

Programmazione dinamica

RICORDA: la programmazione dinamica è applicabile solo a quei problemi che presentano una sottostruttura ottima

Divide et impera | | | | | Programmazione dinamica

DIVIDE ET IMPERA == Applicabile a tutti i tipi di problemi;

- sottoproblemi indipendenti
- top-down
- ricorsione

PROGRAMMAZIONE DINAMICA == Solo problemi di ottimizzazione

- Il numero di sottoproblemi è limitato e sono “condivisi”
- Bottom-up
- iterazione

Programmazione dinamica come risoluzione all'eccessivo uso di memoria della ricorsione

- In aggiunta, permette di memorizzare risultati a sottoproblemi (*così non devo ricalcolare sottoproblemi che ho già risolto mentre ricorro*)

Due diverse tipologie di programmazione dinamica:

- 1) **Bottom-up**: al contrario del *divide et impera* che è top-down, il bottom-up elabora una soluzione partendo dal basso (risultato nullo) e costruisce il risultato man mano -> è applicabile ai problemi di ottimizzazione, ma solo se il problema ha una struttura ottima.
 - *L'approccio bottom-up parte dai problemi elementari e costruisce una soluzione ottima ad ogni passo*
- 2) **Top-down**: simile divide et impera, procede in modo ricorsivo utilizzando la **memoization** - memorizza infatti i risultati dei sottoproblemi risolti e li riutilizza per ridurre la complessità e costo di memoria dell'analisi degli altri sottoproblemi

Come si applica la programmazione dinamica, e quando?

- 1) **Verifica applicabilità** > il problema deve avere una struttura ottima – e quindi anche una sottostruttura. Ad ogni passo dovrò prendere la scelta migliore per quel determinato sottoproblema.

- *Non posso avere una soluzione ottima se la soluzione ai suoi sottoproblemi non è ottima >> ad ogni passo dovrò prendere la soluzione migliore – in questo caso sarà quella minima*
- *Per assurdo, se un valore preso non fosse minimo, allora la soluzione globale non sarebbe minima, e quindi non avrei una soluzione ottima (assurdo, tesi afferma che invece la soluzione è ottima)*
- L'esistenza di una sottostruttura ottima e la dimostrazione che esiste si basa sul **fare una scelta** > questa scelta porta ad una soluzione ottima; non importa sapere come è stata fatta la scelta, ma importa sapere quali sottoproblemi ne derivano e quale sia il modo migliore per caratterizzare lo spazio di sottoproblemi risultante.
- *Un modo di dimostrare per assurdo la sottostruttura ottima di un problema è “incollando” un valore non ottimo in una soluzione ottima >> genereremo così una contraddizione con al tesi, ovvero che la soluzione attuale è ottima.*

2) Ispirazione > calcolo soluzione con metodo ricorsivo e poi la uso come ispirazione per costruire la mia soluzione bottom-up oppure con memoization

3) Costruzione della soluzione

Es. problema catena di montaggio >

- Con soluzione ricorsiva >> $O(2^n)$
- Soluzione bottom-up >> $O(n)$

Catena di montaggio formata da due sequenze di passi, ognuno con un tempo di elaborazione ed un tempo di trasferimento – sulla stessa catena il tempo di trasferimento è nullo . **devi calcolare il percorso con tempo minimo** per arrivare alla fine della catena

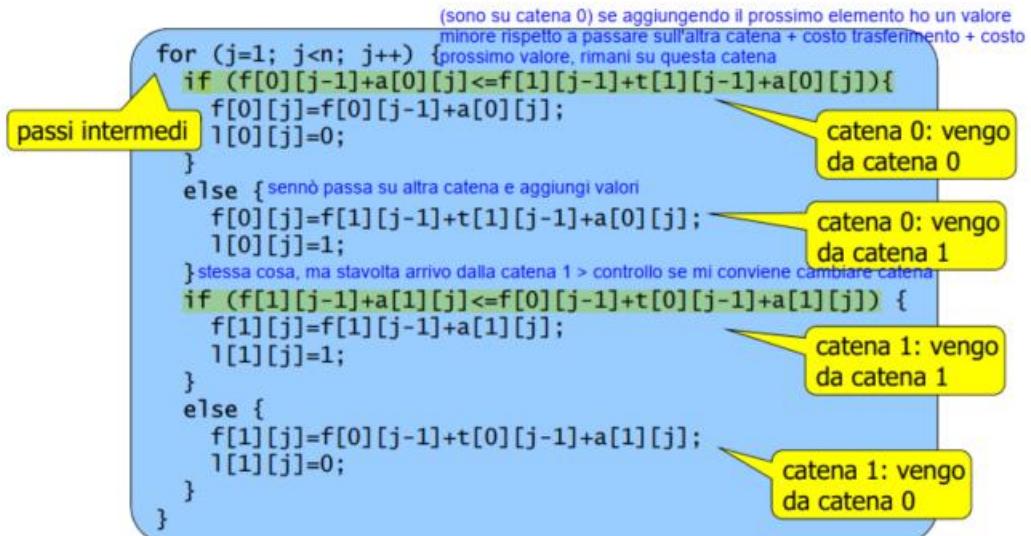
Con soluzione ricorsiva, devo calcolare tutte le possibili soluzioni e poi scegliere quella minima >>

- Principio moltiplicazione > $O(2^n)$

$O(2^n)$ non è accettabile come soluzione -> dobbiamo usare programmazione dinamica, ma come?

Partendo da una soluzione vuota, aggiungo ogni volta la soluzione migliore > **ovvero la soluzione che, aggiunta alla soluzione attuale, permette di avere una soluzione ottima.**

(in questo caso ad ogni passo prenderò l'elemento che mi garantirà il costo minore tra quelli che sono disponibili in quel determinato passo)



esempio di soluzione con calcolo bottom

Così facendo, non devo calcolare tutte le possibili soluzioni, ma costruisco e calcolo una sola soluzione, quella direttamente **ottima** ed in grado di risolvere il problema in modo ottimale. In questo caso specifico, riesco a raggiungere una complessità **O(N)** (*itero una sola volta sul numero totale di elementi*)

Complessità – programmazione dinamica

Genericamente, $T(N) = O(n \text{ sottoproblemi} * \text{costo ricombinazione soluzioni ottime})$

- *In scorso esempio, N problemi con costo di ricombinazione unitario*

Prodotto di due matrici

Due matrici sono compatibili solo se il numero di righe di una è uguale al numero di colonne dell'altra.

- Algoritmo semplice risolve il problema con algoritmo di complessità **O(n^3)**

Cosa succede nel caso dovessi moltiplicare in catena una serie N di matrici, compatibili tra loro a due a due?

Devo stabilire un ordine secondo cui vanno moltiplicate le matrici >> se voglio ottenere un costo minimo, devo tenere conto di tutte le **parentesizzazioni possibili** > **ordine in cui vengono moltiplicate l'una con l'altra**

Perché è importante l'ordine?

- Immaginiamo 3 matrici, una $A = 10 \times 100$, $B = 100 \times 5$, $C = 5 \times 50$
- L'ordine in cui le moltiplichiamo stabilisce anche il numero di moltiplicazioni scalari che vengono svolte

Esaminando due possibili parentesizzazioni...

- $(A*B)*C >> A*B = 10 \times 100 \times 5$ operazioni > matrice risultante è 10×5 , con 5000 operazioni svolte
 - $(A*B) * C = 10 \times 5 \times 50 > 10 \times 5 \times 50$ operazioni, matrice risultante è 10×50 con 2500 operazioni svolte
 - Totale operazioni svolte = **7500**
- $A*(B*C) >> B*C = 100 \times 5 \times 50 >$ matrice risultante è 100×50 , con 25.000 operazioni svolte
 - $A*(B*C) = 10 \times 100 \times 50 >$ matrice risultante è 10×50 con 50.000 operazioni svolte
 - Totale operazioni svolte = **75.000**

Come stabilisco il numero di parentesizzazioni?

