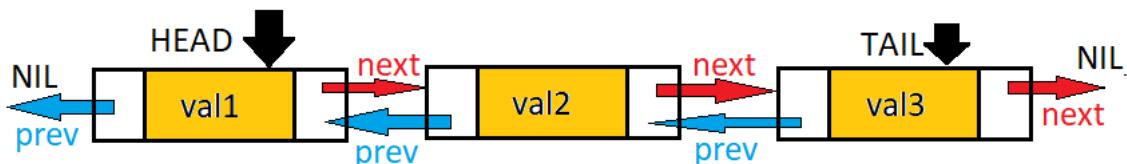
    if(cur->val != val){
        cout << "Element with value " << val << " not found!" << endl;
        return;
    }

    prev->next=cur->next;
    delete cur;
}
```

LISTE DOPPIAMENTE LINKATE (DL)

Differenza fra liste linkate e liste doppiamente linkate (**DL**):

1. **Liste linkate** hanno un **puntatore al nodo successivo**.
2. **Liste doppiamente linkate** hanno **ANCHE** un **puntatore al nodo precedente**, che inizialmente saranno entrambi `nullptr` e avrà anche **un riferimento alla coda** affinché io possa scorrere la lista in modo `head→tail` e `tail→head`.



L'implementazione della lista Doppiamente Linkata è così definita (**DLLList**):

```
template <typename T>
class DLLList{
private:
    DLNode<T> *head; //puntatore alla testa
    DLNode<T> *tail; //puntatore alla coda

public:
    DLLList(){
        head=nullptr;
    }
}
```

```

        tail=nullptr;
    }
}

```

L'implementazione del Nodo della lista Doppiaamente Linkata è così definita (**DListNode**):

```

#include<iostream>
using namespace std;

template <typename T>
class DListNode{
private:
    DListNode<T> *next;
    DListNode<T> *prev;
    T val;
    template <typename U>
    friend class DLList;

public:
    DListNode(T val):val(val),next(nullptr),prev(nullptr){}
}

```

Operazioni su DL List

Sulle liste doppiaamente linkate è possibile eseguire delle operazioni, quali:

1. **Verifica** lista vuota;
2. **Inserimento** (in coda, in testa, in modo ordinato);
3. **Cancellazione**(della testa, della coda, di un nodo specifico);

Verifica DL List vuota

Per verificare che se la lista è effettivamente vuota bisogna semplicemente **controllare** che `head==nullptr` e `tail==nullptr`.

```

bool isEmpty(){
    return (head == tail) && (tail== nullptr);
}

```

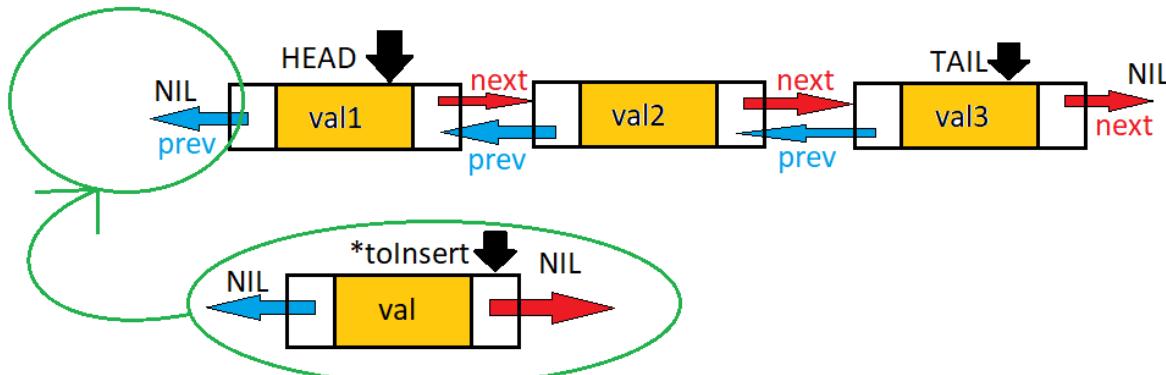
Inserimento in testa di un nodo in una DL List

Per inserire un nodo in testa in una lista doppiaamente linkata bisogna scrivere il codice:

```

void insertHead(T val){
    if(this->isEmpty()){ //verifico se la lista è vuota
        head = new DListNode(val); //creo un nuovo nodo su head
        tail = head; //la lista ha un nodo solo, quindi head=tail
        return;
    }
    DListNode<T> *toInsert = new DListNode<T>(val); //lista non vuota, creo nodo da inserire
    toInsert->next = head; //toInsert.next = head //inserimento prima di head attuale
    head->prev = toInsert; // head.prev = toInsert //il nodo prev di head è toInsert
    head=toInsert; //nuova testa
}

```

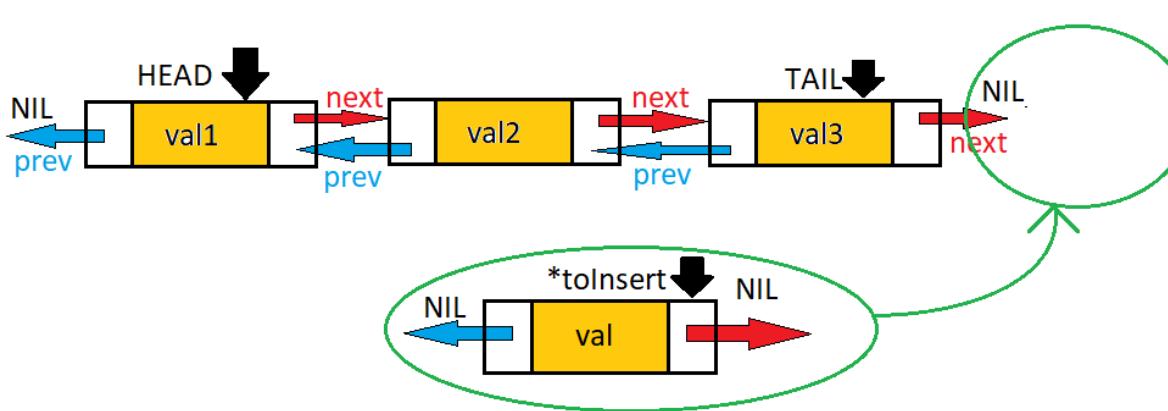


Inserimento in coda di un nodo in una DL List

Per inserire in coda un nodo in una lista doppiamente linkata bisogna seguire il codice:

```
void insertTail(T val){
    if(this->isEmpty()) { //verifico se la lista è vuota
        this->insertHead(val); //se ho lista vuota, inserisco in testa
        return;
    }

    DLNode<T> *toInsert = new DLNode<T>(val); //creo un nuovo nodo da inserire
    toInsert->prev = tail; //prev di toInsert è proprio tail
    tail->next = toInsert; //next di tail è proprio toInsert
    tail = toInsert; //nuova tail = toInsert
}
```



Inserimento di un nodo in modo ordinato in una DL List

```
void insertInOrder(T val){
    if(this->isEmpty()){ //lista vuota? insertHead(val)
        this->insertHead(val);
        return;
    }

    if(head->val >= val){ //ordinamento crescente
        this->insertHead(val);
        return;
    }

    if(tail->val <= val){
        insertTail();
        return;
    }

    //altrimenti scorro la lista
    DLNode<T> *ptr=head;

    while ((val>=ptr->val) && (ptr->next)){
        if(val<= ptr->next->val)
            break;
        ptr=ptr->next;
    }

    DLNode<T> *toInsert = new DLNode<T>(val);
    toInsert->next = ptr->next;
    toInsert->prev=ptr;

    ptr->next->prev = toInsert; //aggiorno questo sennò perdo i successivi nodi
    ptr->next=toInsert;
    // ptr->prev=toInsert; errato
}
```

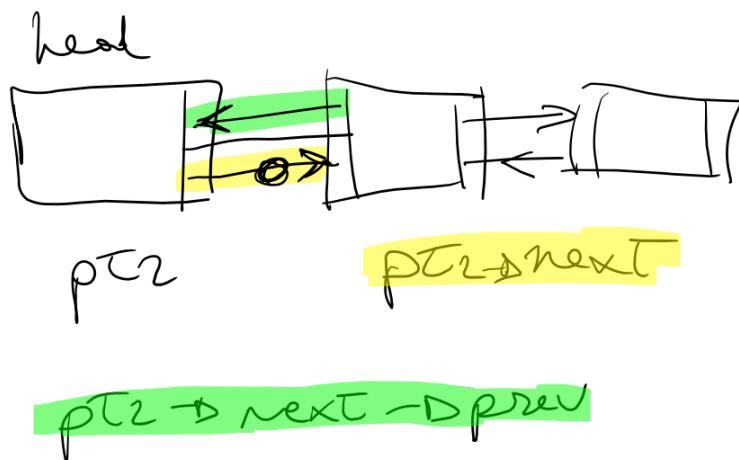
Cancellazione della testa in una DL List

```

void removeHead(){
    if(this->isEmpty()){ //se la lista è vuota, non posso far nulla
        cout << "empty list" << endl;
        return;
    }
    //lista ha un solo nodo?
    if(head==tail){ //head e tail sono uguali
        DLNode<T> *ptr=head; //puntatore alla testa
        head=nullptr;
        tail=nullptr;
        delete ptr; //eliminazione del riferimento rimasto
    }
    //lista con più nodi
    DLNode<T> *ptr=head; //nuovo puntatore alla testa
    ptr->next->prev = nullptr;
    head = ptr->next; //head->next

    delete ptr;
}

```



Cancellazione della coda di una DL List

```

void removeTail(){
    if(this->isEmpty()){ //verifico lista vuota
        cout << "empty list" << endl;
        return;
    }
    //la lista ha un solo nodo?
    if(head==tail){ //se ha un solo nodo head e tail sono uguali
        DLNode<T> *ptr=head; // DLNode<T> *ptr=tail è uguale visto che head==tail
        head=nullptr;
        tail=nullptr;
        delete ptr;
    }
    //la lista ha più nodi
    DLNode<T> *ptr = tail;
    ptr->prev->next=nullptr;
    tail = ptr->prev; //tail=prev

    delete ptr;
}

```

Cancellazione di un nodo specifico in una DL List

```

void remove(T val){
    if(this->isEmpty()){
        cout << "empty list" << endl;
        return;
    }

    if(head->val==val){
        removeHead();
        return;
    }
}

```

```

        if(tail->val==val){
            removeTail();
            return;
        }
        // la lista ha più nodi? scorro la lista

        DLNode<T>* ptr = head;

        while ((ptr->next) && (ptr->val!=val)){
            // ptr->prev = ptr; //non posso assegnare il predecessore al nodo stesso
            ptr=ptr->next;
        }

        if(ptr->val!=val){
            cout << "Element with value " << val << " not found!" << endl;
            return;
        }

        ptr->prev->next = ptr->next;
        ptr->next->prev=ptr->prev;

        delete ptr;
    }
}

```

LA PILA

La **pila** viene chiamata “**Stack**”.

*Una pila si può immaginare come quando **si impilano i piatti lavati accanto al lavabo**. La procedura prevede di lavare i piatti impilati accanto a me. Il **primo che prendo è l'ultimo della “torre di piatti”** che è stato aggiunto a questa impilazione.*

→ Le **operazioni di inserimento e rimozione** in (o da una pila) avvengono sempre dalla cima della pila stessa. Per questo motivo le **pile sono strutture dati di tipo LAST-IN FIRST-OUT (LIFO)**

→ Con le liste non viene considerata la loro dimensione visto che essa è variabile.

Le liste sono una *versione di array* che non hanno problemi con la memoria. Di conseguenza, quello che faccio con le liste lo posso fare con gli array. **L'array ha dimensione prefissata** e invece **la lista ha dimensione variabile**.

- Le operazioni con le pile sono le stesse, a prescindere dall'implementazione.
 - La pila si può implementare come un array, una lista, un mazzo di carte ecc...
Essa rimane concettualmente sempre una pila. Per esempio, una lista, al contrario, non può essere implementata con un array.
- Quindi **l'implementazione** della pila è **UNA DELLE POSSIBILI** implementazioni. (esistono più implementazioni)

Le pile **statiche e dinamiche** hanno delle **proprietà importanti(attributi)**:

1. **dimensione (size)**, ovvero il numero di elementi presenti nella pila;
2. il **TOP** che è un puntatore e rappresenta la **cima della pila**; (che otterremo con il metodo `top()`)
3. un **maxSize** che rappresenta la dimensione massima di una pila limitata.

Le pile hanno dei **metodi** ed essi sono:

1. `push()` che **aggiunge elementi** alla pila;
2. `pop()` che **rimuove un elemento** dalla pila;
3. `isEmpty()` **verifica** se la **pila è vuota**.

Se avessi una **pila limitata** (quindi anche una **masxSize**) e **piena**, e se provassi a fare un `push(19)` non lo potrei fare perché la pila è piena e genererebbe **l'errore Stack Overflow**.

Differenze fra **pila limitata** e **pila non limitata**

Pila **LIMITATA** (o **PILA STATICÀ**) → operazioni e caratteristiche

- `push()` e `pop()`;

- array utilizzabile per l'implementazione data la **dimensione fissata**.

Pila NON LIMITATA (o **PILA DINAMICA**) operazioni e caratteristiche:

- `push()` e `pop()`;
- lista utilizzabile per l'implementazione, data la **dimensione NON fissata**.

Implementazioni pila dinamica

Implementiamo una **pila dinamica** (usando una lista semplicemente linkata).

Considero che ho una lista già implementata:

- voglio implementare `push()`, `pop()`, e `top()`
- mi **servono gli attributi** che mi tengono conto della **size**.

Suppongo di avere una **lista vuota** e voglio fare **push** e **pop**:

- **push**:
 - Creo nuovo nodo. **Top** e **head coincidono** con il nuovo nodo;
 - Se faccio **di nuovo push**, inserisco in testa con `insertHead()`, ovvero in cima alla pila;
 - Quindi **push equivale a un inserimento in testa** e `head == top`;
- **pop**:
 - devo **rimuovere la testa** usando il metodo `removeHead()`.
- **top**:
 - devo semplicemente restituire la **testa della lista**, quindi la **cima** della pila (`return head;`)

→ Per implementare allora **modifico il file list.h**, ma al posto di fare così mi accorgo che posso ereditare dalla classe List.

L'attributo "**head**" della classe List deve essere trasformata da **private** a **protected**.

→ E' utile **osservare**, inoltre, che il **modificatore di accesso** ai membri di List è **protected**, infatti ho:

```
class Stack : protected List<T> {};
```

→ Tutti i **metodi di List non sono utilizzabili dall'esterno della classe** e questo è un modo utile per ereditare List ma senza far modificare la Lista in se visto che *non è nostro interesse farlo al momento*.

```
#ifndef STACK_H
#define STACK_H

#include "list.h"
#include <iostream>

using namespace std;

template<typename T>
class Stack : protected List<T> {

    int size=0;

public:
    Stack() : List<T>(){}
    bool isEmpty(){
        return size ==0;
    }
    Node<T> *top(){
        if(isEmpty()){
            return nullptr;
        }
        return List<T>::getHead();
    }
    void push(T val){
        List<T>::insertHead(val);
        size++;
    }
    Node<T> *pop(){

    }
};
```

```

        if(isEmpty()){
            return nullptr;
        }
        Node<T> *ptr =top(); //prendo un riferimento al nodo in testa
        List<T>::removeHead(); //posso cancellare
        size--;
        return ptr;
    }

    friend ostream& operator<< (ostream& out, Stack<T>& s){
        return out << (List<T>)s; //cast da Stack a List così l'operatore << sia quello della superclasse
    }
};

#endif

```

```

git status
git add "percorsi files"
git status
git commit -m "Dynamic implementation of Stack"
git push

```

Implementazione pila statica

Considerazioni:

1. Logicamente parlando userò un **array** visto che la dimensione è statica;
2. Il **top** sarà l'**indice dell'ultimo elemento inserito** nell'array.
3. Avrò un attributo **size** che terrà conto della **dimensione dell'array** quando si fa **push** e **pop**;
4. Avrò un attributo **maxSize** che terrà conto della **dimensione massima raggiungibile**.

```

#ifndef STATIC_STACK_H
#define STATIC_STACK_H

#include <iostream>
using namespace std;

template<typename T>
class StaticStack{
    T* array;
    int top = -1; //indice dell'elemento top
    int size = 0; //dimensione
    int maxSize=-1; //valgono -1 perché rappresentano la lista vuota perché gli indici nell'array -1 non esistono

public:
    StaticStack(int _maxSize) : maxSize(_maxSize){
        array = new T[maxSize];
    }

    T getTop(){
        if(isEmpty())
            return -1;

        return array[top];
    }

    void push (T val){
        if(top == maxSize-1){
            cout << "Stack Overflow" << endl;
            return;
        }
        array[++top]=val;
    }

    T pop(){
        if(isEmpty())
            return -1;

        return array[top--];
    }

    bool isEmpty(){

```

```

        return top == -1;
    }

    friend ostream& operator<<(ostream& out, StaticStack<T>& s){
        out << "Static Stack: maxSize=" << s.maxSize << endl;
        out << "-----" << endl;
        for(int i=s.top; i>=0; i--){
            out << s.array[i] << " - ";
        }
        return out;
    };
#endif

```

Gestione degli errori per la dimensione della pila

Le **eccezioni** sono oggetti che rappresentano errori nell'esecuzione del programma.

Nella *programmazione ad oggetti* estendiamo il concetto di **eccezioni** nel seguente modo:

1. Se tento di un **estrarre un elemento da una pila vuota**, si deve dire che l'operazione non è permessa;
2. Se tento di **inserire in pila in caso di pila piena**, si deve dire che l'operazione non è permessa visto che la **pila risulta piena**.

Le eccezioni si tratteranno meglio più avanti..

Complessità delle operazioni dello Stack

La **complessità** potrebbe dipendere dalle implementazioni, ma generalmente si hanno 3 operazioni:

1. **push()** → **O(1)** → non serve scorrere la pila;
 2. **pop()** → **O(1)** → serve solo estrarre l'elemento → non si deve scorrere la pila;
 3. **top** → **O(1)** → uguale a prima, cioè *operazioni in tempo costante*.
4. se volessi inserire, quindi push() **in coda** o **estrarre pop() in coda** la complessità passa da **O(1)→O(n)**, con **n=numeri elementi**;

CODA (queue)

La coda viene chiamata **QUEUE**.

L'ordine di accesso alla coda è **First In First Out (FIFO)**, ovvero il **primo che arriva** è il **primo a uscire**.

La **queue** rappresenta una **serie di elementi** uno dopo l'altro che vengono serviti in **ordine di arrivo**.

Gli **attributi** di una coda sono:

- **head**, cioè un puntatore alla testa;
- **tail**, cioè un puntatore alla coda;
- **size**, cioè la dimensione.

Le azioni, quindi i **metodi** che possiamo usare con la coda, sono:

- i nuovi elementi vanno inseriti in coda con **insertTail()**; Si chiama, specificatamente **ENQUEUE**;
- gli elementi da prelevare verranno presi dalla testa, cioè si avrà **removeHead()**; si chiama, specificatamente **DEQUEUE**;
- avremo anche **isEmpty()** che controlla se la coda è vuota o no.

Implementazione dinamica della coda

Per implementare una coda dinamica si userà la **lista doppiamente linkata DLLList**.

Quindi “estendiamo” la classe **DLLList** per implementare la coda, in particolare per implementare i metodi:

1. **enqueue()**, che richiamerà semplicemente **DLLList<T>::insertTail()**;
2. **dequeue**, dove, prima di “prelevare l'elemento” (**removeHead()**), devo **prendere un riferimento** da restituire (il **return**);

3. `isEmpty()`, che lavora sulla `size==0`;

4. `operator<<()`.

- La nuova classe Queue sarà chiaramente *template* per permettere di aggiungere qualsiasi tipo di elemento;
- Per usare la coda doppiamente linkata devo ereditare la classe DLLList con modificatore d'accesso **protected**.

```
template <typename T>
class Queue : protected DLLList<T>{};
```

- Nella classe DLLList, il **modificatore d'accesso private** deve essere **trasformato** in **protected** per poter accedere agli attributi head e tail solo da una sottoclasse;

```
class DLLList{
protected: //era private
    DLNode<T> *head;
    DLNode<T> *tail;
```

Implementazione della Queue

```
#ifndef QUEUE_H
#define QUEUE_H
#include "dllist.h"
#include<iostream>

using namespace std;

template <typename T>
class Queue : protected DLLList<T>{
protected:
    int size=0; //valore di default se non specificato mediante un costruttore
public:
    Queue() : DLLList<T>(){}
    void enqueue(T val){
        DLLList<T>::insertTail(val);
        size++;
    }
    DLNode<T> dequeue(){
        if(isEmpty()){
            return 0;
        }
        DLNode<T> ptr= *(DLLList<T>::head); //prendo un riferimento prima di rimuovere così lo restituisco
        DLLList<T>::removeHead();
        size--;
        return ptr;
    }
    bool isEmpty(){
        return size==0;
    }
    friend ostream& operator<<(ostream& out, Queue<T>& queue){
        out << "Queue starting at " << &(queue.head);
        DLNode<T> *ptr= queue.head;
        while(ptr){
            out << *ptr << endl;
            ptr=ptr->getNext();
        }
        return out;
    }
};

#endif
```

Implementazione statica della coda

Adesso implementiamo la **coda statica**. Si utilizzerà un **array** e, di conseguenza, si avrà una `maxSize`, cioè la **massima dimensione fissata**, oltre la quale non si possono inserire più elementi.