Divido in due la catena >> il numero totale di parentesizzazioni sarà il prodotto tra il numero di parentesizzazioni delle due catene

Dato n quindi dimensione dell'insieme di partenza

Si dimostra che $P(n) = C(n-1)$
dove $C(n)$ è detto numero di Catalan e vale

$$C(n) = \frac{1}{(n+1)} \binom{2n}{n} = \Omega(4^n / n^{3/2})$$

Il costo totale quindi di un'operazione su un insieme, o una matrice, diventa uguale al costo totale dei suoi sottoproblemi, che avranno ognuno soluzione ottima.



Per la parentesizzazione, divido il prodotto di una serie di matrici come diversi sottoproblemi di moltiplicazioni tra matrici >> ovvero, ad ogni passo moltiplico la matrice **attuale** con quella che mi dà costo **minimo** >> segno quindi la scelta in un eventuale vettore, e ho così l'ordine di parentesizzazione del prodotto tra matrici.

Ottengo così una complessità $T(n) = O(n^3)$, rispetto ad un $O(2^n)$ esponenziale.

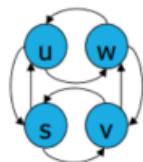
Nel caso venga chiesto orale, ripassarlo – forse più importante LIS però

Altri esempi di problemi **affrontabili (?)** con la programmazione dinamica

1) **Cammini minimi** >> cammini minimi sono cammini *semplici* (no cicli)

- Caso elementare >> due vertici coincidono
- Cerca cammino minimo da $u \rightarrow v$:
 - Trovo un nodo intermedio che mi permette di raggiungere $u \rightarrow w, w \rightarrow v$
 - $U \rightarrow w$ e $w \rightarrow v$ saranno ottimi, poiché se non lo fossero allora esisterebbe un altro nodo o arco che permette di raggiungere v con un cammino minore, e avrei quindi contraddetto la tesi.
- Ne risulta quindi che in un cammino minimo, anche i sottocammini (cammini da un vertice all'altro) devono essere minimi.

2) **Cammini massimi** >> trovare cammino semplice massimo tra $u \rightarrow v$



- **U->w->v** è cammino semplice massimo
 - ma sottocammino $u \rightarrow w$ non è massimo >> il massimo è $u \rightarrow s \rightarrow v \rightarrow w$
 - Idem per $w \rightarrow v$ con $w \rightarrow u \rightarrow s \rightarrow v$
 - Ho quindi un cammino i cui sottoproblemi non hanno soluzione ottimale > la soluzione trovata non sarà quindi ottimale e la **programmazione dinamica non sarà applicabile in questo caso. (oltre al fatto che il cammino che ne risulterebbe considerando eventuali valori ottimi non è semplice)**

!!! Longest Increasing Sequence !!!

arr[]	10	22	9	33	21	50	41	60	80
LIS	1	2		3		4		5	6

Si definisce **sottosequenza** di X di lunghezza k ($k \leq N$) un qualsiasi sottoinsieme di k elementi, con indici crescenti e *non necessariamente contigui*.

- **Prefisso** > prefisso i di una sequenza >> elementi da posizione 0 a posizione $i-1$

Una **LIS** è infatti la sottosequenza con lunghezza massima di un insieme di elementi, ordinati per valore crescente.

Approcci

- Calcolo combinatorio > $O(2^N)$ > eccessivo
- Programmazione dinamica >> sottostruttura è ottima
 - Data una soluzione ottima, se non lo fosse per ogni prefisso, se ne potrebbe trovare una migliore e di conseguenza la soluzione ottima non sarebbe tale.
 - Il numero massimo di sottoproblemi è $N > O(N)$, dimensione dei dati in ingresso
 - Soluzione di ogni sottoproblema è $O(N)$
 - **Complessità totale == $O(N^2)$**
 - Ci sono anche soluzioni $O(n \log N)$

La funzione principale calcolerà infatti la lunghezza massima della LIS

Una volta stabilita l'applicabilità, calcolo la soluzione con il metodo ricorsivo >

- Vettore $C[]$ tiene conto del valore della LIS
- Caso elementare $i = 0$, dimensione sottoinsieme = 0, ritorno 1
- Altri casi >> $0 < i < N \rightarrow$ considero tutti i prefissi X_j dell'intervallo $0 \leq j < i$ che soddisfano la condizione $X_j < X_i$
 - Aggiungo poi X_i alla LIS attuale >> aumento di 1 il valore di $C[X_j]$

wrapper

```

int LIS(int *val) {
    return LISR(val, N-1);
}

int LISR(int *val, int i) {
    int j, ris;
    if (i == 0) → c[X0]=1
    return 1;
    ris = 1;
    for (j=0; j < i; j++)
        if (val[j] < val[i])
            ris = max(ris, 1 + LISR(val, j));
    return ris;
}

```

approccio ricorsivo

Approccio iterativo – bottom up

- Vettore val
 - Vettore L > vettore di N interi per memorizzare la lunghezza della LIS per ogni prefisso i -esimo
 - Vettore P > vettore di N interi per memorizzare l'indice dell'elemento precedente nella LIS
 - $Last$ per memorizzare l'indice dell'ultimo elemento della LIS

Ad ogni passo quindi, dato l'elemento corrente i compreso tra 0 e N-1 >>

- Individuo la LIS del prefisso i
- Registro in P per i corrente l'elemento precedente

<https://www.youtube.com/watch?v=Ns4LCeeOFS4>

Esempio

val L P i=0 $c[X_0] = 1$	val L P i=3 $c[X_3] = 3$
val L P i=1 $c[X_1] = 1$	val L P i=4 $c[X_4] = 3$
val L P i=2 $c[X_2] = 2$	val L P i=5 $c[X_5] = 3$

13 può avere come valori precedenti sia 11 che 7 > controllo i due casi e vedo dove ottengo il valore massimo che so già di dover cercare

```

void LISDP(int *val) {
    int i, j, ris=1, L[N], P[N], last=1;
    L[0] = 1; P[0] = -1;
    for (i=1; i<N; i++) {
        L[i] = 1; P[i] = -1;
        for (j=0; j<i; j++)
            if ((val[j] < val[i]) && (L[i] < 1 + L[j])) {
                L[i] = 1 + L[j]; P[i] = j;
            }
        if (ris < L[i]) {
            ris = L[i]; last = i;
        }
    }
    printf("One of the Longest Increasing Sequences is ");
    LISprint(val, P, last);
    printf("and its length is %d\n", ris);
}
  
```

```

void LISprint(int *val, int *P, int i) {
    if (P[i]==-1) {
        printf("%d ", val[i]);
        return;
    }
    LISprint(val, P, P[i]);
    printf("%d ", val[i]);
}
  
```

stampa inversa - parto dall'ultimo, quindi li stampa in ordine

Lo zaino (discreto)

Dato un insieme di N oggetti ciascuno con un peso W_j e un valore V_j – un peso massimo cap , determinare il sottoinsieme S di N tale che:

- Somma dei pesi $\leq cap$
- Valore degli oggetti MASSIMO (*problema di ottimizzazione*)
- Ogni oggetto è preso (1) o lasciato (0)

- 1) Problema ha sottostruttura ottima?

Se il problema è di trovare un valore possibile con una capacità massima CAP, e la soluzione finale è un insieme di singoli oggetti, posso scomporre il problema in sottoproblemi semplici, riguardn i singoli oggetti
=>

- Controllo se l'ultimo oggetto preso appartiene o no alla soluzione (rientra nella soluzione, peso tot $\leq cap$)

Ho **2** problemi da risolvere

- **Calcolo del valore massimo ottenibile >> ottimizzazione**
- **Ricerca degli elementi >> soluzione dinamica**

Ottimizzazione – calcolo del valore massimo ottenibile

Utilizzo il calcolo combinatorio, valuto tutte le possibili combinazioni e scelte di oggetti:

- Caso elementare >> non rimangono oggetti / il peso totale è stato raggiunto
- **Caso ricorsivo** >> l'oggetto i-esimo non eccede il peso totale ||| non conviene prendere l'oggetto i-esimo

Quando conviene? >> **Valutazione di convenienza**

Se, a parità di oggetti, aggiungere il prossimo oggetto non permette di superare il massimo corrente ottenuto con quel numero di oggetti, allora non lo si considera.

- **Ovvero, se è possibile prendere l'oggetto, prenderlo se migliora il valore rispetto al non prenderlo**

```
int maxValR(Item *items, int i, int j) {  
    if ((i < 0) || (j == 0))  
        return 0;  
    if (items[i].size >j) se oggetto ha peso superiore a max  
        return maxValR(items, i-1, j); prendi il massimo tra il caso in cui considero l'elemento  
    else return max(maxValR(items, i-1, j), nuovo e quello in cui non lo considero  
                    maxValR(items, i-1, j-items[i].size)+items[i].value);  
}  
  
int KNPmaxval(Item *items, int N, int cap) {  
    return maxValR(items, N, cap);  
}
```

Soluzione dinamica – BOTTOM-UP

- Vettore *items* di N elementi
- Matrice *maxVal[0..N, 0..cap]* per memorizzare i valori ed identificare il massimo

Esempio

	0	1	2	3	N = 4
valore	10	6	8	9	oggetti 1,2,3 4
peso	8	4	2	3	cap = 10
oggetto	1	2	3	4	

	j										
0	0	0	0	0	0	0	0	0	0	0	oggetto fittizio
1	0	0	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	
3	0	0	0	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	0	0	
	zaino fittizio	maxval									

Passi:

- 2 cicli for() – uno passa su tutti gli oggetti, uno da 1 a cap
- Oggetto corrente è *items[i-1]* >> se il suo peso eccede cap, non si prende e si mette *maxval[i,j] = maxval[i-1,j]*
- Se invece non è maggiore, valuto se **conviene prenderlo** >> per *i* corrente, se sommando il valore dell'oggetto corrente al totale corrente, ottengo un valore maggiore rispetto al massimo con *i* oggetti, allora prendo l'oggetto in posizione *i* e aggiorno il massimo totale.

Se *maxval[i-1,j] ≥ maxval[i-1][j-items[i-1].size] + items[i-1].value*

- non prendo l'oggetto *i* e *maxval[i,j] = maxval[i-1,j]*
- altrimenti prendo l'oggetto *i* e

$$\maxval[i,j] = \maxval[i-1][j-items[i-1].size] + \text{items}[i-1].value$$

```
int KnapmaxValDP(Item *items, int N, int cap) {
    int i, j, **maxval;
    // allocazione matrice maxval di dimensioni (N+1)x(cap+1)
    for (i=1; i<=N; i++)
        for (j=1; j <=cap; j++) {
            if (items[i-1].size > j)
                maxval[i][j] = maxval[i-1][j]; calcolo del valore massimo e di una soluzione ottima
            else
                maxval[i][j] = max(maxval[i-1][j],
se non posso prenderlo, tengo max precedente
sennò calcolo se mi conviene maxval[i-1][j-items[i-1].size] +
> se sì lo prendo items[i-1].value);
        }
}
```

Ottengo una matrice con tutti i possibili valori e combinazioni per ogni oggetto.

Ogni riga è infatti una possibile soluzione.

	0	1	2	3							
valore	10	6	8	9							
peso	8	4	2	3							
oggetto	1	2	3	4							
					j						

maxval =	0	0	0	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0	0	10	10	10
	0	0	0	0	6	6	6	6	10	10	10
	0	0	8	8	8	8	14	14	14	14	18
	0	0	8	9	9	17	17	17	17	23	23

Una volta ottenuta questa disposizione, itero partendo da $\text{maxVal}[N][\text{cap}]$

- Se $\text{max}[i][j]$ è uguale a quella che non la include $\text{max}[i-1][j]$, l'oggetto corrente non è stato preso, quindi $\text{sol}[i-1] = 0$
- L'oggetto è stato preso, quindi $\text{sol}[i-1] = 1$ e capacità rimanente $j = j - \text{items}[i-1].size$

Memoization

- Approccio simile al divide et impera della ricorsione classico, ma con memorizzazione dei valori raccolti finora
- Approccio top-down (anche chiamata *programmazione dinamica top-down*)

Esempio: numeri di Fibonacci
array knownF
passo un vettore in cui metto i numeri di fibonacci già calcolati - così, se sto risolvendo un problema per cui so già la soluzione, risparmio tempo e memoria e lo riutilizzo

```
unsigned long fib(int n, unsigned long *knownF) {
    unsigned long t;
    if (knownF[n] != -1)
        return knownF[n];
    if(n == 0 || n == 1) {
        knownF[n] = n; registro la soluzione
        return n;
    }
    t = fib(n-2, knownF) + fib(n-1, knownF);
    knownF[n] = t;
    return t;
}
```

La corrispondenza n - indice vettore funziona poiché n è l' n -esimo numero di fibonacci

Come funziona? Una volta arrivato all'ultima chiamata ricorsiva, mentre *risale* controlla se non ha già quei valori >> se li ha, li utilizza

Esempio >> Il problema dello zaino

Uguale a prima, ma questa volta ogni oggetto si ripete k volte.

Principio moltiplicazione >> creiamo in aggiunta un vettore *maxknown*, in cui memorizzo per ogni indice l'eventuale soluzione per ogni capacità k

```

int KNAPmaxValR(Item *items, int N, int cap) {
    int i, space, max, t;
    for (i = 0, max = 0; i < N; i++)
        if ((space = cap-items[i].size) >= 0)
            if ((t=KNAPmaxValR(items,N,space)+items[i].value)>max)
                max = t;
    return max;
}

```

ricorsivo



09knapsackM

Creiamo un vettore, maxknown, in cui memorizzo per ogni indice l'eventuale soluzione per ogni capacità k

```

int KNAPmaxValM(Item *items,int N,int cap,int *maxKnown){
    int i, space, max, t;
    if (maxKnown[cap] != -1)
        return maxKnown[cap];
    for (i=0, max=0; i < N; i++)
        if ((space = cap-items[i].size) >= 0)
            if ((t=KNAPmaxValM(items,N,space,maxKnown)+
                 items[i].value)>max)
                max = t;
    maxKnown[cap] = max;
    return max;
}

```

Paradigma greedy

Alternativa alla *programmazione dinamica e divide et impera*

- Più rapido e semplice
- ***Non offre sempre soluzione ottima***

Il paradigma greedy si basa sul concetto di fare scelte *localmente ottime* per giungere ad una *soluzione globale ottima*.

- Ogni scelta viene fatta secondo un criterio di **appetibilità** >> quanto mi conviene prendere questo oggetto?
 - La funzione di appetibilità o comunque il criterio, sono diversi per ogni problema. Ad esempio in un caso in cui si ha un insieme di oggetti con un peso ed un valore, andrò a stabilire un grado di *appetibilità* dato dal rapporto peso/valore, e prenderò prima tutti gli oggetti con quel maggiore valore

- Altro esempio, insieme di attività con un inizio ed una fine da ordinare – non devono sovrapporsi – prendo come criterio di appetibilità il tempo di fine degli intervalli, in ordine crescente – così prendo subito il primo intervallo e poi controllo tutti gli altri, prendendo solo quelli compatibili e che non si sovrappongono
- Questo porta ad una soluzione **potenzialmente non ottima, poiché non si garantisce la completa esplorazione dello spazio delle soluzioni.**

Esempio di applicazione del paradigma greedy – Cambiamonete

Ho un cambiamonete che deve, dato un resto, determinare quali monete utilizzare per offrire il resto.

Con un approccio greedy, non utilizzo il calcolo combinatorio, ma uso un algoritmo semplice che esegue una serie di scelte localmente ottime e che porta ad una soluzione ottima.