```
#ifndef STATIC_QUEUE_H
#define STATIC_QUEUE_H
```

```

#include<iostream>

#define MAX_SIZE 1000

template<typename T>
class StaticQueue{
private:
    T* array;
    int size=0;
    int maxsize= MAX_SIZE;
    int head=-1;
    int tail=-1;

public:
    StaticQueue(int maxsize = MAX_SIZE) : maxsize(maxsize){
        this->array = new T[maxsize];
    }
    // StaticQueue() : StaticQueue(MAX_SIZE) {}
    // commentato perchè ho messo StaticQueue(int maxsize = MAX_SIZE)

    void enqueue(T val){ // riscritta correttamente più un basso
        if(size==maxsize){
            cout << "queue is full" << endl;
            return;
        }

        if(head==-1) //controllo se la lista è vuota. In tal caso devo aggiornare
                    //l'indice di head che, inizialmente, è fuori array
        head=0;
        array[++tail] = val;
        size++;
    }

    T dequeue(){ // riscritta correttamente più un basso dopo aver valutato
        // i possibili problemi
        if(size==0){
            cout << "queue is empty" << endl;
            return -1;
        }
        return array[head++]; //rimuovo un elemento e la nuova head è
                            //l'elemento successivo
    }
};

#endif

```

Questa coda avrà **alcuni problemi** da tenere in considerazione:

- dopo una serie di `enqueue(val)`, `enqueue(val)`, `enqueue(val)`, `enqueue(val)`, `enqueue(val)`, `dequeue()`, `dequeue()`, `dequeue()`, `dequeue()` → e poi di nuovo `enqueue(val)`, avremo **problemI** con gli **indici di head e tail**. In particolar modo, dobbiamo fare attenzione e usare la **funzione modulo** che abbiamo a disposizione per evitare “**segmentation fault**”, ovvero l’accesso ad una posizione “*non conosciuta*” dal programmatore
- ...quindi le prossime enqueue() hanno **problemI con gli indici**. Non inserisco in `array[tail++]` sennò va fuori array;
- la coda diventa un “**ARRAY CIRCOLARE**”. (queue circolare) e tutto diventa: (`enqueue` e `dequeue` editati)

```

void enqueue(T val) {
    if (size == maxsize) {
        cout << "queue is full" << endl;
        return;
    }

    if (head == -1)
        head = 0;
    tail = (++tail % maxsize);
    array[tail] = val;
    size++;
}

T dequeue() {
    if (size == 0) {
        cout << "queue is empty" << endl;
        return -1;
    }
    T val = array[head];

    head = (++head % maxsize); //uso il modulo per evitare di avere indici fuori array
    size--;
    return val; //rimuovo un elemento e la nuova head è l'elemento successivo
}

```

Si possono usare le code circolari quando, per esempio, si hanno le condivisioni di risorse.

Si osserva che la **coda circolare** si puo fare anche con una **DLLList** in modo tale che `tail` e `head` siano collegate, ovvero `tail->next=head` e `head->prev=tail` e così abbiamo fatto i **collegamenti** per far diventare circolare la **DLLList**.

Per quanto riguarda la **stampa** provvederemo all'implementazione dell'overload dell'operatore di redirezione `<<`. In questo caso si deve stare attenti agli indici dell'array che costituisce la coda perchè, quando gli indici head e tail non rispettano più la condizione `head<tail` (ma diventa `head>tail`) dovremmo usare la **funzione modulo**.

```
friend ostream& operator<<(ostream& out, StaticQueue<T>& queue) {  
  
    if (queue.size == 0) {  
        return out << "queue is empty" << endl;  
    }  
    out << "Static queue - Size " << queue.size << ", maxSize= " << queue.maxSize << endl;  
  
    for (int i = queue.head, count = 0; count < queue.size; count++) {  
        cout << "Queue[" << i << "]=" << queue.array[i] << endl;  
        i = (i+1) % queue.maxSize;  
    }  
    return out;  
}
```

ALBERI BINARI DI RICERCA

Un albero è una **struttura dati**. In natura si tratta di un albero con radici nel terreno, con dei rami e delle foglie. In Informatica è esattamente **al contrario**, ovvero **le radici in alto**, seguite da **rami e foglie**.

E' una **struttura dati gerarchica** perchè si tratta di rapporti **padre → figlio**.

Tutte le foglie sono, in qualche modo, figli della radice.

L'elemento costitutivo dell'albero è proprio il **nodo**. Le sue **PROPRIETA'** sono:

- **ARIETA'**, ovvero il numero di figli possibili;
- **FIGLI**, ovvero un elenco di nodi;
- **PARENT**, ovvero un nodo che deriva dal genitore.

Un nodo i cui figli possono essere 4 figli, si chiama **quaternario**. **N-ario** se un nodo ha **N** figli.

Noi trattiamo gli **alberi binari**, cioè ogni **nodo ha un figlio destro e un figlio sinistro**.

Consideriamo un albero binario: possiamo vedere i nodi che stanno "**allo stesso livello**", cioè stanno alla stessa **profondità** dell'albero. La **radice** ha **profondità 0**, i **figli della radice** hanno **profondità 1** e così via..

La profondità viene chiamata "**LIVELLO DEL NODO**" e il numero di livelli che abbiamo in un albero viene chiamato **ALTEZZA DELL'ALBERO**.

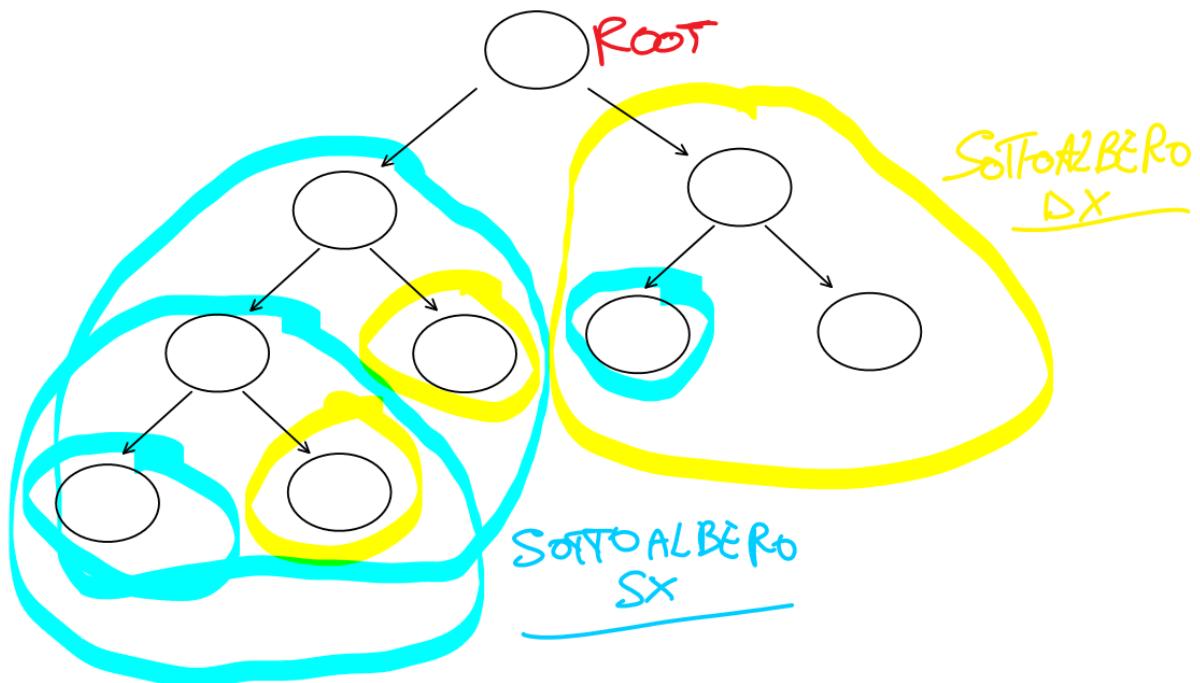
Altra proprietà alberi binari: contare il numero di nodi. A livello 0= 1 nodo. A livello 1=2 nodi . A livello 2=4 nodi. A livello 3=8 nodi. A livello 4=16 nodi. Quindi si ha la potenza del 2.

- Il numero **n** di nodi di un albero è: $n = 2^{\text{altezzaAlbero}} - 1$
- Il numero **n** di nodi è sempre $n \leq 2^{\text{altezzaAlbero}} - 1$
- L'altezza dell'albero **h** è data da $\text{ceil}(\log_2 n) = h$ (il ceil lo uso per un arrotondamento per eccesso)
- Un albero binario è completo se e solo se ha esattamente $2^h - 1$ nodi dove **h** è l'altezza dell'albero.

ALBERO BILANCIATO

E' bilanciato se, per ogni nodo, la differenza dell'altezza fra sottoalbero sinistro e destro è al massimo **1**.

Se ho la radice, il sottoalbero sinistro sono i figli e tutto quello che segue a sinistra e il sottoalbero destro sono i figli nel lato sinistro. Questa definizione si può applicare ricorsivamente ad ogni nodo.

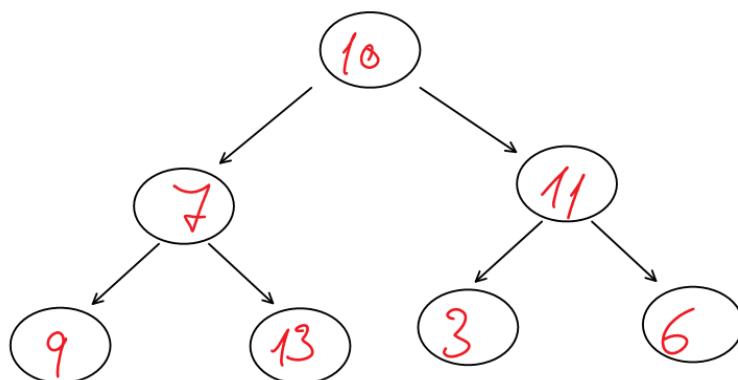


Questo è un albero bilanciato ma non completo.

Il sottoalbero sx ha altezza 3 e il sottoalbero dx ha altezza 2 nei vari nodi. La loro differenza di altezza è 1.

ALBERO BINARIO DI RICERCA

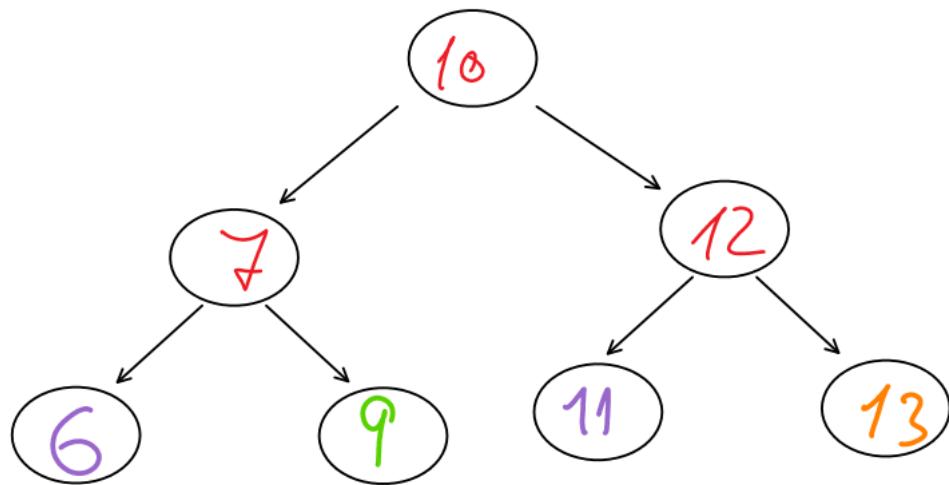
E' un albero binario (BST = Binary Search Tree) per cui tutti i valori messi nel sottoalbero di sinistra della radice sono **minori o uguali** della radice. I valori che si mettono nel sottoalbero di destra della radice sono **maggiori** della radice



Questo è un albero, ma **NON è un BST**.

Si vanno a **sostituire** i valori posizionati nei posti errati che non rispettano la definizione. Non vanno aggiunti nuovi nodi e non si modifica la struttura dell'albero ma vengono solo fatte delle sostituzioni.

Infine l'albero, per essere un BST deve essere così rappresentato:



Nomenclatura recap

1. La **radice** è anche detta **root**;
2. I **nodi che non hanno figli** si chiamano **foglie**!
3. Il numero **n di nodi** di un albero è: $n = 2^{\text{altezzaAlbero}} - 1$
4. Il numero **n di nodi** è sempre $n \leq 2^{\text{altezzaAlbero}} - 1$
5. **L'altezza dell'albero h** è data da $\text{ceil}(\log_2 n) = h$ (il ceil lo uso per un arrotondamento per eccesso)
6. Un albero binario è **completo** se e solo se ha esattamente 2^h nodi dove h è l'altezza dell'albero.
7. **ARIETA'**, ovvero il numero di figli possibili;
8. **FIGLI**, ovvero un elenco di nodi;
9. **PARENT**, ovvero un nodo che deriva dal genitore.

A differenza delle pile, code e liste, negli alberi è necessario avere la possibilità di **cambiare il valore contenuto nel nodo**.

Implementiamo il codice `bst_node.h`

```

#ifndef BST_NODE_H
#define BST_NODE_H

#include <iostream>

using namespace std;

template <typename T>
class BSTNode {
private:
    T key; //valore
    BSTNode<T>* left; //figlio sinistro
    BSTNode<T>* right; //figlio destro

    template <typename U>
    friend class BST; //futura classe dell'albero per poter accedere ai campi privati di questa classe

public:
    BSTNode(T key) : key(key){
        left=nullptr;
        right=nullptr;
    }

    void setLeft(BSTNode<T>* left){
        this->left=left;
    }

    void setRight(BSTNode<T>* right){
        this->right=right;
    }
}
  
```

```

        BSTNode<T>* getLeft(){
            return this->left;
        }

        BSTNode<T>* getRight(){
            return this->right;
        }

        void setKey(T key){
            this->key=key;
        }

        T getKey(){
            return this->key;
        }

        friend ostream& operator<<(ostream& out, BSTNode<T>& node){
            out << "BSTNode@=" << &node << " key=" << node.key << " left=" << node.left << " right=" << node.right;
            return out;
        }

        // //aggiungo il distruttore in piu rispetto alle liste,code ecc..
        // //quando distruggo un nodo, distruggo l'albero quindi non mi interessano più i sottoalberi dx e sx
        // ~BSTNode(){}
        //     delete left;
        //     delete right;
        // }
        //l'ho commentato perchè lo implementeremo in modo diverso perchè i puntatori dovrebbero essere allocati in modo statico quind
};

#endif

```

Implementiamo il codice [BST.h](#)

```

#ifndef BST_H
#define BST_H
#include <iostream>
#include "bst_node.h"

using namespace std;

template<typename T>
class BST{
private:
    BSTNode<T>* root; //radice

public:
    BST(){}
        root=nullptr;
    }

//implementiamo i metodi
bool isEmpty(){
    return root==nullptr; //se la radice è nulla, l'albero è vuoto
}

void insert(T key){
    if(this->isEmpty()){
        root=new BSTNode<T>(key); //creo un nuovo nodo sulla radice
        return;
    }

    insert(root,key);
}

void insert(BSTNode<T>* ptr, T key){
    if(ptr==nullptr){ //caso base
        ptr=new BSTNode<T>(key);
        return;
    }
    else if(key<ptr->key){
        insert(ptr->left,key); //vado a sinistra
    }
    else {
        insert(ptr->right,key);
    }
}

void visit(BSTNode<T>* node){
    cout << *node << endl;
}

```

```

void inorder(BSTNode<T>* ptr){
    if(ptr==nullptr)
        return;

    inorder(ptr->left);
    visit(ptr);
    inorder(ptr->right);
}

void inorder(){
    inorder(root);
}

};

#endif

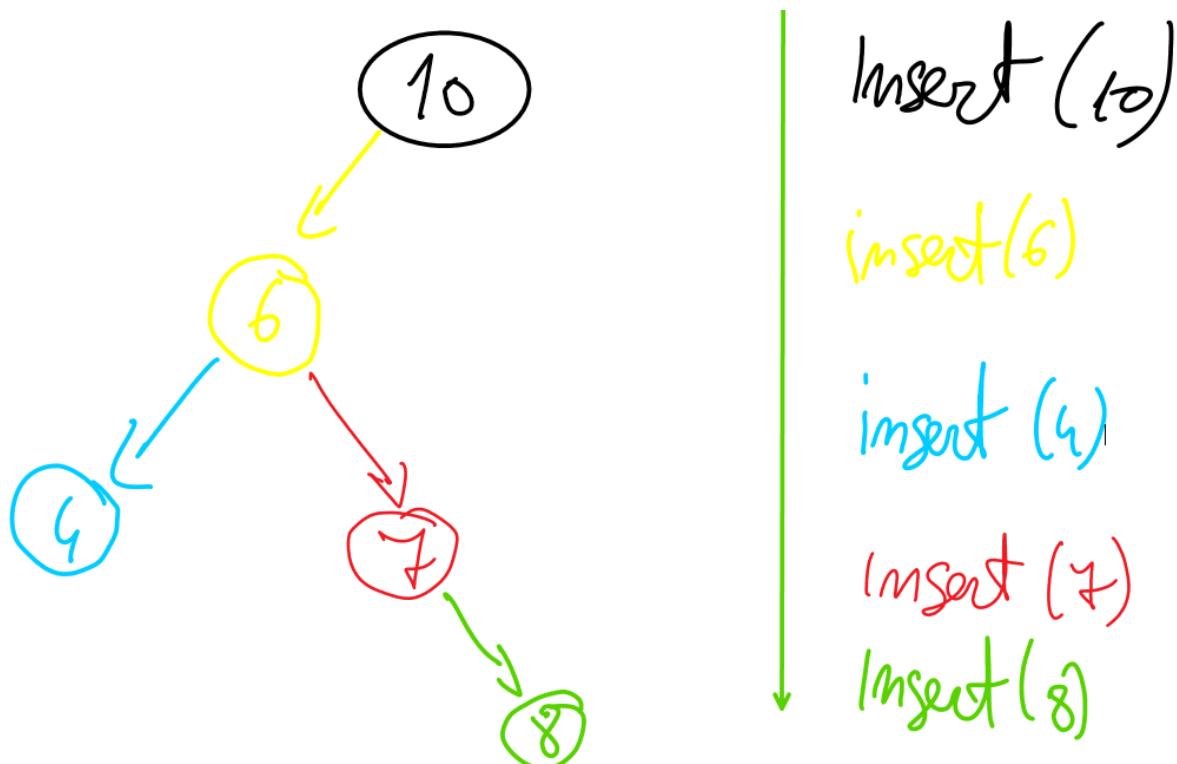
```

Inserimento di un valore

`void insert(T key)`

- All'inizio l'albero è **vuoto** e la radice è il primo nodo da inserire.
Per esempio `insert(10)` → la radice sarà 10.
- se dopo voglio inserire il 6, devo confrontare il 6 con la radice e capire se metterlo a destra o sinistra. In questo caso va inserito a sinistra:
 - In caso di 4, lo metto a sinistra di 6;
 - in caso di 7 lo metto a destra di 7;
 - in caso di 8 lo metto a destra di 7.

Quindi ho una ricorsione.



La procedura ricorsiva deve **avere due parametri**, il valore da inserire e la radice del sottoalbero

La prima chiamata la faccio con la radice. Capisco che vado a sinistra visto che ho un 6. Quindi la prossima chiamata va fatta con `(ptr->left, key)`, altrimenti va fatta con `(ptr->right, key)`

Devo fare questa ricorsione **finchè non trovo nullptr sui puntatori left e right**.

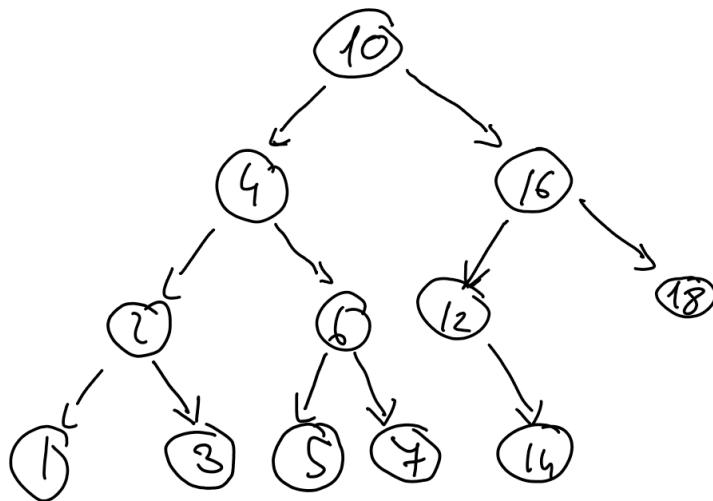
```

void insert(BSTNode<T>* ptr, T key){
    if(ptr==nullptr){ //caso base
        ptr=new BSTNode<T>(key);
        return;
    }

    if(ptr->left==nullptr && key<= ptr->key){
        ptr->left = new BSTNode<T>(key);
        ptr->left->setParent(ptr);
        return;
    } else if (key>ptr->key){
        insert(ptr->left,key);
    } else{
        insert(ptr->right,key);
    }
}

```

Visita di un BST



Come stampiamo un albero di questo tipo? La procedura stampa si chiama **VISITA DI UN ALBERO**.

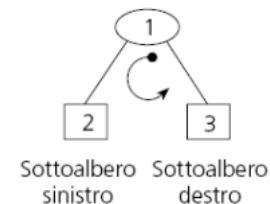
```

void visit(BSTNode<T>* node){
    cout << *node << endl;
}

```

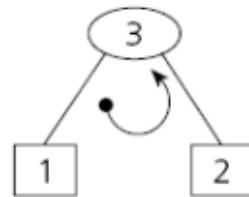
Si fa in 3 modi:

- **VISITA PREORDER**: Visita (stampa) prima la **radice**, poi il **sottoalbero sinistro**(ricorsione) e infine il **sottoalbero destro**(ricorsione). Stampo prima il 10,4,2,1,3,6,5,7,16,12,14,18.
Quindi `root -> left -> right` ricorsivamente



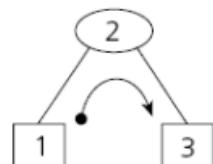
- **VISITA POSTORDER**: Visito ricorsivamente il **sottoalbero sinistro**, poi il **sottoalbero destro**, e infine la **radice**.
Radice → ho un sottoalbero sinistro? sì. Mi sposto a sinistra.. ecc finchè non ho altri sottoalberi sinistri. Di

conseguenza posso stampare 1. Visito il sottoalbero dx e stampo 3. Dopo aver stampato entrambi i sottoalberi della radice 2, posso stampare la radice. Poi stampo 5,7,6,4,14,12,18,16,10
Quindi left → right → root
...Implementabile con una pila...



Sottoalbero Sottoalbero
sinistro destro

- **VISITA INORDER:** Visito il sottoalbero sinistro, poi la radice, e infine il sottoalbero destro.
Vedo radice. Ha un sottoalbero sinistro? sì. Mi sposto su di esso. 1 ha un sottoalbero sinistro? No. Stampo 1.
Stampo la radice 2 e visito il sottoalbero destro. Stampo allora: 1,2,3,4,5,6,7,10,12,14,16,18
Quindi left → root → right



Sottoalbero Sottoalbero
sinistro destro

```

void inorder(BSTNode<T>* ptr){
    if(ptr==nullptr)
        return;

    inorder(ptr->left);
    visit(ptr);
    inorder(ptr->right);
}

void inorder(){
    inorder(root);
}

void inorder(){ // overload di inorder()
    inorder(root);
}
  
```

Modificare il metodo (insert(*, key) e inserire l'overload dell'operatore <<.

Si vuole aggiungere un genitore a BSTNode quindi si aggiunge un attributo al nodo dell'albero chiamato parent. Si modifica il costruttore e metodo insert.

```

template <typename T>
class BSTNode {
private:
    T key; //valore
    BSTNode<T>* left; //figlio sinistro
    BSTNode<T>* right; //figlio destro
    BSTNode<T>* parent; //genitore del nodo
...
...
... //ecc
  
```

Massimo e minimo di un BST

Come trovo il **minimo** di un bst? radice → sinistra → sinistra → sinistra → finchè non trovo nullptr.

Come trovo il **massimo** di un bst? radice → destra → destra → destra → finchè non trovo nullptr

```

BSTNode<T>* min(){
    if(isEmpty()){
        return nullptr;
    }
  
```

```

        BSTNode<T>* ptr = root;
        while(ptr->left){
            ptr=ptr->left;
        }
        return ptr;
    }

    BSTNode<T>* max(){
        if(isEmpty()){
            return nullptr;
        }

        BSTNode<T>* ptr = root;
        while(ptr->right){
            ptr=ptr->right;
        }
        return ptr;
    }
}

```

Successore e predecessore

Il **predecessore** è il nodo con **valore massimo** nel **sottoalbero sinistro** del nodo considerato.

Il **successore** è il nodo con **valore minimo** nel **sottoalbero destro** del nodo considerato.

In caso di `inorder()` i predecessori e successori risaltano subito all'occhio vista la disposizione dei nodi.

Una proprietà di un albero: tutti i nodi di sottoalbero sinistro sono minori del nodo radice e tutti i nodi di un sottoalbero destro sono maggiori del nodo radice.

Di conseguenza si ha una relazione < (ordinamento)

Il predecessore è l'elemento che viene prima nell'ordinamento, invece il successore è l'elemento che viene prima dopo aver ordinato l'albero con il metodo `inorder()`.

E' utile precisare che il successore del nodo con valore massimo (**max**) è **NULL** e il predecessore del nodo con valore minimo (**min**) è **NULL**.

SI PUO VEDERE ESERCIZI PROF ORTIS (anche esercizi)

Scheda docente

Q <http://web.dmi.unict.it/corsi/l-31/docenti/alessandro.ortis>

Definizione formale di successore e predecessore

Dato un nodo x , il successore di x è:

- se **esiste sottoalbero destro** del nodo x , il minimo di questo sottoalbero destro;
- se **non esiste il sottoalbero destro** del nodo x , è l'antenato più prossimo di x il cui figlio sinistro è anche un antenato di x . Cioè , a partire dal nodo preso in considerazione x devo risalire l'albero fino a quando il nodo che ho sopra, genitore, è un figlio destro. Appena il nodo genitore è un figlio sinistro, ho trovato il successore che stavo cercando, cioè il genitore dell'ultimo nodo che prendo in considerazione.

Di conseguenza il **nodo** che sto **considerando** viene chiamato x e il nodo **parent** è chiamato y .

Quindi dato un nodo x che è una foglia, chi è il successore? x è il nodo considerato. y è $\text{parent}(x)$.

1. x è figlio destro di parent ? si. $x = \text{parent}$ e $y = y \rightarrow \text{parent}$ (cioè sono saliti di un livello);
2. x è figlio destro di parent ? si. Allora risalgo e ripeto la procedura.
3. Mi fermo quando x non è più un figlio destro di parent e il successore che sto cercando è proprio y , cioè $\text{parent}(x)$.

```

BSTNode<T>* successor(BSTNode<T>* x){
    if(this->isEmpty()){
        return nullptr;
    }
}

```

```

// I caso: x ha un sottoalbero destro
if(x->right){
    return this->min(x->right);
}

// II caso: h non ha sottoalbero destro
// il successore di x è l'antenato più prossimo di x il cui figlio sinistro è anche antenato di x

BSTNode<T>* y = x->parent;

while(x!=nullptr && x==y->right){
    x=y;
    y=y->parent;
}

return y;
}
//quindi si fa right -> left -> left ... fino a nullptr

```

Dato un nodo x, il suo predecessore è:

- se **esiste un sottoalbero sinistro**, allora il predecessore del nodo x è il massimo del sottoalbero sinistro;
- se **non esiste un sottoalbero sinistro**, allora devo prendere in considerazione il nodo che ho sott'occhio, chiamandolo x e il parent corrispettivo y=parent(x).
In maniera del tutto speculare alla ricerca del successore, il ciclo while lo devo eseguire finchè il nodo che sto considerando x è figlio sinistro di parent.
A fine ciclo avrò y che simboleggerà il predecessore del nodo di partenza del quale cercavo il predecessore, appunto.

```

BSTNode<T>* predecessor(BSTNode<T>* x){
    if(this->isEmpty()){
        return nullptr;
    }

    //I caso: x ha un sottoalbero sinistro
    if(x->left){
        return this->max(x->left);
    }

    //II caso: x non ha un sottoalbero sinistro

    BSTNode<T>* y = x->parent;
    while(x!=nullptr && x==y->left){
        x=y;
        y=y->parent;
    }

    return y;
}
//quindi si fa left -> right -> right fino a nullptr

```

Osservazioni

Come faccio a testare il secondo caso nel main? Devo partire da una foglia. come si fa a prendere in considerazione una foglia? Allora serve un metodo per ricercare un nodo in un albero.

Applicare **ricerca dicotomica** su un albero ordinato data una chiave key.

*La chiave è presente nell'albero? Restituisco il puntatore al nodo sennò **nullptr**.*

```

BSTNode<T>* search(T key){
    return search(root, key);
}

//ricerca di un nodo data una chiave
BSTNode<T>* search(BSTNode<T>* ptr, T key){
    if(ptr==nullptr){
        return nullptr;
    }
    if(ptr->key==key){
        return ptr;
    }

    if(key<=ptr->key){ //dicotomica
        return search(ptr->left, key);
    }
    else{
        return search(ptr->right, key);
    }
}

```

```

        }
        return nullptr;
    }
}

```

Cancellazione

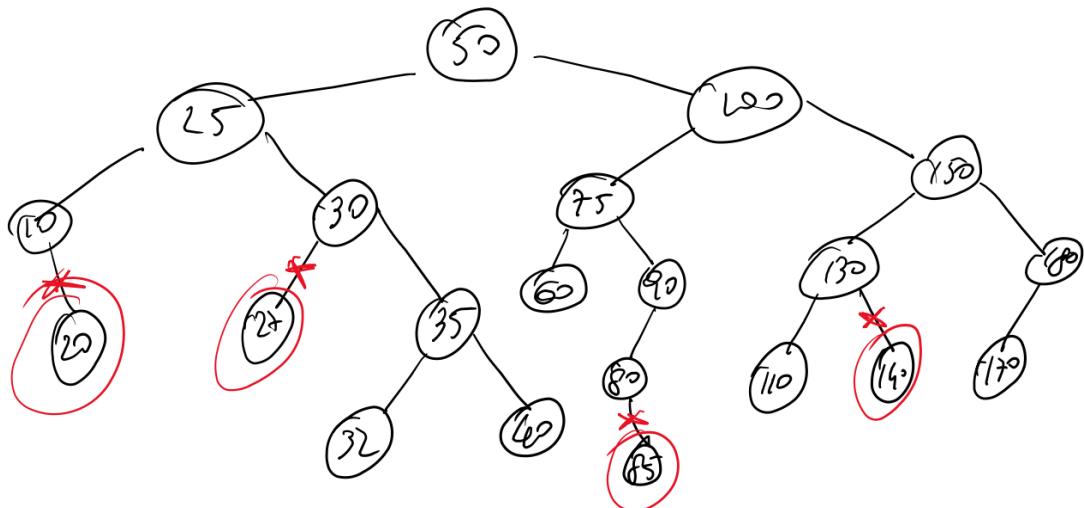
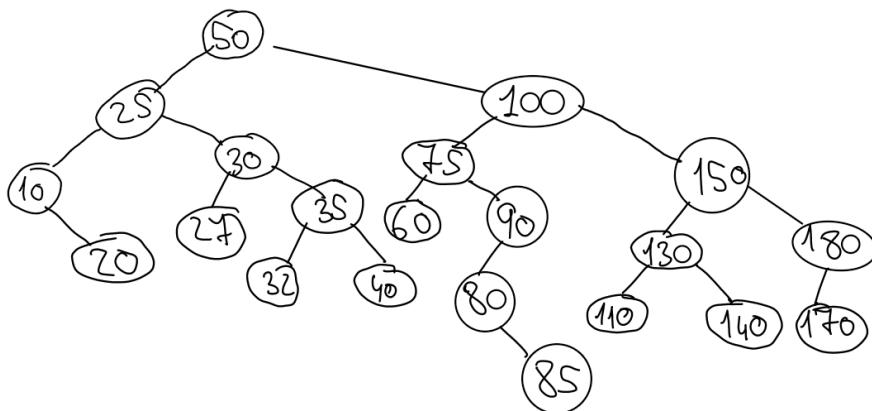
Si verificano 3 casi di cancellazione di un nodo "z" da un albero:

1. la **chiave z** da cancellare è una **foglia**;
2. la **chiave z** da cancellare si trova in un nodo con **un solo figlio**;
3. la **chiave z** da cancellare si trova in un nodo con **due figli**.

Caso 1: z è una foglia

Devo semplicemente rimuovere il riferimento con un ptr temporaneo. e devo valutare se eliminare il figlio sinistro o destro del parent.(in base al nodo da eliminare)

Esempio di albero:



Dal disegno si capisce che il metodo si può applicare ai nodi 20, 27, 85, 140 ecc..(cioè le **foglie**)

In questi casi basta semplicemente:

- rimuovere il collegamento al padre, cioè **eliminare il collegamento parent → foglia**;
- **eliminare** il nodo.

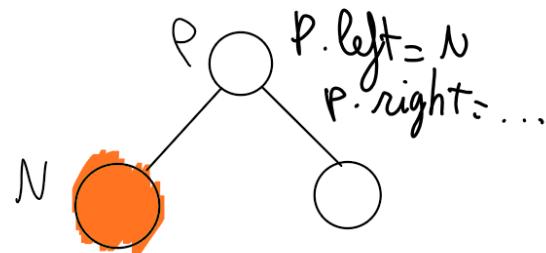
Per **implementare** questo metodo uso un disegno:

1. Il nodo N è quello da rimuovere. Esso ha `left=Nil` e `right=Nil`

2. Il nodo P è il parent di N .

3. Quello che devo fare è:

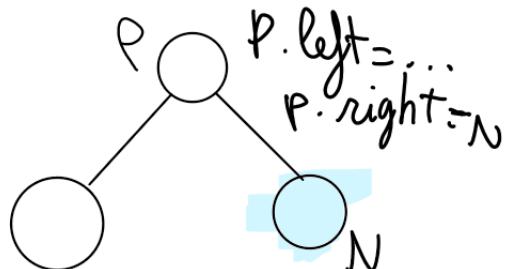
- `node->parent->left=Nil` se N è figlio sinistro



$N.left = Nil$
 $N.right = Nil$
 $N.parent = P$

`node->parent->left=Nil`

- `node->parent->right=Nil` se N è figlio destro



$N.left = Nil$
 $N.right = Nil$
 $N.parent = P$

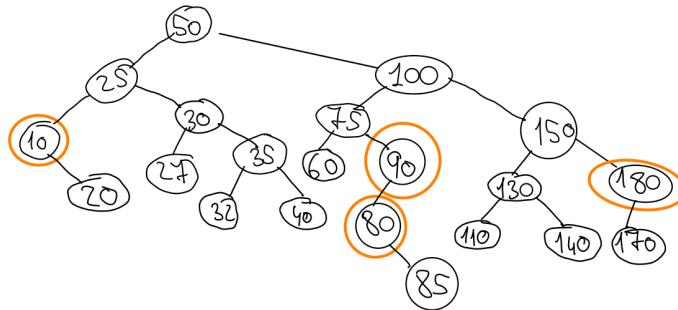
`node->parent->right=Nil`

```
BSTNode<T>* remove(BSTNode<T>* node){  
    //qui facciamo il caso 1 e il caso 2  
    //caso 1 nodo è una foglia  
    if (node->left == nullptr && node->right == nullptr) {  
        if (node == node->parent->left)  
            node->parent->left = nullptr;  
        else if (node == node->parent->right)  
            node->parent->right = nullptr;  
  
        return node;  
    }  
  
    ...  
    //analisi del CASO 2 e scrittura di seguito del codice  
    //nello stesso metodo(funzione), chiaramente
```

Caso 2: la chiave ha 1 figlio

Se z è figlio sinistro, il riferimento sinistro diventerà il figlio del parent(z), quindi l'unico figlio di z prende il posto di z e diventa figlio del parent(z), logicamente.

Esempio di albero:



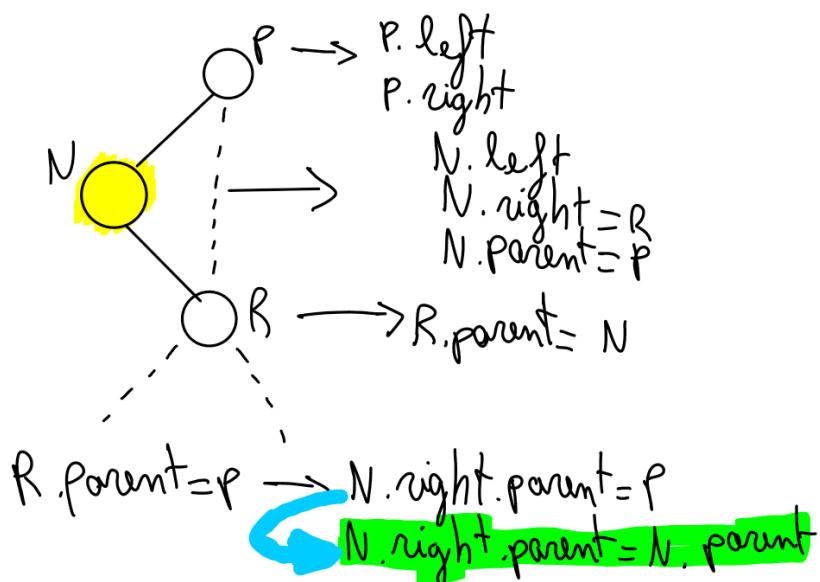
Si può applicare al 10, 90, 80, 180 ecc.. (i nodi che hanno **un figlio**)

In questo caso bisogna:

- collegare l'unico **figlio al parent** del nodo da eliminare;
- rimuovere i collegamenti da e per il **nodo da eliminare**(verso parent e verso il figlio);
- eliminare il nodo interessato.

Per **implementare** questo metodo uso un disegno:

1. Faccio un `node->parent->left=node->right;` così il left di parent è il right di node;
2. Prendo in considerazione il figlio del nodo da eliminare (giallo) e il suo parent è P;
3. Devo vedere se R è figlio destro o sinistro:
 - a. `node->right->parent = node->parent` non dipende dal fatto che il nodo da eliminare sia un figlio sinistro o destro **MA** dal fatto che il nodo abbia un figlio destro o sinistro;
4. Il nuovo figlio di P (come se fosse N) è R.



```

//caso 2 nodo ha 1 figlio
if (node->left == nullptr && node->right != nullptr) { //il nodo da eliminare ha un figlio destro
    node->right->parent = node->parent;
}

if (node == node->parent->left) { //se node è figlio sinistro
    // node->right->parent = node->parent; //parent del figlio destro è il parent del nodo da eliminare
}
  
```

```

//spostato fuori dall'if perchè indipendente
    node->parent->left = node->right; //56:15
}
else if (node == node->parent->right) {//se node è figlio destro
    node->parent->right = node->right;
}
return node;
}

//il nodo da eliminare ha un figlio sinistro
if (node->left != nullptr && node->right == nullptr) {
    node->left->parent = node->parent;
    //il nodo da eliminare è un figlio sx
    if (node == node->parent->left) {
        node->parent->left = node->left;
    }
    //il nodo da eliminare è un figlio dx
    else if (node == node->parent->right) {
        node->parent->right = node->left;
    }
    return node;
}
return nullptr;
}

```

Caso 2: la chiave ha 1 figlio (codice completo)

```

BSTNode<T>* remove(BSTNode<T>* node){
    //qui facciamo il caso 1 e il caso 2

//CASO 1: IL NODO DA RIMUOVERE E' UNA FOGLIA

if (node->left == nullptr && node->right == nullptr) {
    if (node == node->parent->left)
        node->parent->left = nullptr;
    else if (node == node->parent->right)
        node->parent->right = nullptr;

    return node;
}

//CASO 2: IL NODO DA RIMUOVERE HA 1 FIGLIO

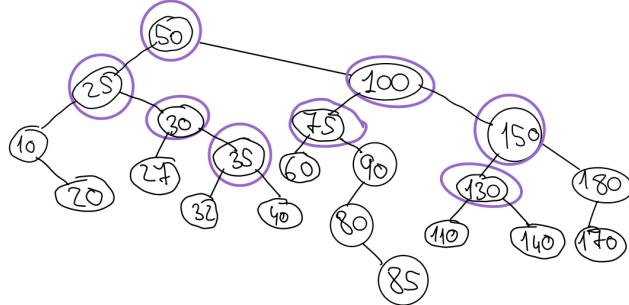
if (node->left == nullptr && node->right != nullptr) {
    node->right->parent = node->parent;