Come?

Algoritmo che, ad ogni passo, diminuisce il resto totale della moneta più vicina al resto totale.

Monetazione:			Esempio		
25, 10, 5, 1			25, 10, 1		
Resto:			Resto:		
	67			30	
Passo	Resto residuo	Moneta scelta	Passo	Resto residuo	Moneta scelta
0	17	2 x 25	0	5	1x25
1	10	1 x 10	1	0	5x1
2	7	1 x 5			
3	2	2 x 1			

soluzione ottima!

soluzione non ottima!

soluzione ottima:
10,10,10

paradigma greedy non ha fornito soluzione ottima

Seconda figura → esempio di come una scelta localmente ottima non ci ha fornito una soluzione globalmente ottima

Paradigma greedy – problema dello zaino

Problema uguale a quello analizzato per la programmazione dinamica

Un oggetto può essere preso (1) o no (0)

- Stabilisco come criterio di *appetibilità* di un oggetto il rapporto tra il suo *valore / peso*
- **Approccio greedy ==** Scelgo ad ogni passo l'oggetto con appetibilità maggiore e che rientra nel peso disponibile dello zaino.
- **Esito >>** non offre una soluzione globalmente ottima.

Se fosse invece possibile inserire anche una frazione degli elementi presenti nello zaino, allora si potrebbe sfruttare lo spazio restante inserendo altri possibili oggetti e avvicinandosi al valore globalmente ottimo.

- Si ottiene infatti un esito globalmente ottimo.

```

for (i=0; i<n && (KEYgetW(objects[i]) <= res); i++) {
    KEYsetF(&objects[i], 1.0);
    stolen = stolen + KEYgetV(objects[i]);
    res = res - objects[i].weight;
} inserisco tutti gli oggetti possibili
Una volta finito, mi rimarrà l'ultimo oggetto di cui potrò mettere solo una frazione >ottengo la frazione dell'oggetto
dividendo il peso restante per il peso stesso - poi ottengo il valore moltiplicando la frazione per il suo valore
    KEYsetF(&objects[i], res/KEYgetW(objects[i]));
    stolen = stolen + KEYgetF(objects[i])*KEYgetV(objects[i]);

```

Codici di HUFFMAN – APPETIBILITÀ DINAMICA

Finora abbiamo parlato di un'appetibilità statica – che non cambia nel corso dell'algoritmo.

Codice > stringa di bit associata ad un simbolo *S*

Codifica > trasformazione Simbolo > Codice

Decodifica > Codice > simbolo

I codici di Huffman sono utilizzati per la compressione dei dati, indipendentemente dal loro tipo – questo viene fatto comprimendo i dati senza l'uso di prefissi e con codici a lunghezza variabile, risparmiando spazio.

Codici

- **Lunghezza fissa >>** facilità di decodifica, li uso quando ho simboli che compaiono spesso
- **Lunghezza variabile >>** difficoltà di decodifica, ma permettono maggiore risparmio di memoria. *Uso simboli con frequenze diverse >>*
 - Simboli che compaiono più volte verranno codificati con lunghezze brevi
 - Simboli che compaiono poche volte verranno codificati con lunghezze più lunghe

Codici a lunghezza variabile però possono avere il problema della decodifica >> come faccio a decodificare un codice che potrebbe più di un possibile significato?

>>

Codici prefissi

Un codice è libero da prefisso se *nessuna parola di codice valida è prefisso di un'altra parola di codice*.

Codifica >> giustapposizione di stringhe (una stringa messa accanto all'altro)

Decodifica >> percorimento di albero binario

Allo stesso modo si potrà quindi decodificare una stringa -> è soltanto una serie di caratteri ->
la decodifco bit per bit

Per ogni carattere, e quindi ogni serie di bit, e quindi ogni bit, percorro l'albero fino a trovare ogni simbolo corrispondente e decodificarlo

Codifica:

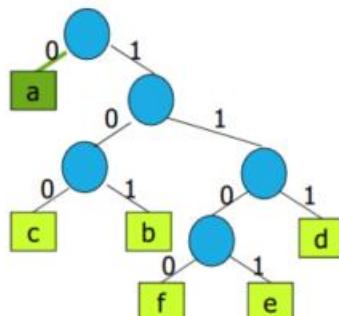
a b f a a c

0101110000100

Decodifica:

0101110000100

a



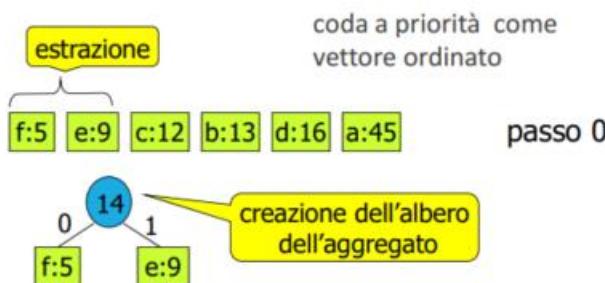
esempio di codifica di "a b f a a c"

Come creare questa struttura ad albero? Coda a priorità

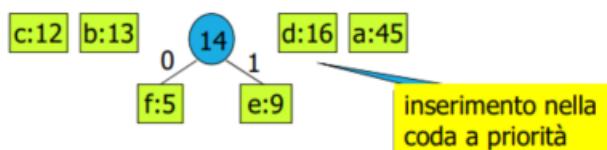
Struttura dati vista in seguito

Inizialmente ho una unica foglia, che corrisponde ad un simbolo.

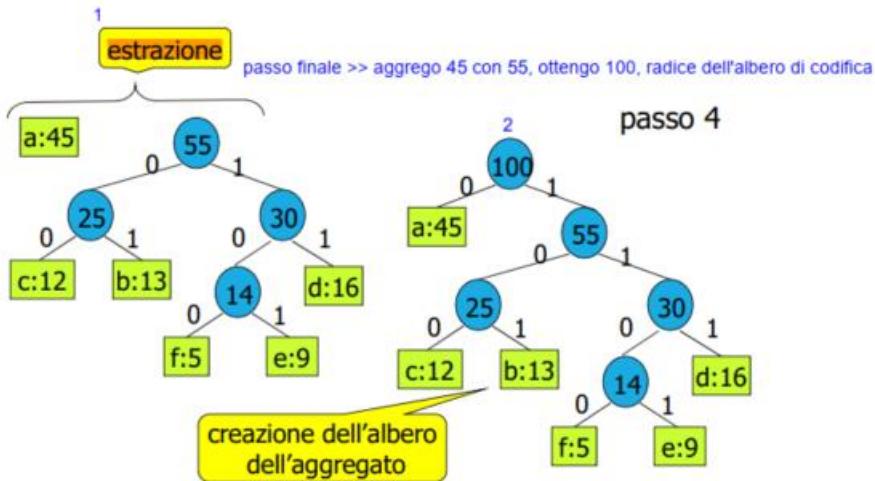
Ad ogni passo, creo un albero aggregato con la foglia che ho ed un altro nodo – come radice avrò un nodo che è l'unione dei due.



Una volta creato questo nodo, che sarà quindi il nodo con priorità 14, lo inserisco di nuovo nella coda a priorità.



Ripeto questo procedimento, fino ad avere un unico nodo, che sarà il nodo dell'albero che avrò creato per la codifica della stringa o del valore in generale.



Modularità

Programma si dice modulare quando risolve un problema, dividendolo in *sottoproblemi*

- Più precisamente, questo viene fatto, dato un obiettivo, dividendo le varie parti di un programma in funzioni, che risolvono ognuna una parte dell'obiettivo.
- Una struttura dati è resa modulare identificandone le parti, e associando a ciascuna le funzioni che vi operano.

Un **dato modulare** è un tipo di dato con le relative funzioni.

L'obiettivo della modularità è quello di **mantenere una coerenza a livello di implementazione per quanto riguarda il programma, e le funzioni che lo vanno a comporre**.