    if (node == node->parent->left) {
        // node->right->parent = node->parent; //parent del figlio destro è il parent del nodo da eliminare
        // spostato fuori dall'if perchè indipendente
        //se node è figlio dx o sx
        node->parent->left = node->right; //56:15
    }
    else if (node == node->parent->right) {
        node->parent->right = node->right;
    }
    return node;
}
//il nodo da eliminare ha un figlio sx
if (node->left != nullptr && node->right == nullptr) {
    node->left->parent = node->parent;
    //il nodo da eliminare è un figlio sx
    if (node == node->parent->left) {
        node->parent->left = node->left;
    }
    //il nodo da eliminare è un figlio dx
    else if (node == node->parent->right) {
        node->parent->right = node->left;
    }
    return node;
}
return nullptr;
}

```

Caso 3: la chiave ha 2 figli

Esempio di albero:



Dal grafico in figura si vede che il metodo si può applicare al nodo 25, 30, 50, 35, 75, 100, 150 ,130 ecc..

In questo caso **non vogliamo rimuovere il nodo**, ma la **chiave** che abbiamo indicato, cioè si vuole rimuovere il **VALORE** del nodo e non l'indirizzo del nodo stesso.

Cioè non vogliamo rimuovere la “struttura dati nodo”, ma solo il contenuto, il numero, la chiave.

In base a questa osservazione potremmo pensare di:

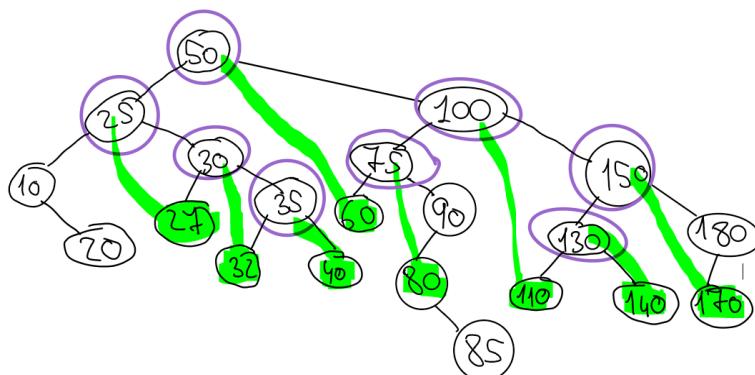
- **sostituire** il valore da eliminare con qualcosa che ora vedremo;
- **riorganizzare** l'albero BST in modo da mantenere la proprietà.

Adesso quindi non si parla di cancellare ma **sostituire e riorganizzare**.

Con quale chiave sostituisco il nodo da “cancellare”?

Devo trovare un valore che sia **il più vicino** possibile al nodo di cancellare che permette di fare **poche operazioni** per mantenere la proprietà del BST stesso.

Sostituendo la chiave da eliminare con il **successore di esso** si mantiene la proprietà.



Il successore è nel sottoalbero destro o nell'antenato. In questo caso lo trovo sempre nel sottoalbero destro perché ho un nodo con 2 figli quindi il sottoalbero destro c'è sempre.

Dal disegno soprastante salta all'occhio che il successore ha al più un figlio.

Per esempio: il successore di 75 è 80.

80 ha un figlio destro.

se 80 avesse un figlio sinistro, esso sarebbe il successore di 75.

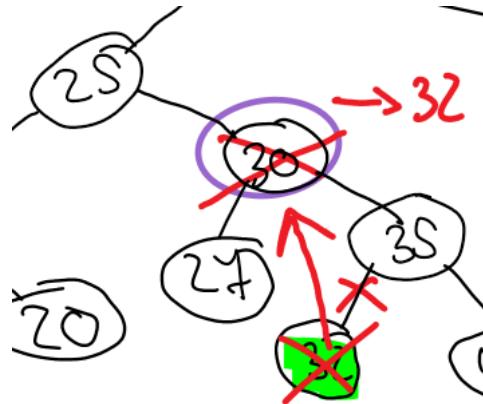
Il successore di un nodo con 2 figli avrà al più 1 figlio destro

A questo punto la logica è la seguente:

"Posso sostituire la chiave da eliminare con il successore della chiave stessa ed eliminare il successore dopo averlo assegnato alla chiave che abbiamo preso in considerazione(appunto)."

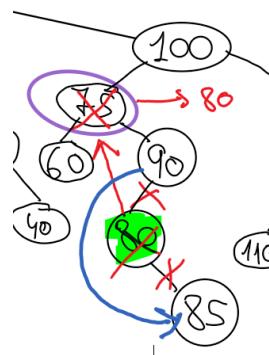
Esempio di applicazione 1:

- voglio eliminare il 30;
- trovo successore(30)=32;
- al posto di 30 metto 32;
- elimino il 32 e ricordo nel primo caso (0 figli del nodo 32);
- Verificando di nuovo la proprietà del BST si vede che è comunque mantenuta.



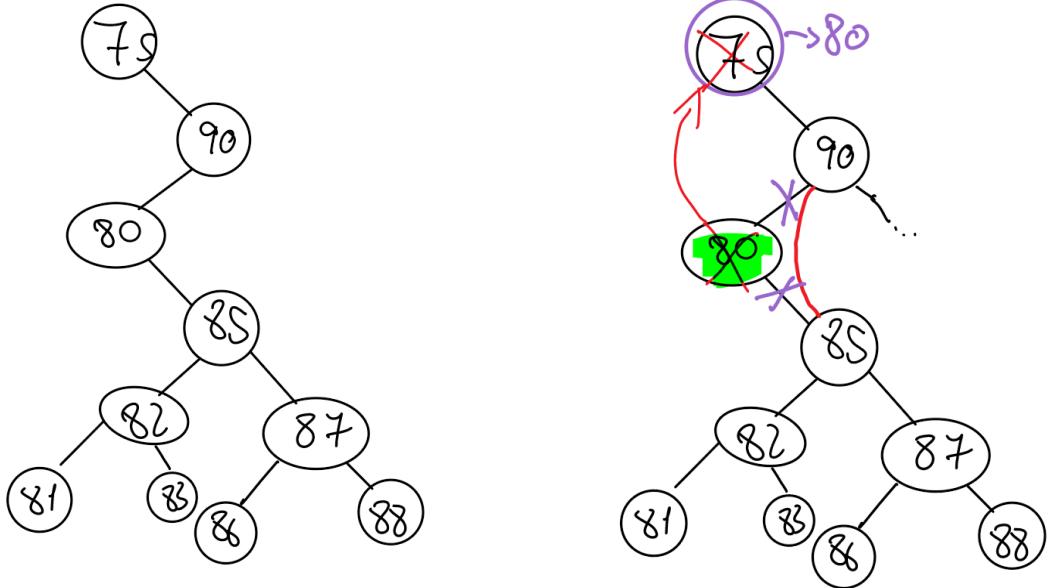
Esempio di applicazione 2:

- Voglio eliminare il 75;
- successore di 75 è 80.
- sostituisco 75 con 80;
- cancello 80. il figlio destro(in questo caso) diventa il figlio sinistro di 90 e i 2 collegamenti vengono rimossi



NOTE:

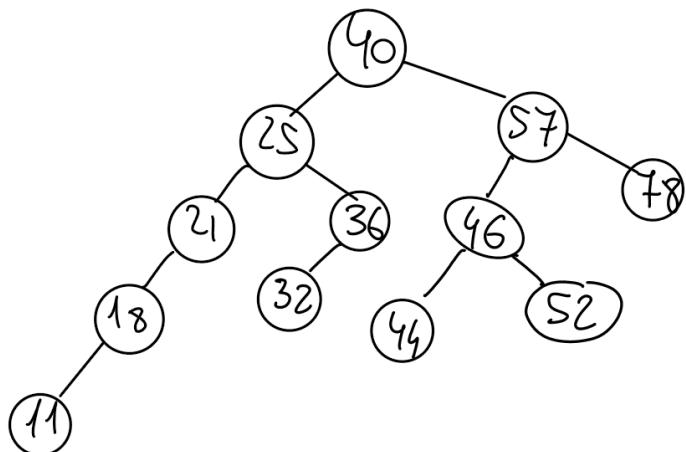
Se volessi rimuovere il **75**, **85** si porterà appresso tutto il suo sottoalbero tranquillamente quindi non mi devo preoccupare dei vari sottoalberi che non considero.



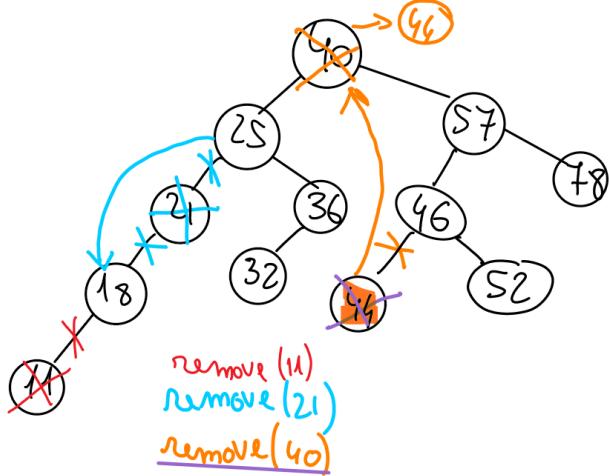
Non troverò mai un caso dove il successore del nodo da eliminare è un antenato.

Esercizio: Simulare le seguenti operazioni

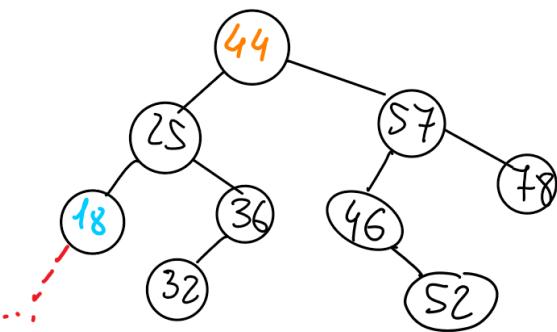
- Inserimento di 40,25,21,18,57,36,46,32,78,52,11,44



- Cancellazione di 11,21,40



L'albero finale è:



```

BSTNode<T>* remove(T key) {
    if (this->isEmpty()) {
        return nullptr;
    }

    BSTNode<T>* node = search(key); //vado a trovare il nodo che devo eliminare

    if (node == nullptr) {
        return nullptr; //se il nodo non è stato trovato
    }
    //se il nodo è stato trovato

    BSTNode<T>* toDelete = this->remove(node);
    if(toDelete != nullptr){
        return toDelete;
    }

    //nodo da eliminare ha 2 figli, non serve un altro if perchè è l'unica opzione rimasta
    //sostituisco la chiave nel nodo da eliminare con la chiave del suo successore
    BSTNode<T>* next = this->successor(node);
    //sostituzione della chiave
    T swap = node->key;
    node->key = next->key;
    next->key=swap;

    //devo eliminare next. Come?
    //fare una funzione che gestisce separatamente i primi 2 casi e la funzione che
    //richiamo prenderà come parametro un BSTNode<T>
    //richiamo la procedura di cancellazione dei primi 2 casi sul successore del nodo da eliminare
    toDelete = this->remove(next);

    return toDelete;
}

```

Algoritmi di visita

Un algoritmo di visita che è possibile applicare è quello chiamato “**stampa per livello**”.

Esso permette di stampare i **nodi** che si trovano allo **stesso livello** dell'albero.

Il codice non lo abbiamo scritto, era da fare come esercizio

Gestione errori “**throw**”

```
BSTNode<T>* max(BSTNode<T>* from) {  
    if (isEmpty()) {  
        return nullptr;  
    }
```

Per esempio non ha senso questo uso di `isEmpty()` perchè, in caso di albero vuoto ha poco senso fare un `return nullptr`. Si usa `throw()<stringa>` che solleva un'eccezione. Quindi diventa:

```
BSTNode<T>* max(BSTNode<T>* from) {  
    if (isEmpty()) {  
        throw "Empty BST!";  
    }
```

E se provo nel main, ottengo:

```
BST<int> bst;  
  
bst.max(); //genera l'errore  
  
output  
terminate called after throwing an instance of 'char const*'
```

Come lo gestisco? Si scrive la funzione `try { fun(); } catch (const char){...}`*

```
try{  
    bst.max();  
}  
catch (const char*){  
    cout << "empty" << endl;  
}
```

Provo `bst.max()`. Se l'errore viene sollevato allora “catcha” l'errore che sarà di tipo const char* (in questo caso) e stampa “empty”, cioè il messaggio del throw.

Chi chiama l'errore lo deve gestire. (con try catch).

Nel throw ho messo la stringa che non me l'ha stampata. Per recuperare il messaggio del throw basta dare un nome al const char*. in questo modo:

```
try{  
    bst.max();  
}  
catch (const char* ex){  
    cout << ex << endl;  
}
```

Esiste una classe EXCEPTIONS che gestisce tutte le eccezioni. Esse sono errori a run-time.

Per estendere la classe exception si può creare una propria classe che eredita da exception.

```
BSTNode<T>* max(BSTNode<T>* from) throw () {  
    if (isEmpty()) {  
        throw "Empty BST!";  
    }
```

```

1 // using standard exceptions
2 #include <iostream>
3 #include <exception>
4 using namespace std;
5
6 class myexception: public exception
7 {
8     virtual const char* what() const throw()
9     {
10         return "My exception happened";
11     }
12 } myex;
13
14 int main () {
15     try
16     {
17         throw myex;
18     } catch (exception& e)
19     {
20         cout << e.what() << '\n';
21     }
22     return 0;
23 }
```

throw(). scritto dopo la firma della funzione, vuol dire che il metodo può sollevare un'eccezione.
Questo procedimento è deprecato, cioè non si usa più, quindi lasciamo perdere questo discorso e dimentichiamolo :DDDD

Complessità operazioni BST

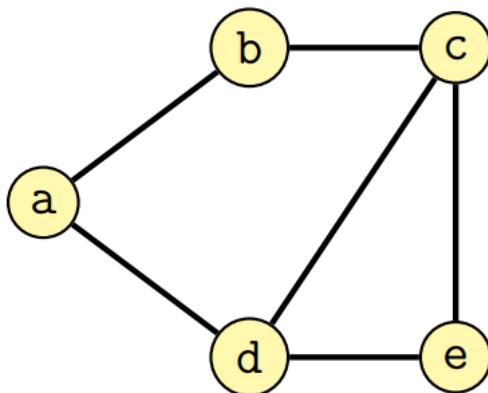
La complessità delle operazioni in un BST è $O(\log_n)$ perché essa varia in base all'altezza dell'albero.

GRAFI

Un grafo è un **insieme di nodi V** detti **vertici** connessi da collegamenti chiamati **archi** raggruppati in un insieme E.

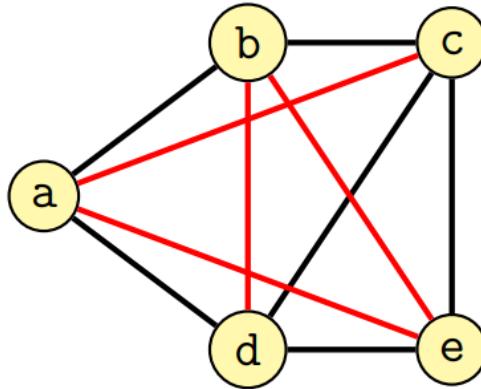
- Il grafo si definisce $G = (V, E)$;
- Grafo **orientato**: gli archi hanno una direzione;
- Grafo **non orientato**: gli archi non hanno una direzione;
- L'insieme E è definito $V \times V \rightarrow \mathbb{R}$ quindi sono coppie di vertici che danno origine a un numero reale.
- **GRAFO SPARSO** è un grafo dove il **numero degli archi** è **MOLTO MINORE** rispetto al quadrato del numero dei vertici(gli archi collegano solo alcuni nodi), quindi si ha $|E| \ll |V|^2$.

Sostanzialmente, è sparso se il grafo ha “pochi archi”.



GRAFO DENSO ci sono **tantissimi archi**, quindi si ha $|E| \simeq |V^2|$

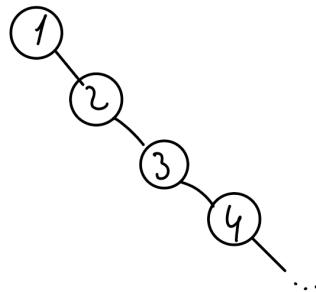
Sostanzialmente, è denso se il grafo ha “molti archi”.



- Un nodo v_i si dice **ADIACENTE** a un nodo v_j se esiste un arco $e_{ij} = (v_i, v_j)$ che li collega;
 - In caso di arco **non orientato** fra A—B, allora A è adiacente a B e **VICEVERSA**.
 - In caso di arco **orientato** A → B, allora A è adiacente a B ma **NON VICEVERSA**.

Sequenza di inserimento di esempio: 1 2 3 4 5 6 7 8 9 10. Se li metto in un BST cosa esce?

Si produce un albero che degenera in una lista. Per questo motivo la **lista può essere vista come un grafo orientato**.



Rappresentazione dei grafi

I grafi possono essere rappresentati in due diversi modi:

1. **LISTA DI ADIACENZA**;
 - a. Contiene una “lista”, appunto, di vertici adiacenti al vertice che si prende in considerazione, quindi può anche essere infinita;
2. **MATRICE DI ADIACENZA**;
 - a. Ha dimensione $|V| \times |V|$
 - b. se esiste un arco fra i e j , l’elemento in posizione $[i, j]$ avrà valore 1
 - c. spazio richiesto in memoria $|V|^2$
 - d. La complessità $\Theta(|V|^2)$ vale a prescindere da E , quindi per questo è sempre $|V|^2$.

In caso di **GRAFO DENSO** è molto conveniente utilizzare una **matrice di adiacenza**.

In caso di **GRAFO SPARSO** è molto conveniente utilizzare una **lista di adiacenza**.

LISTA DI ADIACENZA

Un grafo si può rappresentare con una **lista di adiacenza**.

Gli elementi da tenere a mente (per la creazione della struttura dati) sono:

- Vertice del grafo `GraphVertex<T>` che non avrà nessun attributo;

- I vertici adiacenti (una lista di nodi del grafo) che avrà gli attributi:

```

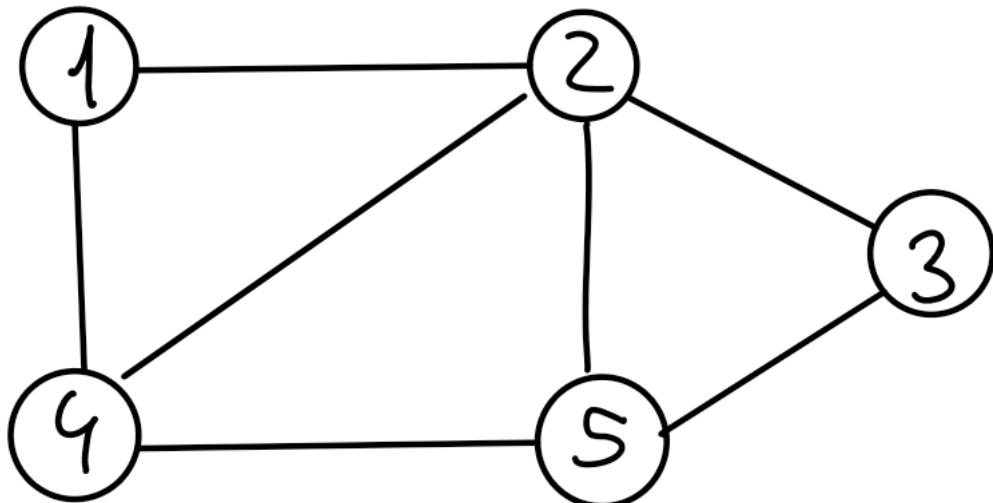
    o List< GraphVertex<T> > vertices;
    o bool isOriented ;
    o int nVertices ;
  
```

Cerchiamo gli adiacenti di:

- **1** = 2, 4;
- **2** = 1, 3, 4, 5;
- **3** = 2, 5;
- **4** = 1, 2, 5;
- **5** = 2, 3, 4;

Ogni nodo ha una **lista** che contiene i nodi ad esso adiacenti.

Sarà scelta nostra se i nodi saranno limitati o no.



- Per rappresentare i **nodi (finiti)** posso usare un **array**. Per ogni posizione dell'array c'è una lista che contiene i nodi adiacenti. Quindi si ha un **array di liste**.
- Per rappresentare i **nodi (non finiti)** posso usare una **lista**. Per ogni nodo della lista c'è, a sua volta, una lista di nodi adiacenti al nodo preso in considerazione. Quindi si ha una **lista di liste**.

Per ogni vertice nel grafo si inseriscono i nodi adiacenti in una lista linkata ad esso collegato, creando, così, la *lista di liste*.

Osservazioni

- In caso di uso della **lista di adiacenza** per la rappresentazione del grafo ci vuole **più tempo** per accedere ad un elemento (rispetto alla matrice) perché **per accedere** alla lista devo "scorrere" con un puntatore ([ptr](#)).
- Se il **grafo è piccolo** conviene **utilizzare la matrice di adiacenza** per l'implementazione.

Codice liste di adiacenza

Ci si accorge che, usando una **lista già implementata**, in maniera **ereditaria**, in testa ho la chiave del vertice e successivamente ci sono i nodi adiacenti.

Per **accedere ad un vertice** ben preciso del grafo è utile scrivere il metodo `search()` che restituisce il nodo ricercato.

```

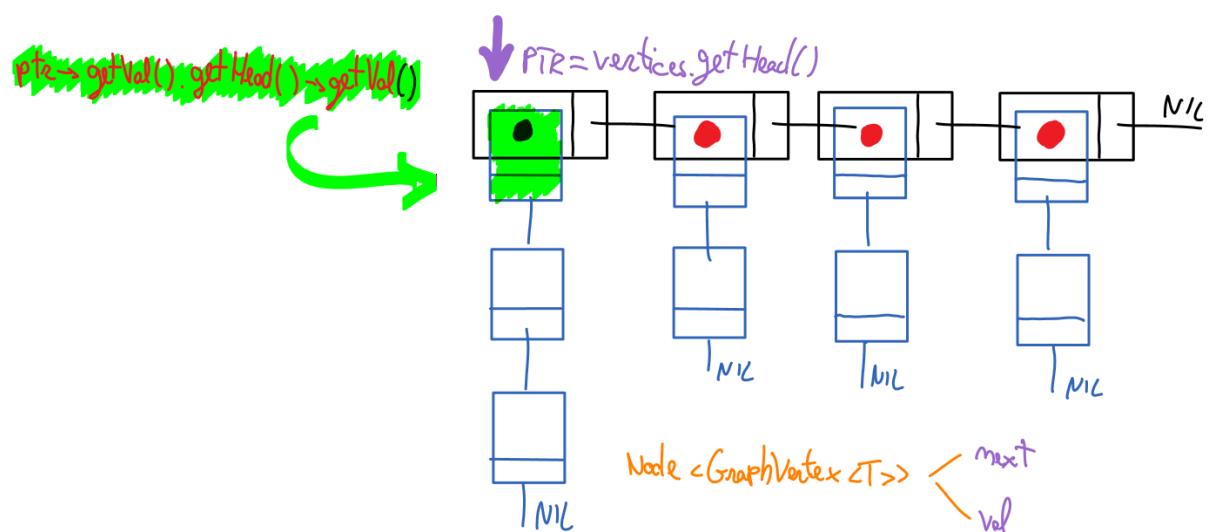
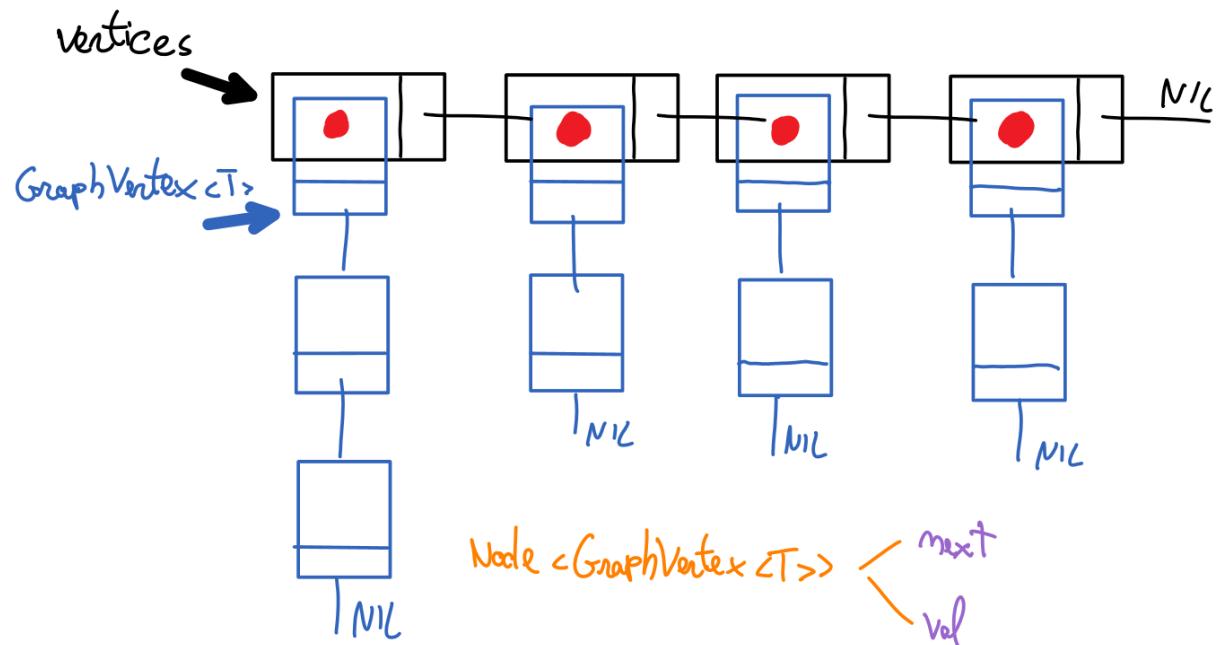
Node< GraphVertex<T> *>* search(T key){
    if(nVertices==0)
        return NULL;
    // GraphVertex<T>* ptr=vertices.getHead(); //è giusta questa riga? no. Per questo diventa(riga sotto)
    Node< GraphVertex<T> *>* ptr=vertices.getHead(); //dove ho cambiato (anche) il tipo di ritorno del metodo
  
```

```

        while(ptr){
            //ptr->getVal() dà come risultato la lista di adiacenza del vertice ptr
            //ptr->getVal().getHead() è la testa della lista di adiacenza
            //ptr->getVal().getHead()->getVal() è il vertice di testa della lista di adiacenza
            if(key==ptr->getVal().getHead()->getVal()){ //la chiave è == alla head della lista del nodo?
                return ptr;
            }
            ptr=ptr->getNext();
        }
        return NULL; //ptr è andato a null. Non ho trovato il valore cercato
    }
}

```

Ho una lista fatta di nodi di tipo `GraphVertex<T>`. Essa è una lista di una lista. Quando si cerca un valore, ci interessa la head all'interno della lista contenuta nel nodo, cioè `ptr->getVal().getHead()`



Dopo aver implementato il codice per la ricerca di un vertice all'interno del grafo, possiamo aggiungere i metodi per **l'aggiunta di un vertice e l'aggiunta di archi**.

Di seguito il codice completo contenente anche il main (per semplicità di rappresentazione):

```

#ifndef GRAPH_LIST_H
#define GRAPH_LIST_H

```

```

#include <iostream>
#include "list.h"
using namespace std;

//estendo la classe list in modo da avere il vertice nella head e la lista di adiacenza nei successori(next)
template<typename T>
class GraphVertex : public List<T>{
public:
    GraphVertex(T key) : List<T>(){
        List<T>::insertTail(key); //in questo modo ho già il nodo in testa inserito
    }

    friend ostream& operator<<(ostream& out, GraphVertex<T>& v){
        out << "Graph Vertex with key " << v.getHead()->getVal() << ": ";
        out << "\tAdjacency List: ";
        Node<T>* ptr = v.getHead()->getNext();
        while (ptr){
            out << " ->" << ptr->getVal();
            ptr=ptr->getNext();
        }
        return out;
    }
};

template <typename T>
class GraphList{
    List< GraphVertex<T> > vertices;
    int nVertices=0; //contatore dei vertici
    bool isOriented;
public:
    GraphList(bool isOriented=true):isOriented(isOriented){}

    void addVertex(T key){
        GraphVertex<T> toAdd(key);
        vertices.insertTail(toAdd);
        nVertices++;
    }

    void addEdge(T key1, T key2){
        Node< GraphVertex<T> > *node1 = this->search(key1);
        Node< GraphVertex<T> > *node2 = this->search(key2);

        if(node1 && node2){ // node1 e node2 non sono nullptr, quindi sono stati trovati
            //vuol dire che devo aggiungere node1 alla lista di adiacenza di node2
            //vuol dire che devo aggiungere node2 alla lista di adiacenza di node1
            node1->getVal().insertTail(key2); //prendo la lista di node1 e aggiungo node2 con un insertTail
            if(!this->isOriented){
                node2->getVal().insertTail(key1);
            }
        }
    }

    Node< GraphVertex<T> >* search(T key){
        if(nVertices==0){
            return NULL;
        }

        // GraphVertex<T>* ptr=vertices.getHead(); //è giusta questa riga? Diventa
        Node< GraphVertex<T> >* ptr=vertices.getHead(); //quindi ho cambiato il tipo di ritorno del metodo
        while(ptr){
            if(key==ptr->getVal().getHead()->getVal()){ //la chiave è == alla head della lista del nodo?
                return ptr;
            }
            ptr=ptr->getNext();
        }

        return NULL; //ptr è andato a null. Non ho trovato il valore cercato
    }

    friend ostream& operator<<(ostream& out,GraphList<T>& g){
        out << g.vertices ;
        return out;
    }
};

#endif

int main(){
    GraphList<int> g;

    g.addVertex(10);
    g.addVertex(11);
    g.addVertex(12);
    g.addVertex(13);

    //aggiungo archi
}

```

```

g.addEdge(10,11);
g.addEdge(12,13);
g.addEdge(13,11);

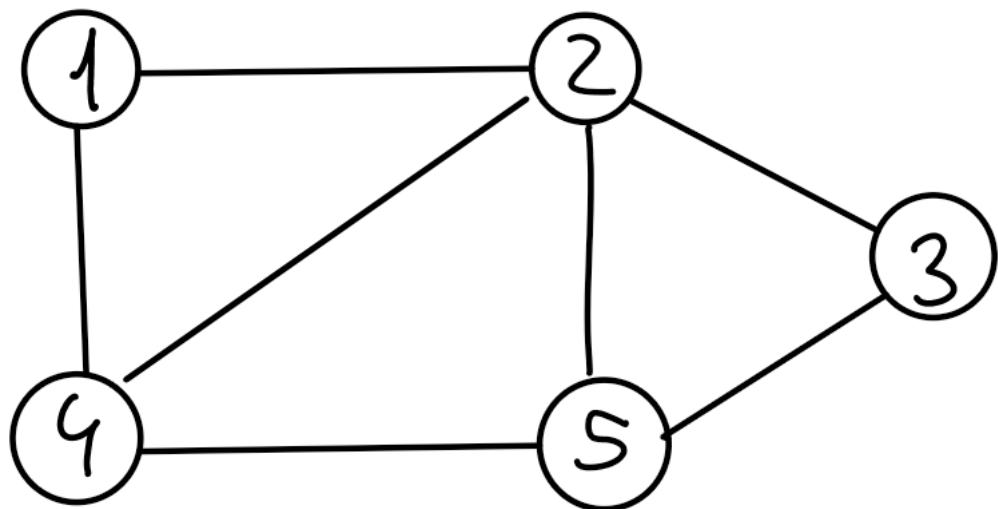
cout << g;
}

```

MATRICE DI ADIACENZA

Un grafo si può rappresentare anche con una **matrice di adiacenza** dove per ogni cella i, j c'è il numero di archi che collegano i e j , quindi verranno usati array bidimensionali.

Costruiamo la **matrice di adiacenza per il grafo non orientato** in figura:



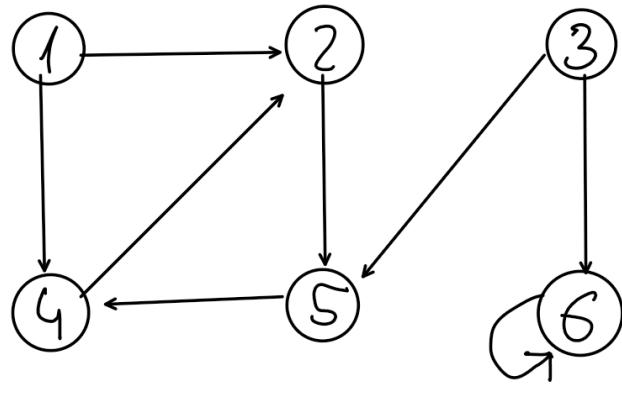
	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	1
3	0	1	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

La **diagonale principale** segna la simmetria della matrice (c'è simmetria perché il grafo non è orientato).

Se si immagina di piegare sulla diagonale l'angolo giu-sx sul lato su-dx, i valori che si sovrappongono sono uguali.

I grafi rappresentano tutti gli algoritmi principali che ci possono essere(algoritmi di ricerca online, google maps, rete stradale). Il grafo è molto flessibile e molto "potente".

Adesso vediamo la **matrice di adiacenza per un grafo orientato**:



	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

In questo caso la **relazione di adiacenza non è simmetrica** e se la rappresentassi con delle liste avrei delle **liste più corte**.

Matrice di adiacenza codice

Implementiamo un grafo non orientato con una matrice di adiacenza:

- `Vertex<T>` che avrà gli **attributi**:
 - `T key`;
- Grafo con matrice di adiacenza `Graph<T>` che avrà gli **attributi**:
 - Una **massima dimensione** `maxSize`;
 - Degli **array di puntatori** ai vertici `Vertex<T>** vertices`;
 - La **matrice di adiacenza** `bool** adj`;
 - Il **numero di vertici** attuali `int nVertices`;

```

#ifndef GRAPH_H
#define GRAPH_H

#include <iostream>

using namespace std;

template <typename T>
class Vertex{
private:
    T key;
    template <typename U>
    friend class Graph;

public:
    Vertex(T key) : key(key){}
    Vertex():Vertex(NULL){}
    bool operator==(Vertex<T>& v){
        return this->key==v.key;
    }
    friend ostream& operator<< (ostream& out, Vertex<T>& v){
        out << v.key;
        return out;
    }
};

template <typename T>
class Graph{
private:
    Vertex<T>** vertices; //vedi riga 38. Ho creato
    bool** adj;
}
  
```

```

int maxSize=0;
int nVertices=0;
public:
    Graph(int max_size) :maxSize(max_size){
        vertices=new Vertex<T>*[max_size]; //aggiungo * per avere un array di puntatori a vertici per evitare ambiguità con la new
        adj=new bool*[max_size];
        for(int i=0; i<max_size;i++){
            adj[i]=new bool[max_size] {0} ; //inizializzo tutto a false
            // for(int j=0; j<max_size; j++){
            //     adj[i][j]=false;
            // }
        }
    }

    void addVertex(T key){
        if(this->nVertices==this->maxSize){
            cerr << "Graph is full" << endl;
            return;
        }
        vertices[nVertices++]=new Vertex<T>(key);
    }

    void addEdge(T key1, T key2){
        int i=this->search(key1);
        int j=this->search(key2);

        if(i!=-1 & j!=-1){
            adj[i][j]=true;
            adj[j][i]=true;
        }
        else{
            if(i==-1)
                cerr << "There is no vertex with key " << key1 << endl;
            else
                cerr << "There is no vertex with key " << key2 << endl;
        }
    }

    // Vertex<T>& search(T key){ //errata, modificata sotto
    //     if(this->nVertices==0)
    //         return NULL;
    //     for(int i=0; i<this->nVertices;i++){
    //         if(vertices[i].key==key){
    //             return vertices[i];
    //         }
    //     }
    //     return NULL;
    // }

    int search(T key){ //devo restituire un indice per accedere al vettore
        if(this->nVertices==0)
            return -1;
        for(int i=0; i<this->nVertices;i++){
            if(vertices[i]->key==key){
                return i;
            }
        }
        return -1;
    }

    friend ostream& operator<<(ostream& out, Graph<T>& g){
        for(int i=0; i<g.nVertices;i++){
            out << "v[" <<i<<"] = " << *(g.vertices[i])<<"\t";
        }
        out << endl;

        for(int i=0; i<g.nVertices;i++){
            for(int j=0;j<g.nVertices;j++){
                if(g.adj[i][j]){
                    out << "(" << i << ", " << j << ")" << endl;
                }
            }
        }
        return out;
    }
};