- La coerenza riguarda anche come le funzioni vengono implementate e quali argomenti è necessario passarci.

NON sono programmi modulari > programmi dove la risoluzione del problema è completamente affidata al main

Sono esempi di un programma modulare, invece, dei programmi dove:

- Il main chiama solo delle funzioni, a cui è invece affidata la manipolazione di una struttura dati ed eventuali operazioni su di essa > es. ho una struct punto, il main chiama solo funzioni per fare la sua init(), e altre operazioni su di essa

Un programma modulare può far uso sia dell'allocazione automatica, che quella dinamica > in questo caso le funzioni non restituiranno struct ma puntatori a struct.

- È importante comunque mantenere sempre una coerenza > se una funzione lavora su un valore e lo restituisce, allora tutte le altre dovranno restituirlo dopo averci lavorato su (*poco sensato avere che una funzione ha un valore di ritorno, mentre le altre sono tutte void*)

Posso anche lavorare direttamente sui valori >> uso funzioni che prendono valori by reference >> tutte funzioni di return type *void* >> programma risulta anche più efficiente, non devo allocare nuova memoria per copie di valori (*passing by value*)

COME EVITARE MEMORY LEAK

Finora, è a discrezione del programmatore e pressochè indifferente quale metodo si scelga per lavorare sulla struttura dati – è però fondamentale avere un metodo chiaro per quanto riguarda la *creazione* e *distruzione* (*free*) di istanze della struttura dati con cui lavoriamo – ovvero:

- *La creazione di un dato deve essere gestita con uniformità, e deve essere evidente*
- *Lo stesso vale per la distruzione di istanze create precedentemente >> chi crea istanze deve anche distruggerle*
 - *se quindi l'allocazione dinamica è gestita dal client, allora anche la distruzione di essa sarà gestita dal client.*
 - *Se creazione è visibile, anche distruzione deve esserlo.*
- **Attenzione!** Spesso succede che si abbia una memory leak a causa di funzioni che allocano nuovi nodi senza che l'utente ne sia al corrente – così, al momento della terminazione del programma, quando l'utente prova a de-allocare vecchi nodi (che magari sono già stati de-allocati da altre funzioni) il programma crasha – oppure non de-allocava tutti i nodi effettivamente allocati, e si ha un memory leak.

Composizione ed aggregazione

Strategie per raggruppare dati o riferimenti a dati in un unico dato composto, tenendo conto delle relazioni gerarchiche e possesso

Composizione ed aggregazione specificano quindi “come” il programmatore decide di inserire un dato all’interno di, ad esempio, una *struct* o una *struttura dati generica*.

COMPOSIZIONE

- **Stretta con possesso** (*per valore*)>> *struct A* contiene il valore di *B*
- **Con riferimento** >> *struct A* contiene riferimento a *B* > viene comunque considerato suo

Se A possiede B, ha la responsabilità di distruggerlo >> prima di liberare la *struct*, libero i suoi dati interni.

VANTAGGI >> ogni dato è indipendente e ha un compito specifico – il tipo di dato “più alto” nella gerarchia coordina il lavoro di quelli più bassi

AGGREGAZIONE

- (composizione) **Senza possesso >>** *struct A* fa riferimento a *B* – *B* è però un dato esterno indipendente, o addirittura una struttura dati.
 - *A può anche contenere una sola parte dei dati di B*
 - *A non è responsabile della creazione e distruzione di B >> B esiste già*

Es. di composizione con aggregazione >> elenco tesi di laurea che contiene nome docente e studente >> entrambi i nomi appartengono a basi di dati diversi, che hanno anche altre informazioni >> io ho solo però questi due dati

Strutture dati contenitore – wrapper

Struttura di più alto livello che racchiude una serie di dati modificabili

Sono esempi di strutture wrapper, contenitore, le seguenti:

- Vettori, liste, code, tabelle di simboli, alberi, grafi... (ADT che vedremo dopo)
- Si effettuano operazioni su di esse tramite le funzioni apposite, che prendono come input il contenitore in sé >
 - Creazione / inserimento / cancellazione / conteggio / accesso / ordinamento / distruzione

```
typedef struct {  
    int *v;  
    int n;  
} ivet_t;
```

struttura dati che contiene un vettore + il suo numero di elementi

```
void ordinaVettoreConWrapper(ivet_t *w);
```

funzione che modifica il vettore contenuto nella struttura dati > prende come parametro l'intera struttura dati

Programmazione multi-file

In un file unico, ad ogni singolo cambiamento nel codice occorre *ri-compilare* l'intero programma >

- Poco efficiente

Entra in gioco la programmazione multi-file >

- Più efficiente > i moduli su più file sono compilati e testati individualmente > non devo ricompilare tutto il programma se cambio solo un singolo modulo
- Permette di lavorare più facilmente tra diverse persone sullo stesso progetto
- Rende più leggibile il codice e meglio strutturato

Come?

File header .h > definiscono le interfacce con cui lavorerà il programma

File implementazione .c > definiscono le effettive implementazioni di strutture dati e funzioni con cui il programma andrà ad operare

Nei file header si inseriscono quindi solo i prototipi delle funzioni – il main farà un semplice include di questi file.

Al momento della compilazione, poi, il client andrà a ricercare i file .c in cui è contenuta l'implementazione delle funzioni definite dentro le *header*.

-il main non deve per forza includere tutte le *header* di un programma > se un modulo *creatura* contiene al suo interno tutti gli altri moduli *punto.h*, *inventario.h* ecc. il main dovrà semplicemente fare *include* del modulo *creatura*

-Attenzione! Includere due volte lo stesso file manderà in errore il programma.

Come evitare l'inclusione multipla ? >>> Compilazione condizionale

```
// header1.h
#ifndef _HEADER1
#define _HEADER1
// istruzioni di header1.h
...
#endif
```

```
// header1.h - prima inclusione
#ifndef _HEADER1
#define _HEADER1
// istruzioni di header1.h
...
#endif
// header1.h - seconda inclusione
#ifndef _HEADER1
#define _HEADER1
// istruzioni di header1.h
...
#endif
```

Questa parte è disabilitata:
è come se non ci fosse

ADT

Le informazioni utilizzabili da un sistema sono dette dati e sono memorizzate all'interno di strutture dati (*interne, esterne, statiche, dinamiche*)

Il **tipo** di dato definisce *l'organizzazione e la manipolazione* dei dati

Tipo di dato:

Standard > int, float, char...

Definiti dall'utente > *typedef, funzioni*

Un ADT è un tipo di dato astratto la cui implementazione è nascosta all'utente – questo viene fatto creando prima un'interfaccia (*file .h*) che definisce *il tipo di dato e le funzioni che andranno ad essere usate* – ma non le implementa

- l'implementazione verrà poi svolta da un file esterno al main, e comunque non raggiungibile dall'utente
- finora abbiamo usato **quasi ADT** > strutture la cui implementazione è completamente visibile all'utente – si fa quindi affidamento all'utente che questo non cambi le implementazioni ma si limiti ad usare le funzioni prestabilite.
- I **quasi ADT** permettono infatti all'utente, o al main, di vedere la struttura della struct su cui stiamo lavorando.

Come posso quindi *nascondere* la struct e l'implementazione dell'ADT in sé dal main?