#endif

```

COMPLESSITÀ LISTA vs MATRICE DI ADIACENZA

E' possibile fare dei paragoni per quanto riguarda la complessità delle 2 rappresentazioni dei grafi.

Per quanto riguarda la **MATRICE DI ADIACENZA**:

- spazio richiesto in memoria $|V|^2$
- $\Theta(|V|^2)$ a prescindere da E , quindi per questo è sempre V^2 .

Per quanto riguarda la **LISTA DI ADIACENZA**, lo spazio richiesto in memoria:

- $\Theta(|V| + |E|)$ In caso di grafo sparso si ha $E << V^2$
- Orientato: $2|E|$
- Non orientato: $|E|$

Asintoticamente parlando cresce in meno tempo $\Theta(|V| + |E|)$ quindi conviene sempre usare una lista di adiacenza piuttosto che la matrice di adiacenza per la rappresentazione dei grafi.

Visita in ampiezza di un grafo (Breadth-First Search)BFS

Questo algoritmo può **partire da un nodo qualunque**.

Il nodo di partenza è detto **NODO SORGENTE**.

“In ampiezza” perchè la visita procede per **distanza dal nodo sorgente**.

Quindi al primo passo visito tutti i nodi a distanza 1. Al passo k avrò scoperto nodi a distanza k dal nodo sorgente.

Alla fine dell'algoritmo di visita **BFS** si può notare che i **valori di discovery** dei vari nodi corrispondono esattamente al **numero di cammini minimi che esistono fra la sorgente e il nodo considerato**.

Significato dei colori

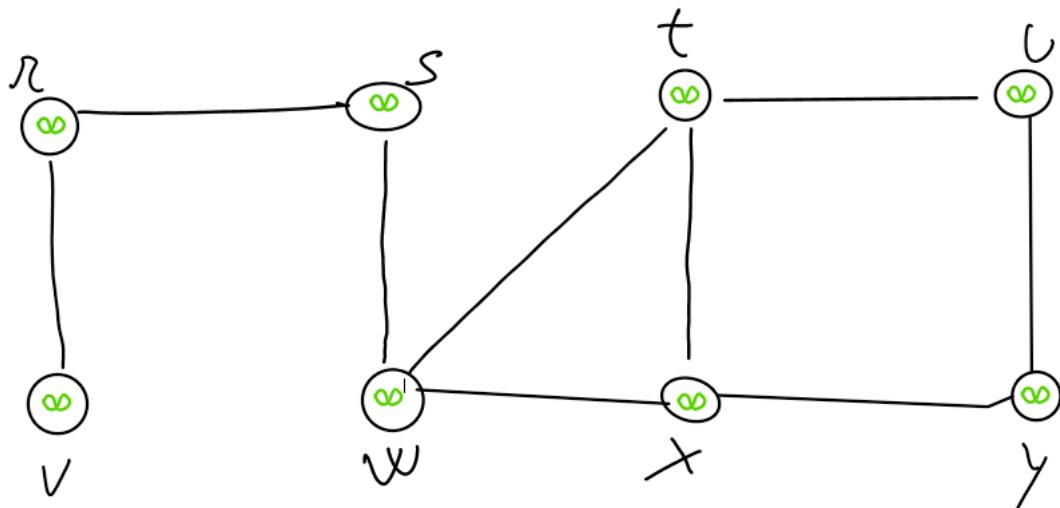
1. **BIANCO** → Nodo **non ancora scoperto**, non visitato;
2. **GRIGIO** → Nodo **scoperto parzialmente** (non totalmente esplorato);
3. **NERO** → **Scoperto totalmente**.

Procedimento

Inizialmente tutti i nodi devono essere **BIANCHI** e tutti i nodi devono avere **INFINITO** come valore di discovery al loro interno. (vedi figura sotto)

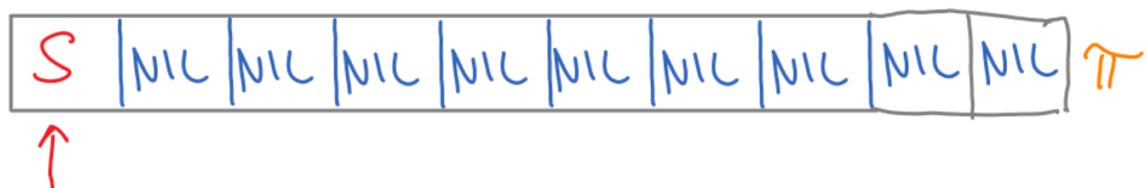
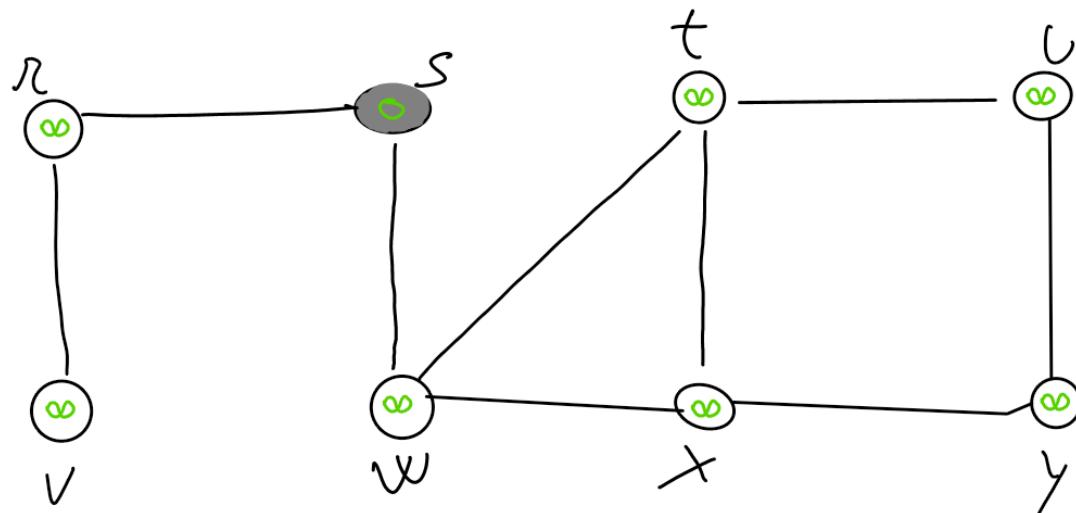
ARRAY DISCOVERY “d” = contiene un numero che rappresenta l'istante di quando il nodo è stato scoperto;

CODA DEI PREDECESSORI π = array dei predecessori contiene i predecessori dei nodi che si stanno considerando in quel preciso istante.

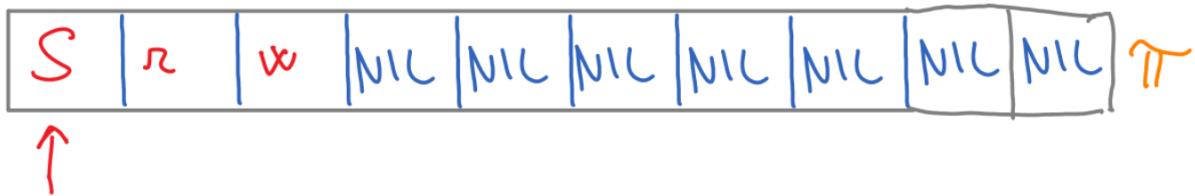


Dopo questa inizializzazione si sceglie il nodo sorgente. In questo esempio scegliamo S come sorgente.

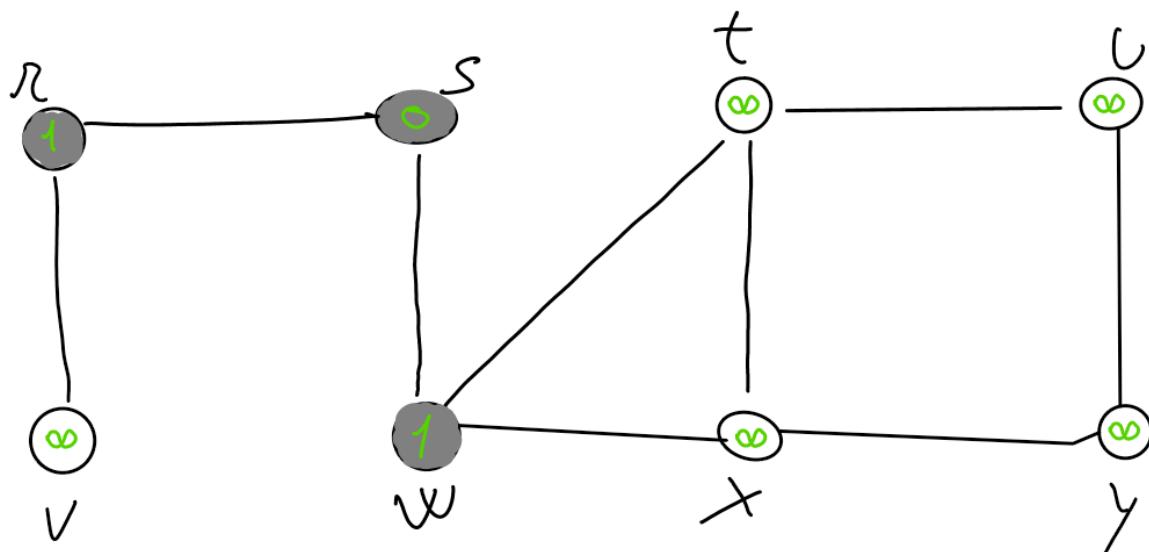
1. Scelgo s come nodo sorgente;
2. Aggiungo s alla coda, lo faccio diventare GRIGIO e quindi lo visito.
3. **s ha nodi adiacenti bianchi?** Sì, ovvero r e w;



4. Aggiungo alla coda con `insertTail(r)` e `insertTail(w)` i nodi adiacenti di s che sono BIANCHI;

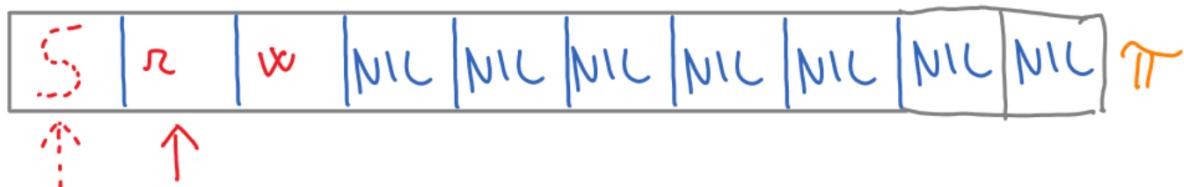


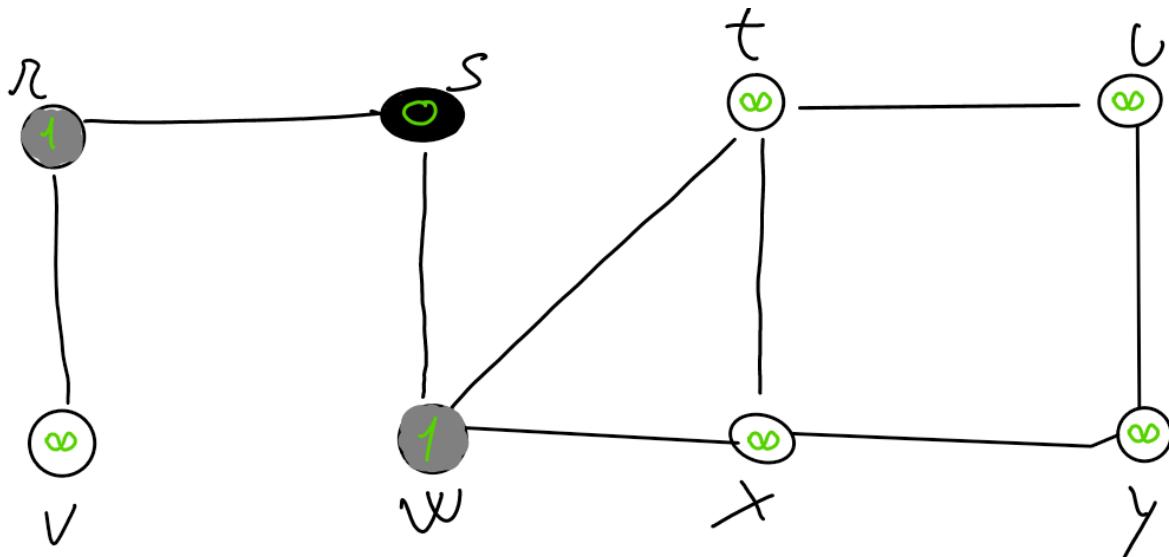
5. *r* e *w* diventano **GRIGI**, vengono visitati e il loro valore di discovery vale esattamente *predecessore* + 1, quindi in questo caso 1.



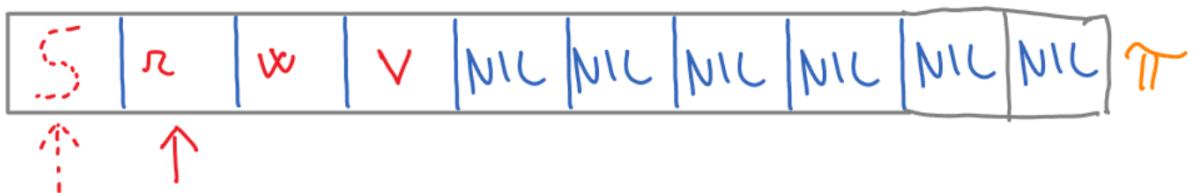
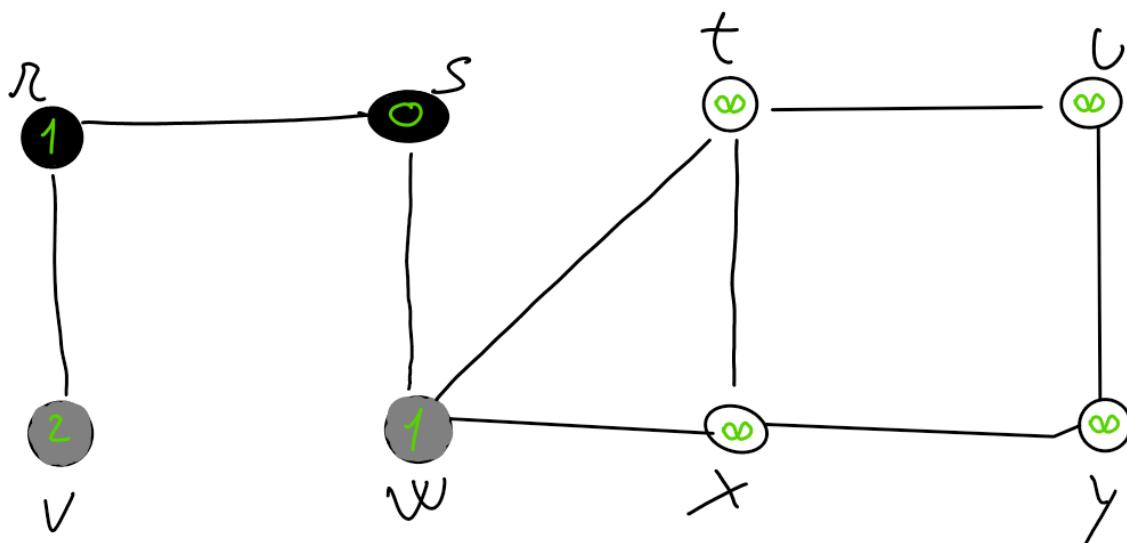
6. Guardo la coda e vedo che il puntatore punta al nodo *s*. *s* ha nodi adiacenti bianchi non visitati? No. Allora rimuovo dalla coda *s*, lo faccio diventare **NERO** e faccio **avanzare il mio puntatore**, così da considerare il nodo *r*, quindi ho $s \rightarrow r$.

Il puntatore viene fatto avanzare finché la coda non è terminata.





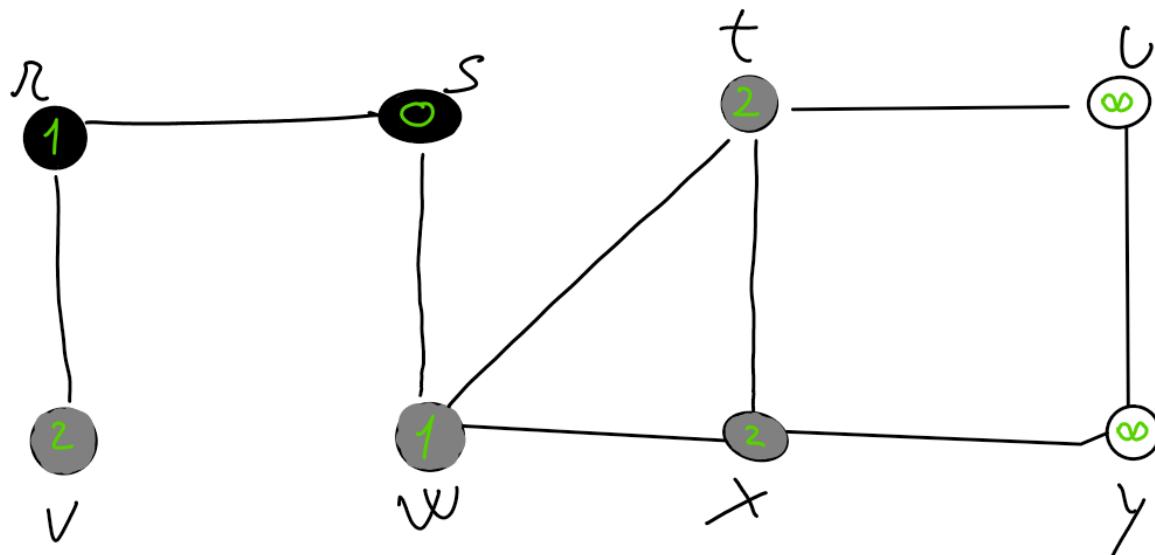
7. Il mio puntatore punta sul nodo r .
 8. r ha nodi adiacenti bianchi? Sì. In particolare un solo nodo v .
 9. v viene aggiunto alla coda, viene visitato, diventa **GRIGIO** e il suo valore di discovery è $predecessore + 1$, cioè 2.



- r ha ancora nodi adiacenti bianchi?* No. Quindi diventa **NERO**, viene rimosso dalla coda e faccio avanzare il mio puntatore.
 - Il puntatore della coda punta sul nodo *w*.

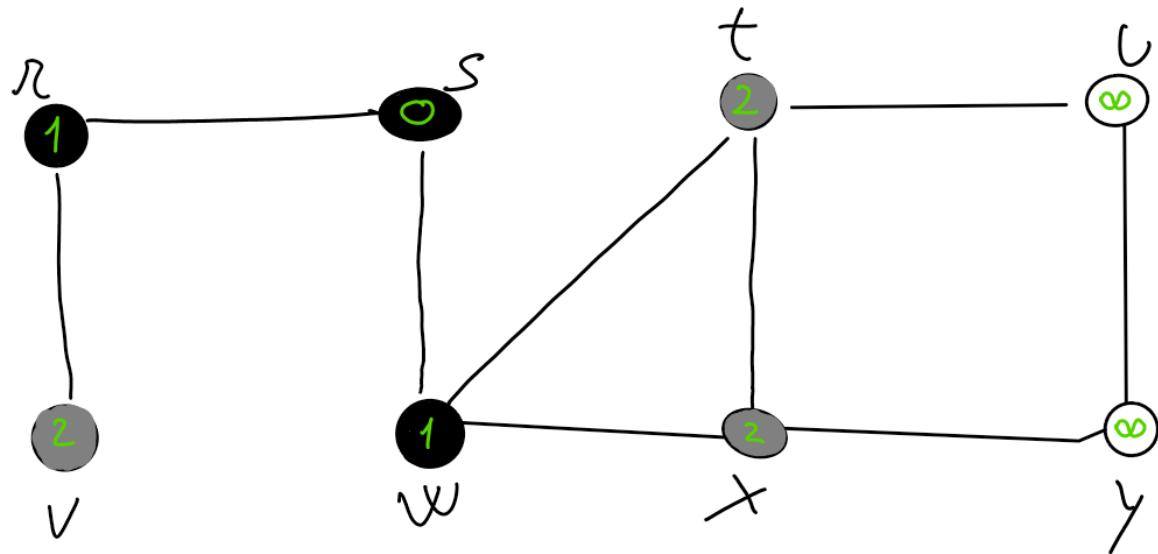
S	<i>s</i>	<i>w</i>	<i>v</i>	NIL	NIL	NIL	NIL	NIL	NIL	π
↑	↑	↑								

12. *w* ha nodi adiacenti BIANCHI? Sì, in particolare *t* e *x*. Questi ultimi vengono aggiunti alla coda, vengono visitati, diventano GRIGI e il loro valore di discovery vale *predecessore* + 1, cioè 2.

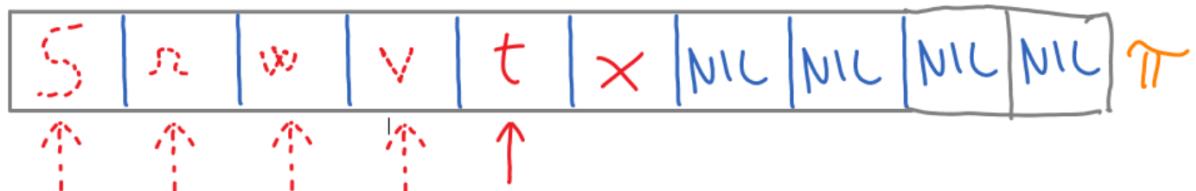
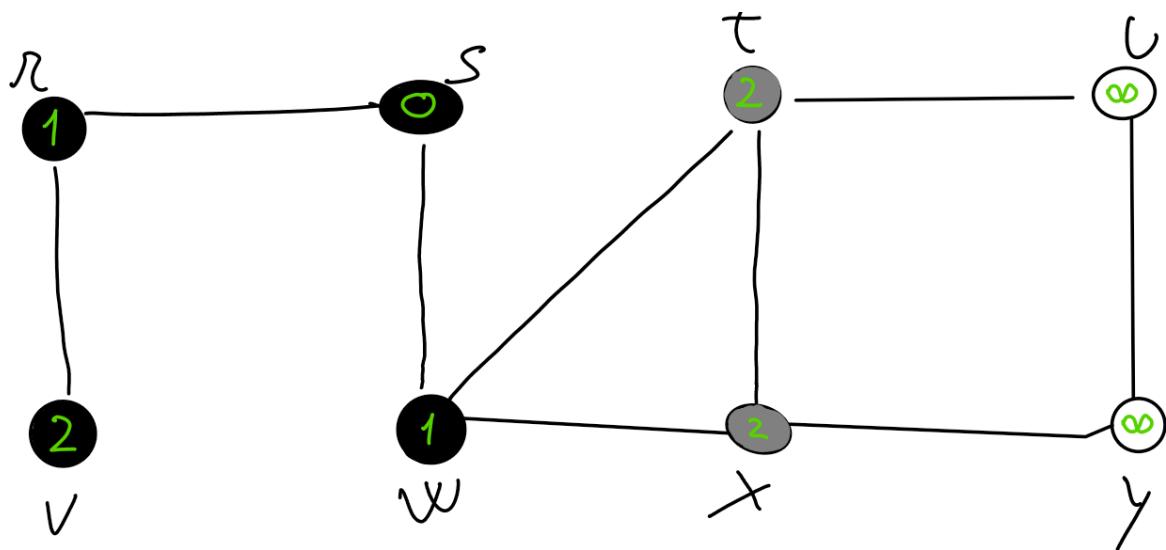


S	<i>s</i>	<i>w</i>	<i>v</i>	<i>t</i>	<i>x</i>	NIL	NIL	NIL	NIL	π
↑	↑	↑								

13. *w* ha altri nodi adiacenti BIANCHI? No. Quindi diventa NERO, viene rimosso dalla coda e viene fatto avanzare il puntatore *w* → *v*.

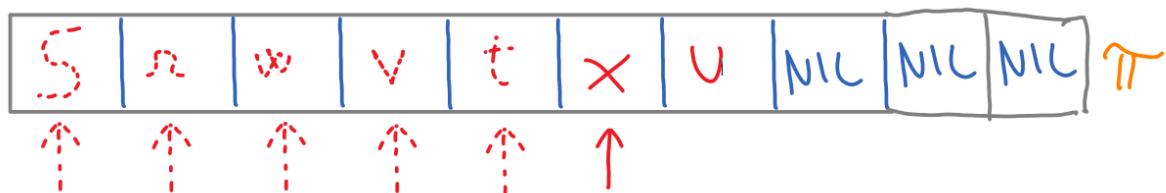
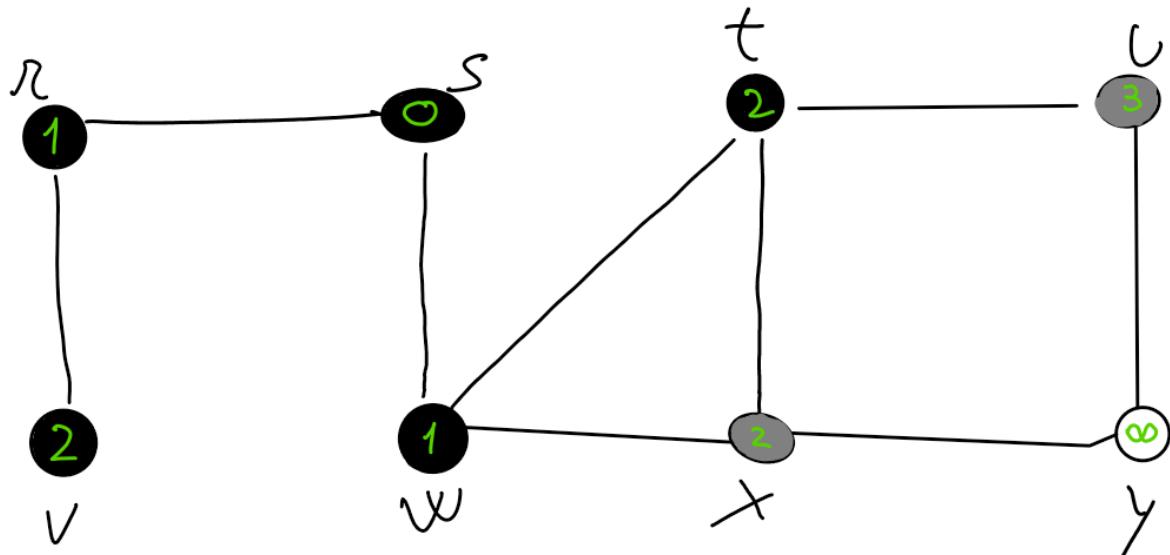


14. Il puntatore è su v . v è grigio, quindi vuol dire che è stato già visitato. Non ha altri nodi vicini che siano BIANCHI, quindi diventa **NERO**, quindi lo rimuovo anche dalla coda e faccio avanzare il puntatore $v \rightarrow t$

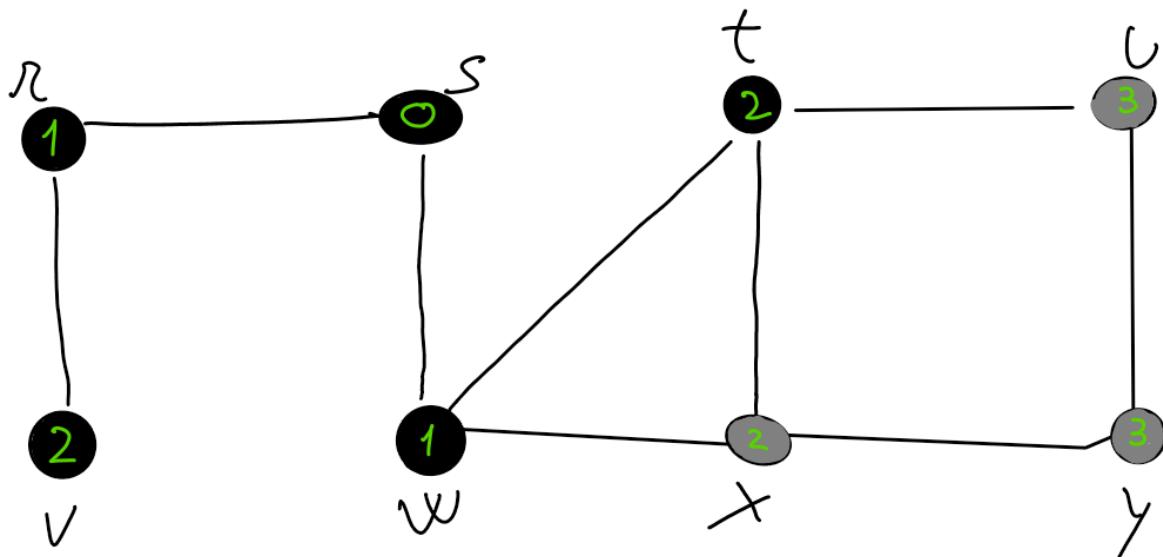


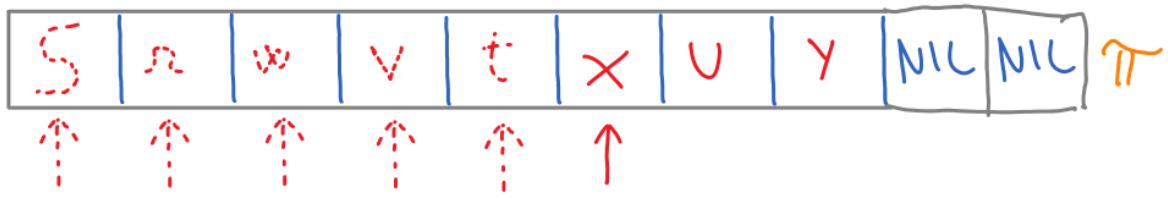
15. t ha nodi adiacenti bianchi? Sì, ovvero u . Esso viene visitato, diventa GRIGIO, il suo valore di discovery è $predecessore + 1$, ovvero 3 e viene aggiunto alla coda;

16. t ha altri nodi adiacenti bianchi? No. Allora t diventa NERO e si fa avanzare il puntatore della coda, quindi $t \rightarrow x$;

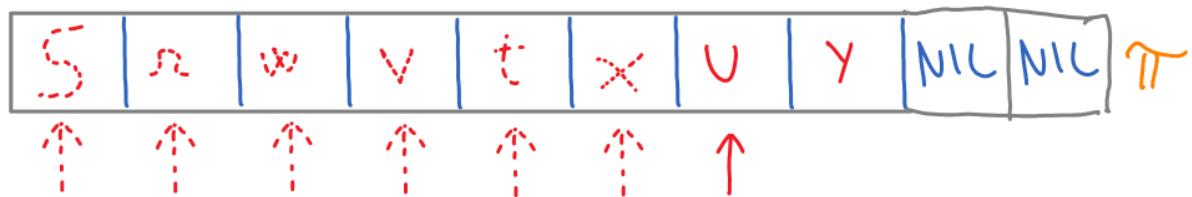
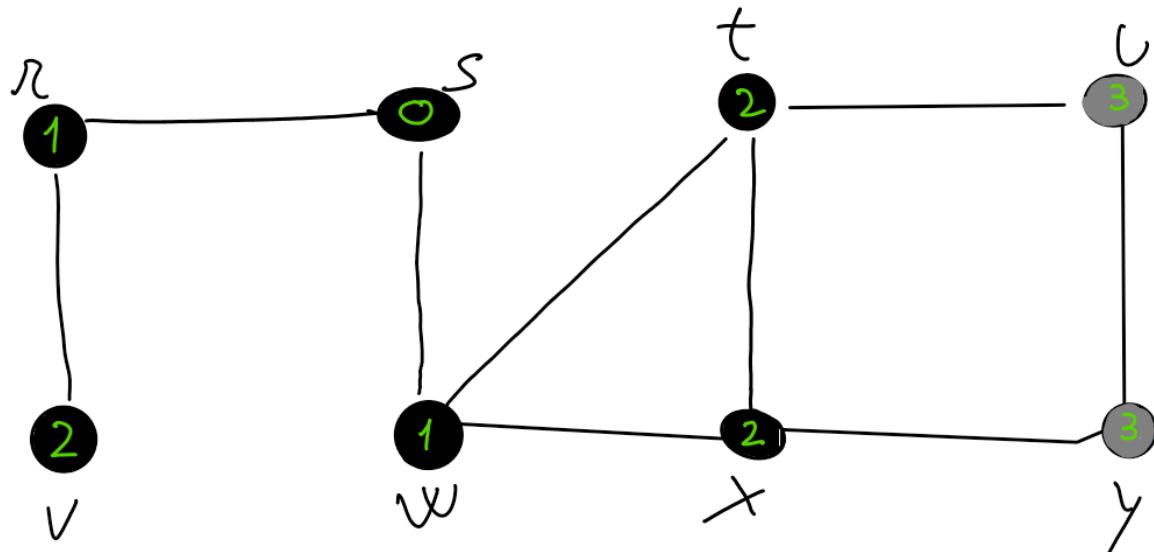


17. x ha nodi adiacenti BIANCHI? Sì, in particolare y . Esso viene visitato, diventa GRIGIO, il suo valore di discovery è $predecessore + 1$, ovvero 3 e viene aggiunto alla coda;

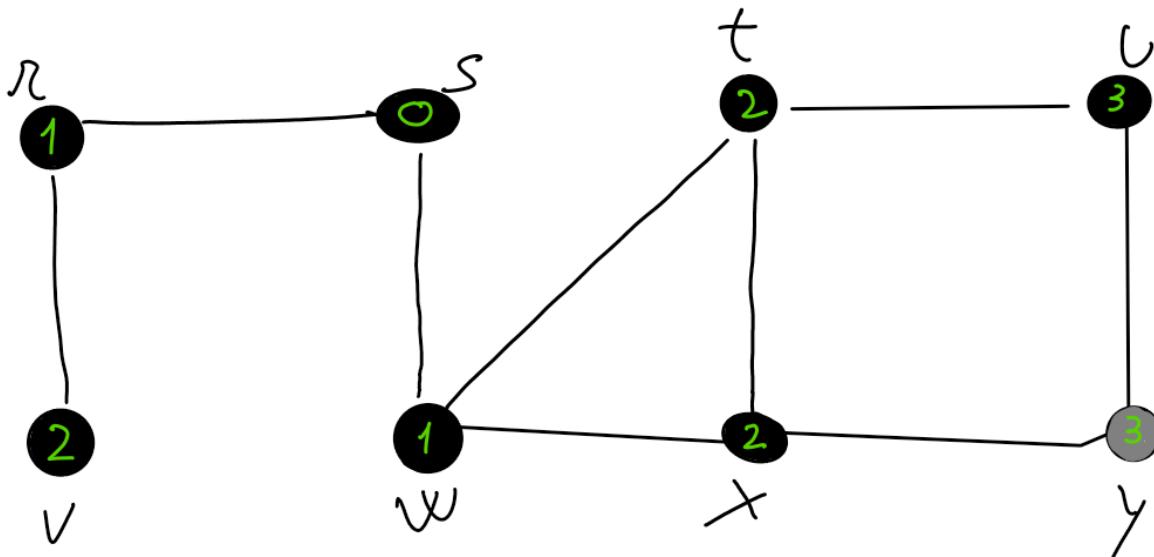


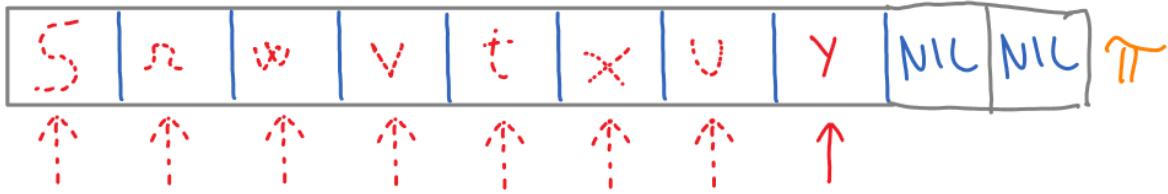


18. x ha altri nodi adiacenti BIANCHI? No. Allora esso diventa NERO, e avanza il puntatore della coda, quindi $x \rightarrow u$;

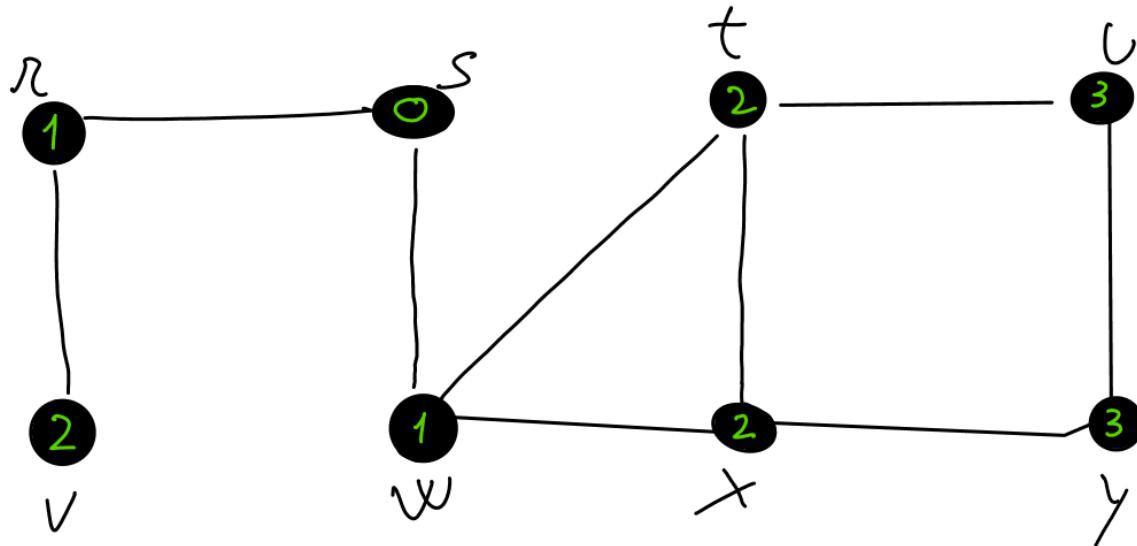


19. u ha nodi adiacenti BIANCHI? No. Allora diventa NERO e viene fatto avanzare il puntatore alla coda, quindi $u \rightarrow y$;

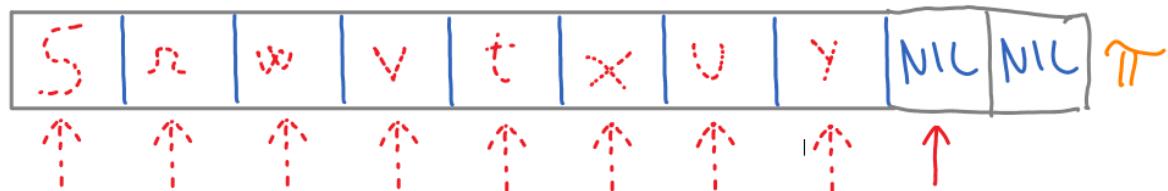




20. y ha nodi adiacenti BIANCHI? No, allora esso diventa NERO.



21. La condizione non è più soddisfatta visto che la coda è terminata e quindi il `ptr->next` è `nullptr`, quindi vuol dire che la visita può terminare.



Visita in profondità

Rispetto alla visita in ampiezza, quindi BFS, nel caso della visita in profondità non tutti i vertici a distanza “ k ” verranno scoperti all’istante “ k ”.