- **handle** >> Interfaccia implementa un puntatore a struttura dati effettiva (*handle è in file.h, mentre struttura dati vera è in file.c, insieme all'implementazione di tutte le altre funzioni*)

Quasi-ADT > soluzione più semplice, non richiede allocazione dinamica

ADT > richiede allocazione dinamica, nasconde implementazione all'utente ma assicura che il programma venga usato “come è inteso che venga usato”

- il tipo di dato ADT fa uso di puntatori, e allude a quello che sarà nella programmazione ad oggetti il *polimorfismo* (*utilizzo di una interfaccia comune per diversi tipi di dati*)

Come definisco un ADT di I classe?

```
typedef struct item *Item;
```

Item è un nuovo tipo di dato > punta ad una struct di tipo **item**, che non è ancora però stata definita (*viene definita nel .c*)

- ogni implementazione andrà quindi a lavorare con un tipo di dato **Item**, ovvero con il puntatore alla struct effettiva. Questo fa sì che si nasconde l'implementazione della struttura dati in sé e il client non possa accedervi.
 - *Tutte le funzioni ricevono quindi invece della struct, un puntatore alla struct.*
 - *Quando dovrò allocare, allocherò per il contenuto del puntatore, quindi per sizeof(*Item)*

ADT per collezioni >>

Nel .h si crea soltando il puntatore alla struct wrapper >>

- Nel .c poi si implementa sia la struttura dati, che ciò che andrà a comporla (*quindi anche eventuali struct node ecc. non vengono definiti nel .h ma solo nel .c*)
- Ciò vuol dire che per una lista, ad esempio, nel .h definisco semplicemente un *typedef struct List *List_p,* e le eventuali funzioni >>
 - *Init(), Insert(), Delete(), Count(), Free(), Visit()*...

La funzione init() permette di creare un'istanza dell'ADT...*come per le classi.*

Esempi di ADT di I classe

```

list.h
typedef struct list *LIST;

void listInsHead (LIST l, Item val);
Item listSearch(LIST l, Key k);
void listDelKey(LIST l, Key k);

```

ADT di 1 classe – **SET (insieme)**

Collezione di dati non ordinati che supportano operazioni di unione, intersezione insiemistica e operazione di display se un elemento appartenga o meno ad un insieme

```

Set.h
typedef struct set *SET;

SET SETinit(int maxN);
void SETfree(SET s);
void SETfill(SET s, Item val);
int SETsearch(SET s, Key k);
SET SETunion(SET s1, SET s2);
SET SETintersection(SET s1, SET s2);
int SETsize(SET s);
int SETempty(SET s);
void SETdisplay(SET s);

```

Un set può essere implementato sia con un vettore, che con una lista. (ordinati e non)

Domanda esame orale – Perché non si adatta una soluzione stile union-find?

(vettore con corrispondenza dato-indice?)

In union-find, ogni elemento appartiene ad un solo insieme – inoltre, gli insiemi sono disgiunti, senza parti in sovrapposizione.

Il caso generale è diverso > un elemento può appartenere a più insiemi, che supportano diverse operazioni di unione, intersezione, differenza...

Vettore o Lista?

- **Vettore >**
 - *Ordinato* > ricerca dicotomica $O(n \log N)$, inserimento $O(N)$
 - *Non ordinato* > ricerca lineare $O(N)$, inserimento $O(1)$
- **Lista >**
 - *Ordinata* > ricerca lineare $O(N)$, inserimento $O(N)$ – posso interrompere ricerca prima
 - *Non ordinata* > ricerca lineare $O(N)$, inserimento $O(1)$ in testa

SETunion & SETintersection >>

- Lista / vettore ordinato >> $O(N)$
- Lista / vettore NON ordinato >> $O(N^2)$

SETUNION E SETINTERSECTION – LISTE & VETTORI ORDINATI

SETUNION – IMPLEMENTAZIONE

```
Set.c  Unione di due insiemi
SET SETunion(SET s1, SET s2) {
    int i=0, j=0, k=0, size1=SETsize(s1);
    int size2=SETsize(s2);
    SET s;
    s = SETinit(size1+size2);
    for(k = 0; (i < size1) || (j < size2); k++)
        if (i >= size1) s->v[k] = s2->v[j++];
        else if (j >= size2) s->v[k] = s1->v[i++];
        else if (ITEMless(s1->v[i], s2->v[j]))
            s->v[k] = s1->v[i++];
        else if (ITEMless(s2->v[j], s1->v[i]))
            s->v[k] = s2->v[j++];
        else { s->v[k] = s1->v[i++]; j++; }
    s->N = k;
    return s;
}
```

strategia simile alla Merge del MergeSort

k sovrdimensionato come dimensione i + dim j

casi in cui uno dei due vettori è finito\

casi di confronto tra i due valori > quello che aggiungo, aumento il suo indice

vettori sono ordinati -> si guardano i primi >

caso in cui i due valori sono uguali > aumento entrambi gli indici

Allocò prima un nuovo set di dimensione dim1 + dim2.

Poi scorro con un ciclo, finché non ho finito gli elementi di tutti gli insiemi – mi fermo quando gli indici sono arrivati alla dimensione del loro set.

Prendo per volta un elemento di ciascun insieme, e li confronto:

- Se uguali, aumenta gli indici di entrambi i set e salvane solo uno.
- Sennò salva il più grande, e aumenta l'indice per quel set.

SETINTERSECTION – IMPLEMENTAZIONE

Set.c**Intersezione di due insiemi**

```

SET SETintersection(SET s1, SET s2) {
    int i=0, j=0, k=0, size1=SETsize(s1);
    int size2=SETsize(s2), minsize;
    SET s;
    minsize = min(size1, size2);
    s = SETinit(minsize);
    while ((i < size1) && (j < size2)) {
        if (ITEMeq(s1->v[i], s2->v[j])) {
            s->v[k++] = s1->v[i++]; j++;
        }
        else if (ITEMless(s1->v[i], s2->v[j])) i++;
        else j++; se son diversi, aumenta indice del più piccolo
    }
    s->N = k;
    return s;
}

```

Unico caso in cui salvo elemento in risultato (e poi aumento indice k)

dimensione soluzione = dimensione insieme più piccolo

int min (int x, int y) {
 if (x <= y)
 return x;
 return y;
}

se uno dei due vettori finisce, ho sicuramente finito (no intersezioni)

Il nuovo insieme avrà AL MASSIMO la dimensione del set più piccolo.

Finché non finisco uno dei due vettori:

- Se i due oggetti correnti sono uguali, salvali nel vettore nuovo.
- Sennò aumenta l'indice del più piccolo elemento dei due.

SETUNION E SETINTERSECTION – LISTE & VETTORI NON ORDINATI
Set.c**UNIONE DI LISTE NON ORDINATE**

```

SET SETunion(SET s1, SET s2) {
    link x1, x2; int founds2, counts2=0;
    SET s = SETinit(s1->N + s2->N);
    x1 = s1->head;
    while (x1 != NULL) {
        SETfill(s, x1->val); x1 = x1->next; per ogni elemento ,mette elemento in risultato
        for (x2 = s2->head; x2 != NULL; x2 = x2->next) {
            x1 = s1->head;
            founds2 = 0; controlla quali elementi sono presenti in x1 e s2 -> prenderò quei valori
            while (x1 != NULL) {
                if (ITEMeq(x1->val, x2->val)) founds2 = 1;
                x1 = x1->next;
            }
            if (founds2 == 0) {
                SETfill(s, x2->val); counts2++; }
        }
        s->N = s1->N + counts2;
        return s;
    }

```

Deve prendere tutti elementi, tranne doppioni >

inserimento in testa a lista non ordinata

inserimento in testa a lista non ordinata

Prima inserisce tutti gli elementi del primo insieme in risultato.

Poi ciclo sul secondo set >> se trovo un elemento che non è contenuto nel primo set, allora lo inserisco.

Set.c

```

SET SETintersection(SET s1, SET s2) {
    link x1, x2; int counts=0; SET s;
    s = SETinit(s1->N + s2->N); sbagliato, è SETinit(min(s1->N, s2->N))
    x1 = s1->head;
    while (x1 != NULL) { itero su tutti gli oggetti e controllo quali sono
        x2 = s2->head; uguali
        while (x2 != NULL) {
            if (ITEMeq(x1->val, x2->val)) {
                SETfill(s, x1->val); counts++; break;}
            x2 = x2->next;
        }
        x1 = x1->next;
    }
    s->N = counts;
    return s;
} per le liste non ordinate e i vettori ordinati, l'intersezione si può fare in entrambi
i modi.

```

inserimento in testa
a lista non ordinata

Ciclo su ogni oggetto del primo set > ciclo poi sul secondo set, dove è uguale, lo inserisco ed interrompo.

!!!!

DA QUI IN POI SONO SPIEGATI DIVERSI TIPI DI ADT E STRUTTURE DATI FONDAMENTALI

!!!!

Code generalizzate (QUEUE)

Code e Stack, Coda prioritaria

Code generalizzate > collezioni di oggetti di tipo *Item* con operazioni di

- Insert / search / delete
- *Init()*, *count()*, *free()*, *copy()*

Stack / Pila >>

- **LIFO > Last in, First out >>** *delete()* elimina l'ultimo elemento aggiunto
 - *Operazioni di inserzione push() e estrazione pop()*

Queue / coda >>

- **FIFO > First in, first out >>** delete() elimina il primo elemento inserito
 - *Inserzione in coda ed estrazione in testa*

Per le code a priorità, l'estrazione in testa garantisce una estrazione del valore con priorità massima.

- L'operazione di estrazione può quindi sia ritornare un valore casuale, che un valore ben definito >> dipende dalla struttura dati con cui lavoriamo, e se i dati seguono un ordine di priorità o meno. (*tavella di simboli*)

ADT STACK - PILA

LIFO, last in first out – elementi si sovrappongono uno sopra l'altro – prima di togliere quello in fondo devo togliere tutti quelli prima.

- Push() e Pop() operazioni fondamentali.
- Si implementano sia con liste che con vettori >> **le operazioni sono comunque sempre O(1)**
 - *Vettori* > se ho dimensioni prestabilite
 - *Liste* > dimensioni variabili, programmi dinamici

Implementazione avviene se in forma di ADT con puntatore alla struct originale >>>

- La struct wrapper avrà un contatore con il numero di elementi (Facilita operazione di STACKempty()) e un vettore / puntatore alla testa della lista – che implementa la struttura dati in sé.

```
stack.h
void STACKinit(int maxN);
int STACKempty();
void STACKpush(Item val);
Item STACKpop();
```

L'implementazione è quasi uguale per lista e vettore, poiché gli elementi vengono sempre inseriti in testa, e quindi la complessità è sempre O(1) con accesso diretto agli elementi interessati.

ADT QUEUE - CODA

FIFO – first in, first out. Gli elementi inseriti per primo vengono rimossi per primi.

- Enqueue / put >> mette elemento in coda

- Dequeue / get >> toglie elemento più vecchio dalla coda

Stesso discorso che per stack per quanto riguarda l'utilizzo di vettore oppure lista – dipende se i dati sono dinamici o statici.

- Head + tail >> li uso sia per vettori che per liste > permettono di accedere al primo elemento inserito (*head*) e all'ultimo inserito (*tail*) per ordine cronologico

Implementazione – vettore

Head – tail sono indici >

- Put() inserisce elemento all'ultima posizione disponibile, ovvero tail > **O(1)**
- Get() prende l'elemento in posizione 0, ma fa scalare poi tutti gli altri > **O(N)**.

E' possibile anche un'altra implementazione, in cui head non ha solo valore 0 ma un valore compreso tra 0 e N-1

- In questo caso, la head è un valore che cambia > ad ogni get() si sposta in avanti (avrò bisogno di, nel caso head raggiunga la dimensione massima, farlo ritornare a 0)
- **Get() e put()** sono **O(1)**
- **Coda è piena quando tail = head**

Questo può essere facilitato *usando un contatore per gli elementi – una volta raggiunto quel massimo, non si può più inserire (put) ma solo prendere (get) – così non devo controllare che head == tail (può anche essere vero se la coda è vuota)*

- Funzione QUEUEempty() basta che controlli numero elementi == 0

Implementazione – liste

Head – tail sono puntatori ai diversi nodi

- *Put() usa tail ed inserisce direttamente dopo l'ultimo elemento > O(1)*
- *Get() usa head ed estrae dalla prima posizione > O(1)*

```
void QUEUEput (QUEUE q, Item val) {
    if (q->head == NULL){
        q->tail = NEW(val, q->head) ;
        q->head = q->tail;
        return;
    }
    q->tail->next = NEW(val,q->tail->next);
    q->tail = q->tail->next;
}

Item QUEUEget(QUEUE q) {
    Item tmp = q->head->tmp;
    link t = q->head->next;
    free(q->head); q->head = t;
    return tmp;
}
```

ADT CODA PRIORITARIA – vettori e liste

Implementazione con *heap* trattata successivamente

- Vettore / lista non ordinato >

- Inserzione > $O(1)$
- Estrazione / visit del max/min > $O(N)$
- Cambio di priorità > $O(N)$ (*devo cercare elemento*)
- Vettore / lista ordinato >
 - Inserzione > $O(N)$
 - Estrazione / visit del max/min > se ho head / tail $O(1)$
 - Cambio di priorità:
 - *Liste > $O(N)$ – devo cercarlo, eliminarlo e re-inserirlo ($O(N) + O(1) + O(N)$)*
 - *Vettore > $O(N)$ – ricerca dicotomica, eliminazione e re-inserimento ($O(\log N) + O(N) + O(N)$) – eliminazione comporta spostamento di tutti gli altri valori*

Liste (*linked lists*)

Sequenze lineari > sequenza di elementi di tipo struct / “Item”, in cui ad ogni elemento è associato un indice univoco.

- Sulle coppie di elementi è definita una relazione predecessore / successore (*ogni elemento è, rispetto ad un altro, predecessore o successore*)

La sequenza è *ordinata con criterio posizionale* (*primo, secondo, terzo dato...*)

- Può essere ordinata in base ad una *chiave* o no
- L'accesso viene eseguito tramite chiave > *ricerca, $O(n)$*
- L'accesso viene eseguito tramite *posizione*
 - Accesso diretto > $O(1)$, vettore
 - Accesso sequenziale > $O(N)$

Esempio di sequenza lineare: vettore

- Memorizzato come insieme di dati contigui in memoria
- Accesso diretto, **$O(1)$**

Altro esempio di sequenza lineare: lista concatenata

- Dati non contigui in memoria
- Accesso sequenziale > per arrivare ad un oggetto devo prima passare per tutti quelli che lo precedono > **$O(N)$**

Come realizzare una lista?

- Con vettori: *se so già il numero massimo di elementi, oppure sfrutto la ri-allocazione (dopo un tot di elementi ri-alloco per evitare di assegnare elementi in posizioni non esistenti)*
 - Contiguità degli elementi, posso sfruttare indici
- Con concatenazione: *utilizzando strutture ricorsive allocate individualmente >> se non è noto o stimabile il numero di elementi;*
 - Se ho un'unica direzione di percorrimento, si dice **lista concatenata semplice >>** predecessore -> successore
 - Se ho due direzioni di percorrimento, **lista concatenata doppia >>** predecessore <-> successore (*ogni item avrà un campo next e previous*)

Operazioni sulle sequenze lineari (liste)

- *Ricerca* di un elemento per un parametro (che sia chiave di ricerca o altro valore della struct *Item*)
- *Inserimento* >
 - Lista non ordinata >> in testa / in coda >> O(1)
 - Lista ordinata >> in posizione corretta per mantenere ordinamento >> O(N)
- *Cancellazione* >
 - Lista non ordinata >> in testa [O(1)] oppure per una determinata posizione R [O(R)]
 - Con campo di ricerca uguale a quello da eliminare > prima ricerca poi elimina, quindi O(N)
 - *La cancellazione può sia cancellare l'elemento che cancellarlo e restituirlo come valore di return (funzione di Pop)*

Liste concatenate

Sono delle strutture dati dinamiche realizzate tramite una sequenza di nodi.