Anche in questo caso valgono i significati per i vari colori assegnati precedentemente. Li riscriviamo:

1. BIANCO → Nodo **non ancora scoperto**, non visitato;
2. GRIGIO → Nodo **scoperto parzialmente** (non totalmente esplorato);
3. NERO → **Scoperto totalmente**.

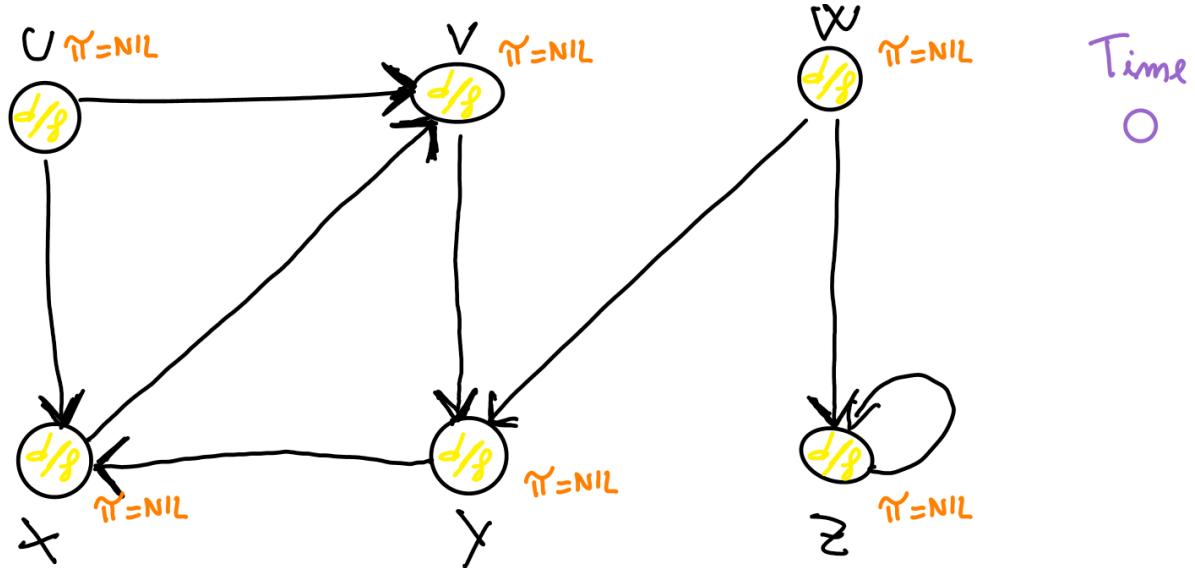
A differenza della visita in ampiezza dove avevamo un valore unico di discovery all’interno del vertice, nel caso della visita in profondità **abbiamo 2 valori (d/f)** dove d rappresenta il **tempo di scoperta (discovery)** della visita BFS e f rappresenta il **l’istante di completamento della visita**, quindi l’istante in cui il vertice da **GRIGIO** diventa **NERO**.

In più si deve **tenere conto degli istanti di tempo** che trascorrono, quindi si avrà una **variabile time** che verrà incrementata di volta in volta a dovere. Grazie a questa variabile sarà possibile appunto definire i valori corretti di d/f .

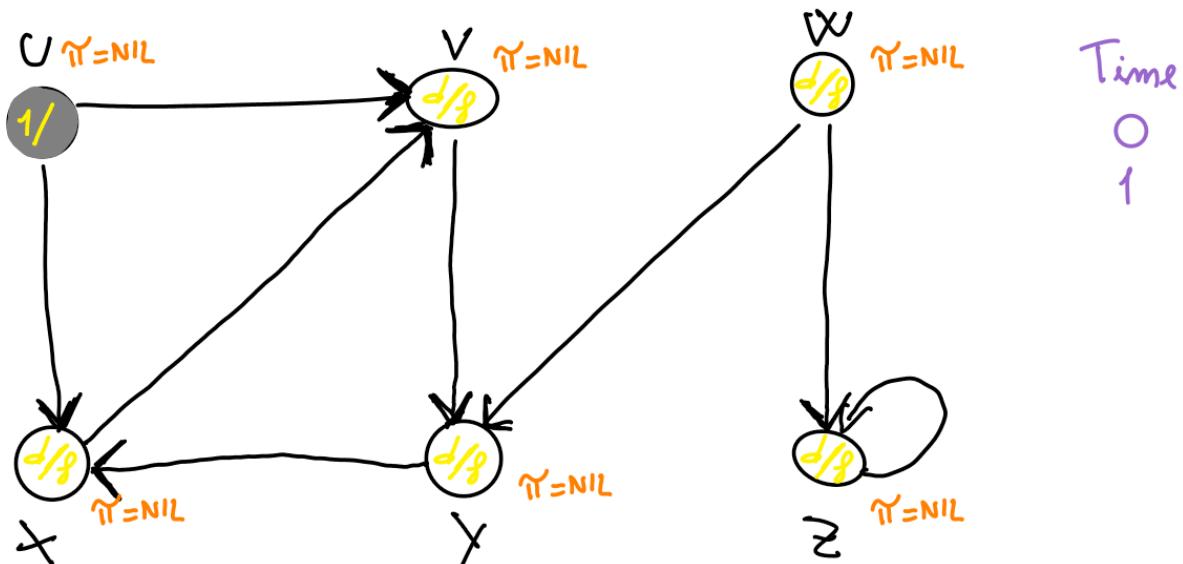
La visita è una **funzione ricorsiva**. Se il nodo ha nodi adiacenti che sono **BIANCHI**, allora procedo con visita, sennò la visita termina e il nodo diventa nero perché non ha adiacenti.

Procedimento

Proviamo un esempio con un grafo diretto (orientato) e facciamo le adeguate inizializzazioni di d/f che teoricamente non hanno un valore iniziale visto che l'algoritmo non è iniziato e π che rappresenta il **predecessore del nodo considerato**.

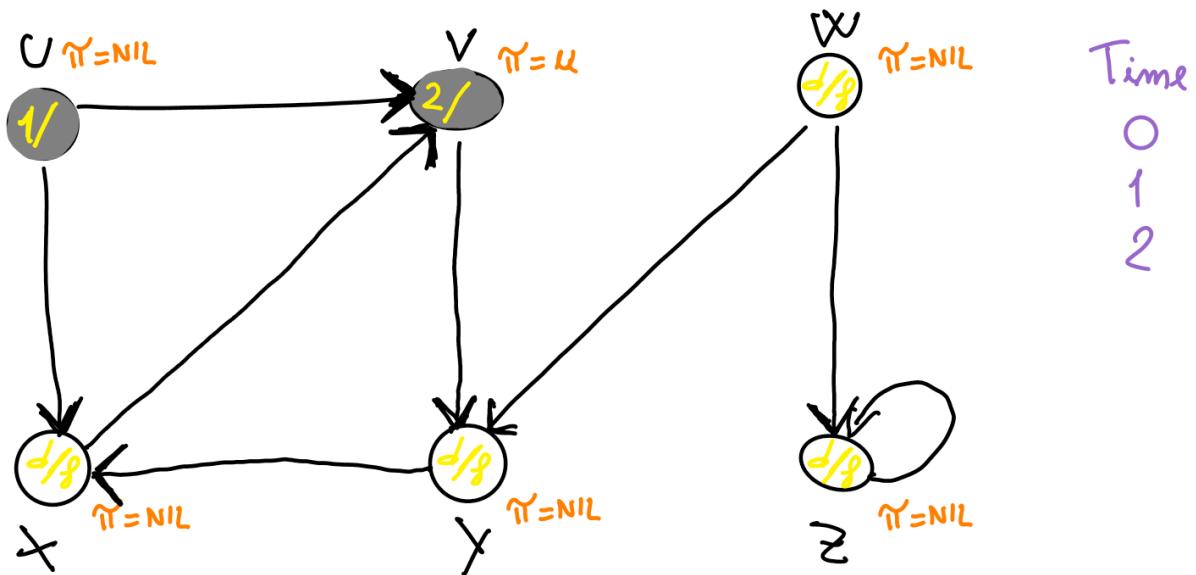


1. In questo tipo di visita non si ha un "nodo sorgente" ma viene semplicemente considerato tutto il grafo.
2. Quindi partiamo dal nodo u . Visito u e quindi lo faccio diventare **GRIGIO**.
3. Incremento il *time* che diventa 1.
4. Discovery del nodo u diventa 1, appunto.

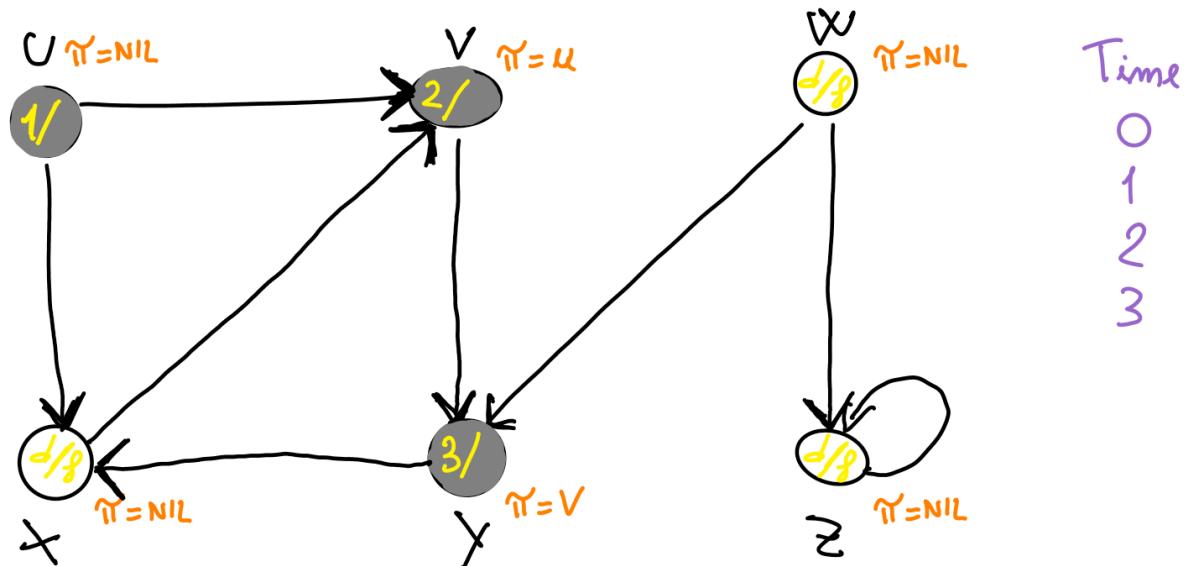


5. Prendo la **lista di adiacenza** del nodo che sto considerando, quindi di u . Ho quindi che questa lista di adiacenza che è così fatta $u \rightarrow x \rightarrow v$. Quindi x e v sono i nodi adiacenti a u .
6. Controllo i colori dei **nodi adiacenti** nella lista di adiacenza di u e vedo che sono **BIANCHI**;
7. Quindi ci si pone **la domanda: Il nodo preso in considerazione ha nodi adiacenti BIANCHI?**

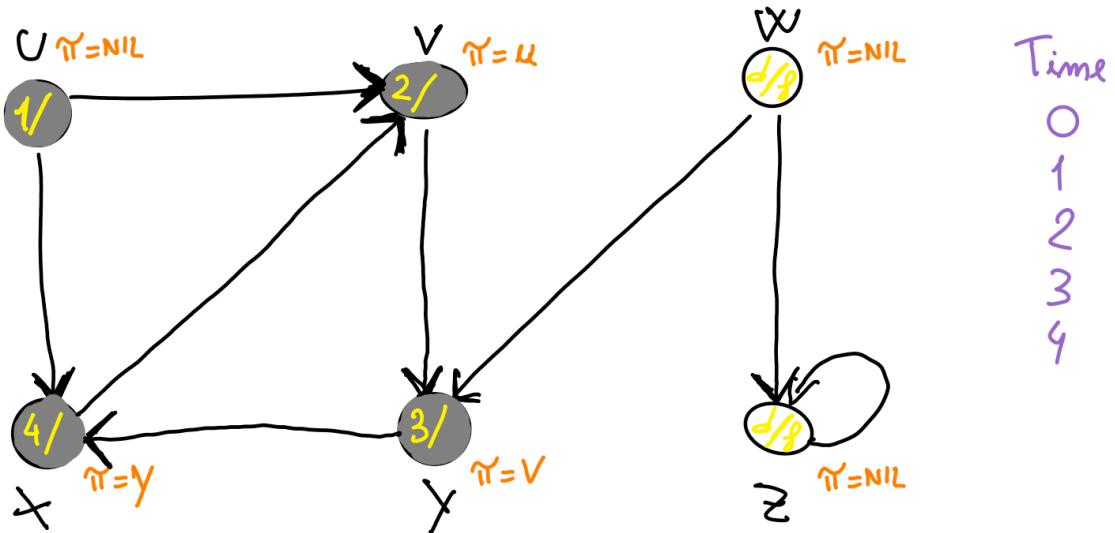
8. Allora posso **impostare il predecessore** al nodo che sto visitando (u).
9. Posso scegliere UN **POSSIBILE NODO BIANCO FRA QUELLI DISPONIBILI.**
10. Quindi il $\pi(v) = u$. (tradotto vuol dire che "il predecessore di v è u ");
11. Incremento il *time*, quindi diventa 2;
12. Faccio diventare **GRIGIO** il nodo v . Il suo discovery time sarà 2;



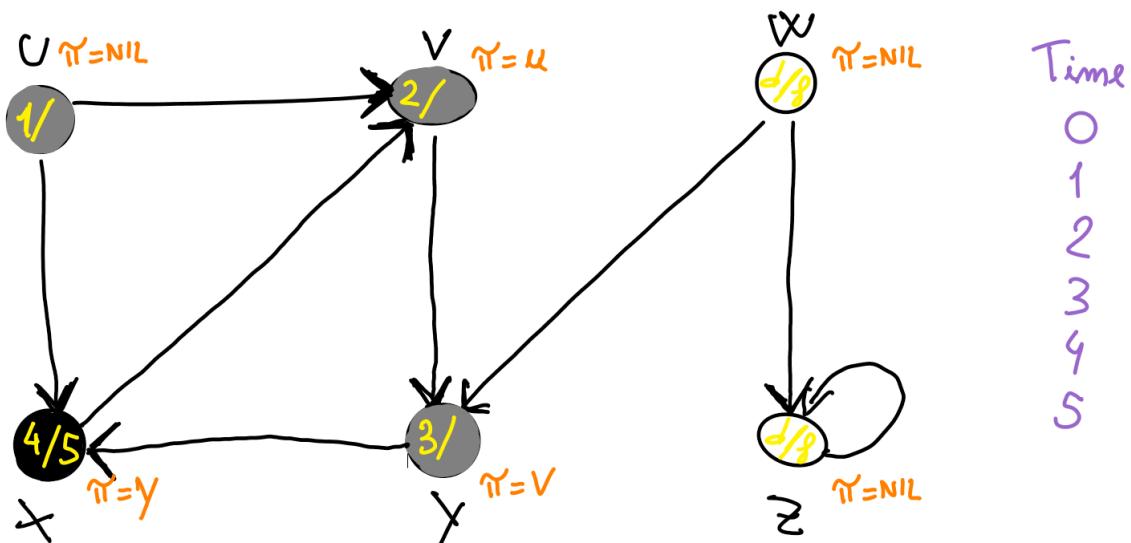
12. v ha nodi adiacenti BIANCHI? Sì, ed è y ;
13. Incremento il *time*, quindi diventa 3;
14. Imposto il discovery di y a 3 e lo visito, quindi diventa **GRIGIO**;



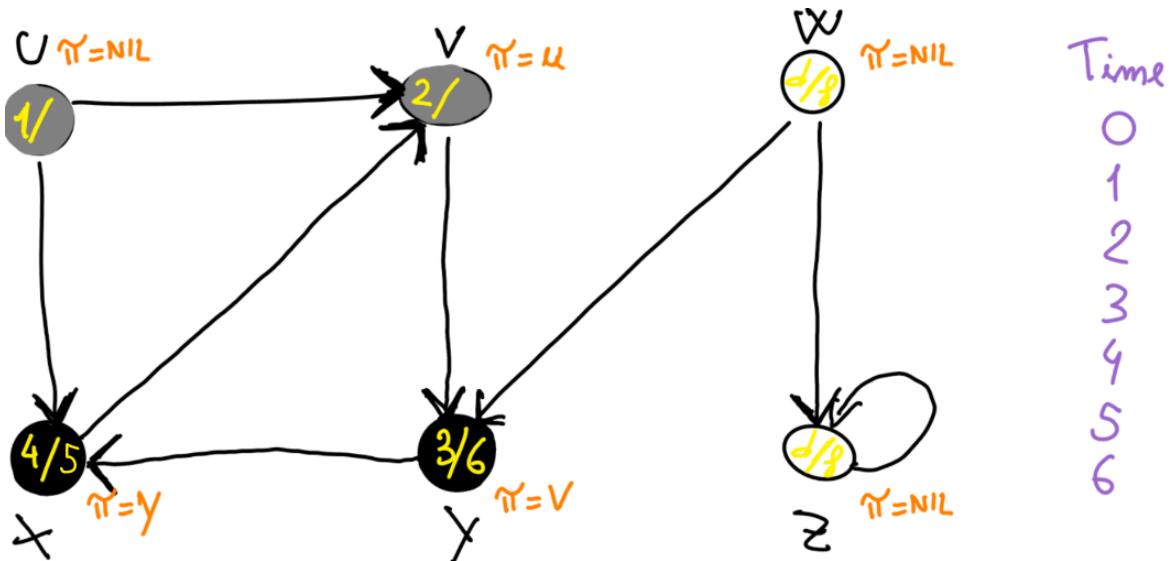
15. y ha nodi adiacenti BIANCHI? Sì, ed è x .
16. Incremento il *time*, quindi diventa 4, lo faccio diventare **GRIGIO**, lo visito e il tempo di discovery di x diventa proprio 4 e imposto il $\pi(x) = y$, ovvero il predecessore di x è y .



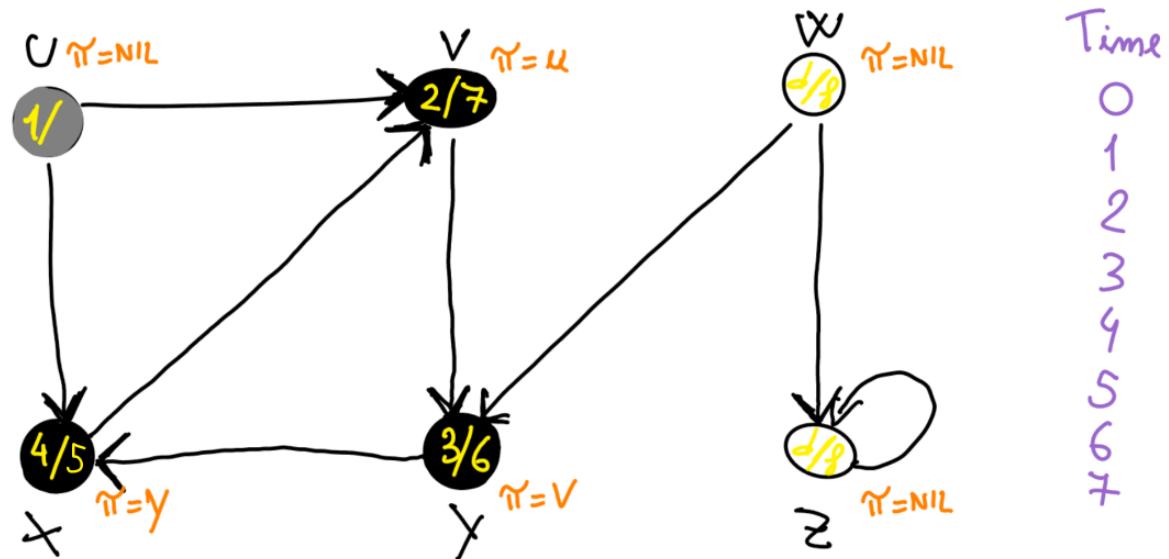
17. x ha nodi adiacenti BIANCHI? No;
18. Incremento il *time*, quindi diventa 5 e sarà il tempo di *fine scoperta* di *x*, quindi 5. Di conseguenza il nodo *x* diventa **NERO**;



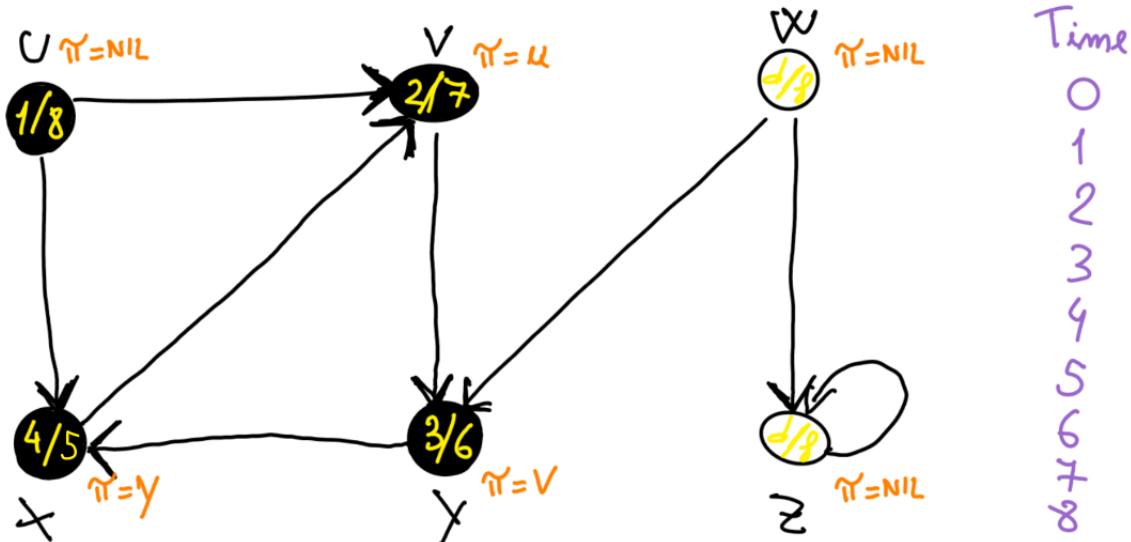
19. Adesso si torna indietro mediante i predecessori π .
Chi è il predecessore di *x*? E' *y*.
20. Quindi torno indietro su *y*
21. Incremento il *time*, quindi diventa 6 e mi chiedo "Questo nodo ha ALTRI nodi adiacenti BIANCHI?"
22. *y* non ha altri nodi adiacenti BIANCHI.
23. Allora il tempo di *fine scoperta* di *y* è 6 e lo faccio diventare **NERO**.



24. Torno al $\pi(y) = v$ quindi su v
25. Incremento il *time*, quindi diventa 7.
26. v ha altri nodi BIANCHI? No. Quindi può diventare **NERO** e il suo tempo di *fine scoperta* è 7.

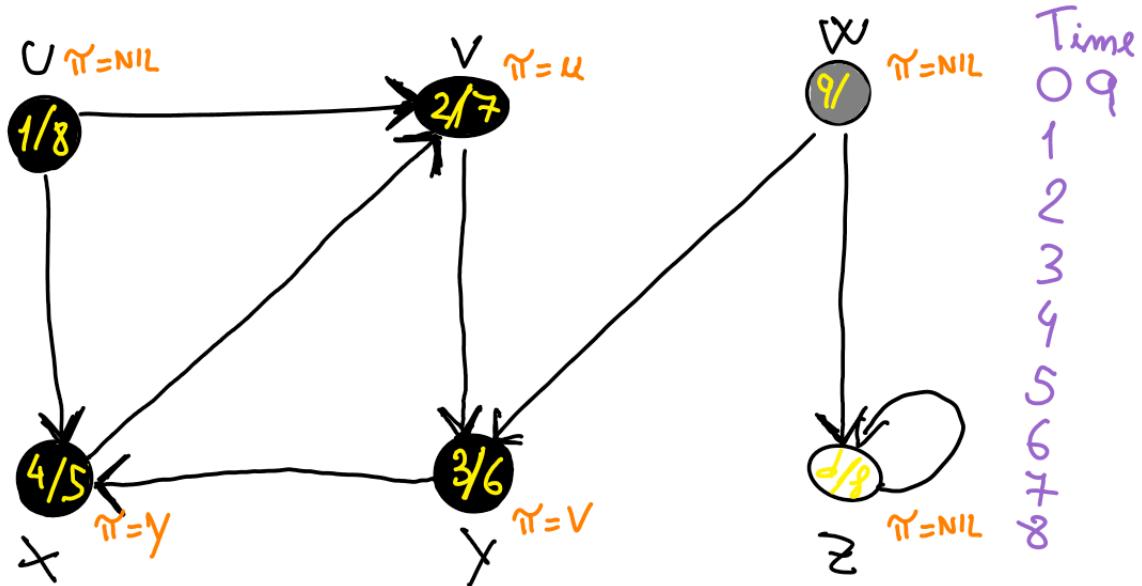


27. Torno al $\pi(v) = u$, quindi su u .
28. Incremento il *time*, quindi diventa 8;
29. u ha altri nodi BIANCHI? No. Quindi può diventare **NERO** e il suo tempo di *fine scoperta* è 8
30. Torno su $\pi(u) = NIL$. Quindi l'algoritmo, per la prima parte del grafo, finisce.

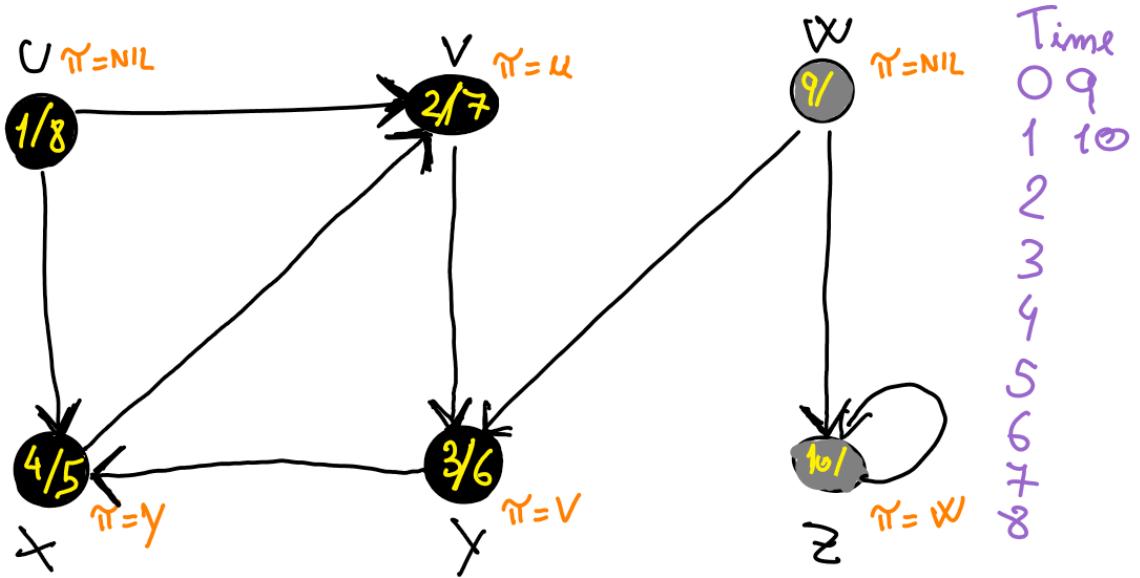


Non tutti i nodi sono raggiungibili con questo algoritmo, quindi devo procedere con l'altra parte del grafo.

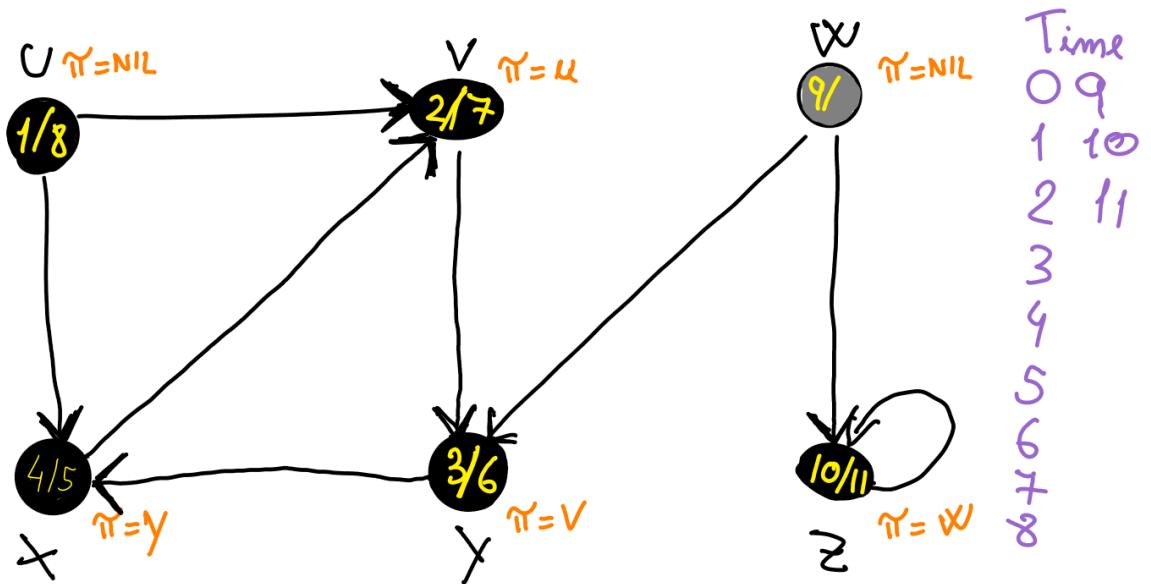
1. Adesso procedo con l'altro lato del grafo.
2. Prendo in considerazione il nodo w , lo visito e lo faccio diventare GRIGIO.
3. Incremento il *time*, quindi diventa 9.



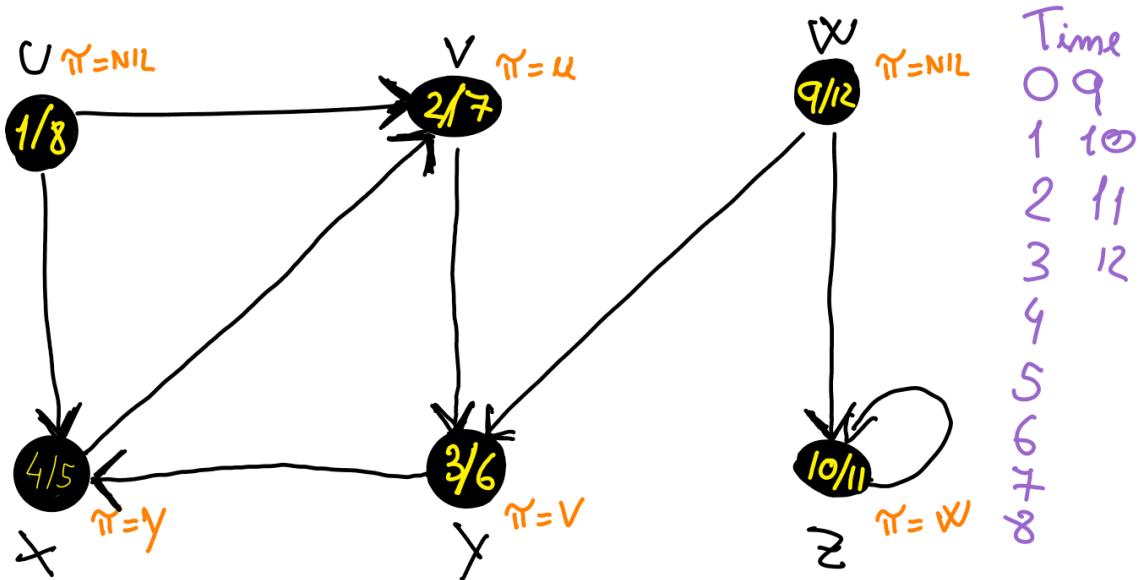
4. w ha nodi adiacenti BIANCHI? Sì, ed è z , lo visito, lo faccio diventare GRIGIO e imposto il $\pi(z) = w$, ovvero il predecessore;
5. Incremento il *time*, quindi diventa 10 e il tempo di discovery di z è 10.



6. z ha nodi adiacenti BIANCHI? No, quindi può diventare NERO.
 7. Incremento il time, quindi diventa 11 e il tempo di fine scoperta di z è 11.



8. Torno al predecessore mediante $\pi(z) = w$, quindi torno a w ;
 9. Incremento il time, quindi diventa 12;
 10. w ha altri nodi adiacenti BIANCHI? No, allora assegno il tempo di fine scoperta a 12 e diventa NERO.



11. Torno al predecessore ma $\pi(w) = NIL$ quindi l'algoritmo finisce.

Come ci si sposta dall'altra parte del grafo?

E' da considerare che tutti i **nodi presenti nel grafo** sono in una **lista di nodi**. Quindi automaticamente ci si sposta al prossimo nodo finché i nodi del grafo non terminano.

La visita **DEPTH FIRST SEARCH** produce la **"foresta dei predecessori"** e si può determinare un **Ordinamento topologico del grafo**.

Ordinamento topologico

Ordinamento su grafo orientato aciclico. Si fa un ordinamento **in modo decrescente** rispetto al **tempo finale** della visita di ogni vertice.

Pseudo codice BFS e complessità

Per quanto riguarda la BFS si distinguono due funzioni: **inizializzazione** e **visita**:

```

INIZIALIZZAZIONE
BFS(G, s) (Grafo, nodo "s"orgente)

for ogni vertice u nel grafo G:
    color[u]=white
    d[u]=inf (tempo scoperta di u)
    p[u]=null predecessore di u

color[s]=gray
p[s]=null
d[s]=0

```

```

VISITA
Q = \ coda vuota
Q.enqueue(s)
while Q!=\ //diversa da vuoto --> O(|V|) -> al max |V|
    u=Q.dequeue();
    for ogni vertice v in adj[u] // al max |E| -->O(|E|) perchè dipende dal numero di adiacenti che puo avere un grafo
        if(color[v]==white)
            color[v]=gray
            d[v]=d[u]+1
            p[v]=u
            Q.enqueue(v)
    color[u]=black

```

Osservazioni sulla complessità

La **complessità di visita** è **MOLTO MAGGIORE** rispetto alla **complessità inizializzazione**;

- Complessità inizializzazione $\rightarrow \Theta(|V|)$
 - Complessità di visita $\rightarrow O(|E|)$
- COMPLESSITA' BFS TOTALE $\rightarrow O(|V|) + O(|E|)$**

Pseudo codice DFS e complessità

Anche nel caso dell'algoritmo DFS si distinguono due funzioni: **inizializzazione e visita**:

```
INIZIALIZZAZIONE
DFS(g)
    for ogni vertice u in G: |V| volte
        color[u]=white
        d[u]=0; discovery
        f[u]=inf finish
        p[u]=null
    global time=0
    for ogni vertice u in G.
        if(color[u]==white)
            DFS-visit(u)
```

```
VISITA
DFS-visit(u)
    color[u]=gray
    time=time+1 //globale
    d[u]=time
    for v in adj[u] |E| volte
        if color[v]==white
            p[v]=u
            DFS-visit(v)
    // nodo non ha piu nodi adiacenti bianchi, allora:
    color[u]=black
    time=time+1
    f[u]=time
```

Osservazioni sulla complessità DFS

- Complessità inizializzazione $\rightarrow O(|V|)$
- Complessità visita $\rightarrow O(|E|)$

COMPLESSITA' TOTALE DFS $\rightarrow O(|V| + |E|)$

ESEMPI DOMANDE ESAME

1. Qual è la complessità del seguente algoritmo?
2. Qual è l'ordine di grandezza della funzione $G=T1*T2$?
3. Quale riga genera un errore di compilazione?
4. Qual è l'output del seguente codice?
5. Considerando il seguente BST, indicare quale sarebbe il padre del nodo X dopo la rimozione del nodo Y..
6. Quando si dice che una classe è astratta?
7. Cosa intendiamo dire quando diciamo che un algoritmo X è asintoticamente più efficiente di un altro algoritmo Y?

PROVA PRATICA: Come mi esercito?

- Esercizi svolti in aula e possibili varianti;
- Esercizi elencati alla fine di alcuni blocchi di slides;
- Implementazioni suggerite nelle slides;
- Esercizi proposti nei libri di testo

Esempi:

1. Definire una funzione membro della classe albero per visualizzare tutti i nodi di un albero binario di ricerca il cui valore del campo chiave sia maggiore di un valore accettato in input.
2. Definire una classe albero che restituisca una istanza della classe albero che rappresenta il simmetrico dell'oggetto albero.
Per simmetrico intendiamo un albero in cui ogni nodo ha sottoalberi sinistro e destro invertiti.