Un nodo non è altro che una struct contenente:

- Un valore (*Item*)
- Un riferimento
 - *Al prossimo elemento* > liste concatenate semplici
 - *Al prossimo elemento && al precedente* > liste concatenate doppie
- ****Una chiave** (*la chiave può essere sia un intero che una stringa*)
 - La chiave ha il compito di facilitare il riconoscimento di un particolare nodo; sono quindi presenti funzioni di utilità come KEYget, KEYeq, KEYless, KEYgreater che restituiscono nodi o valori comparando gli item di un nodo con quella particolare chiave.

Come posso definire un nodo? Diversi modi >>

I primi due modi sono i più facili ed intuitivi; il primo crea un nuovo tipo che è il tipo di dato link, che è un puntatore al dato node;

il secondo crea una semplice struct.

```
typedef struct node *link;
struct node {
    Item val;
    link next;
};
```

```
struct node {
    Item val;
    struct node *next;
};
```

```
typedef struct node {
    Item val;
    struct node *next;
} node_t, *link;
```

```
typedef struct node node_t;
struct node {
    Item val;
    node_t *next;
};
```

Creare una lista

Per creare una lista, non devo fare altro che inizializzare un nodo a null.

Quando vorrò inserisci un nuovo nodo, ci inserirò un nuovo nodo dopo averlo creato semplicemente facendo `head = nuovoNodo`.

- Come alloco un nuovo nodo? Creo una funzione per farlo così ogni volta posso semplicemente richiamarla...**newNode(Item x, link head)**
 - Fa la malloc del contenuto di link, ovvero della struct nodo. Setta poi l'Item a x e lo inserisce in `head` usando le funzioni definite tra poco

Liste: Operazioni atomiche (fondamentali):

- Inserimento >> se lista non ordinata, inserisco in testa >> setto al nuovo nodo come prossimo l'elemento corrente in `head`, e poi assegno ad `head` quello nuovo.
- Cancellazione >> attraverso la lista fino al nuovo desiderato >> due modi, o controllo il prossimo del nodo corrente, oppure controllo il nodo corrente tenendo un doppio puntatore (uno a quello corrente, uno al suo precedente) >> una volta trovato il nodo da eliminare, setto a quello precedente come prossimo il nodo prossimo a quello da eliminare.
- Attraversamento >> diversi metodi
 - *Semplice puntatore al nodo attuale >>> continuo finché non è NULL*
 - *Doppio puntatore >> uno al precedente, uno a quello corrente. Quando quello corrente è null, mi fermo.*
 - *Ricorsivo >> controllo nodo attuale, se è null mi fermo, sennò ricorro su nodo->next*
 - Questo metodo può essere sfruttato per fare operazioni all'inverso su una lista > se prima della chiamata ricorsiva metto la funzione da eseguire, verrà eseguita in ordine. Se la metto dopo, verranno eseguite in ordine inverso (*questo perché la funzione verrà eseguita solo quando la chiamata ricorsiva sarà terminata, e quindi quando avranno finito anche le successive > il che vuol dire che prima di eseguire la funzione sul nodo attuale, dovranno avere finito tutte quelle che sono state fatte dopo, ovvero sugli item successivi*)

RICORDA: per quanto riguarda le liste, ogni operazione fatta iterativamente si può probabilmente fare anche ricorsivamente. Vale lo stesso per le seguenti funzionalità.

1) Liste non ordinate

INSERIMENTO > L'inserimento può essere fatto sia su una lista ordinata, che non. Cambiano le metodologie

- controllo che *head* non sia null. Se lo è, inserisci direttamente in *head* (*head = newNode(..)*), sennò posso:
 - Inserirlo in testa, **O(1)** e il nodo corrente in testa diventa il next di quello nuovo
 - Inserimento in coda, attraversare lista fino ultimo nodo e metterlo alla fine **O(N)**
 - Inserirlo in coda **con puntatore a ultimo elemento > O(1)**, serve però un puntatore aggiuntivo alla coda della lista.

CANCELLAZIONE / ESTRAZIONE >

- *Dalla testa > se lista non è vuota, aggiorna head che punterà al secondo elemento (current.next), e libera il primo.*
 - *Estrazione >> aggiorna head con current.next e restituisci il nodo da cancellare*
- **Con chiave data** > attraverso la lista con due puntatori, uno al corrente uno al precedente. Una volta che *corrente.chiave == chiave*, aggiorno il *next* del precedente e elimino quello cercato. **O(N)**

RICERCA >

- Le liste non permettono una ricerca che non sia lineare **O(N)** – questo perché non c'è accesso diretto agli elementi, ma sequenziale.

2) Liste ordinate

Dati ordinati in base alla loro chiave (comunque ordinati in base ai dati che contengono)

- **INSERIMENTO > O(N)**
 - Devo prima cercare la sua posizione esatta
- **RICERCA && CANCELLAZIONE > O(N)**
 - *Posso però interrompere prima la ricerca, se il valore attuale è minore di quello cercato.*

3) Liste concatenate particolari

Liste concatenate doppie, liste con sentinelle, nodi fintizi

Liste con nodi fintizi – “*nodi sentinelle*”

- Nodo con dato fintizio (in testa / coda) usato per rimuovere casi speciali, come lista vuota oppure inserimento / cancellazione del primo / ultimo nodo
 - Ad esempio quando creo lista, metto entrambi *head* e *Z* == NULL; quando faccio poi operazioni come l'attraversamento, continuo finché *x != Z*.

Liste circolari

- L'ultimo nodo punta al primo
- Il punto di stop di qualsiasi operazione sulla lista è quando il nodo che scorre sulla lista è di nuovo uguale a quello iniziale (*head*)

Lista concatenata doppia

- Ho un puntatore al precedente *previous* e uno al successivo *next*

```
typedef struct node *link, node_t;

struct node {
    Item val;
    link next;
    link prev;
};
```

- INSERIMENTO > arrivo in posizione precedente a quella dove inserire > prima aggiorno il precedente del prossimo (*next->prev = newNode*), poi *newNode->prev = current*, *newNode->next = current->next*, poi *current->next = newNode*
- CANCELLAZIONE > devo solo aggiornare i due puntatori > aggiorno il *next* di quello attuale, e il *previous* del prossimo >

Applicazioni comuni di liste

- Inversione di lista
- Insertion Sort

Inversione di lista – 2 metodi

- 1) Creo una nuova lista, ciclo su quella vecchia ed inserisco in testa in quella nuova ogni oggetto.
- 2) Ciclo ricorsivamente sulla lista attuale, e dopo la chiamata ricorsiva aggiungo l'item attuale alla lista. (*metodo già accennato precedentemente, gli item vengono aggiunti in ordine inverso poiché prima devo arrivare alla fine della lista ricorsivamente*)
- 3) **Versione integrata >> faccio tutto su una lista**

- L'obiettivo è scambiare il verso di ogni nodo -> come?
- Creo nodi *previous*, *current* e *next*
- Prima di tutto *salvo il nodo che c'è dopo* > non avrà modo di raggiungerlo dopo che ho cambiato il verso di quello corrente
- Scambio il verso del nodo attuale > *il suo next diventa previous*
- Il precedente diventa ora quello attuale
- Quello attuale diventa quello salvato all'inizio, ovvero il vecchio successivo (*avanzo e ripeto*)
 - Alla fine, *metto head = current* e ho invertito la lista

```
void invertLinkedList(link head)
{
    link temp,h;
    link prev,current,next;
    prev = head;
    h = head;
    while(h != NULL)
    {
        temp = h->next;
        h->next = prev;
        prev = h;
        h = temp;
    }
    head = h;
}
```

Insertion Sort

Poiché non è possibile l'accesso diretto agli elementi, è necessario un ordinamento quadratico **O(N^2)**

- Estraggo in testa da vecchia lista, inserisco in modo ordinato in quella nuova (N volte dalla lista vecchia per N ordinamenti di quella nuova)
- Anche qui sia versione con funzioni già implementate (basta fare una `listExtractHead` e poi una `listInsertSort`) e **una versione integrata**
 - **Ciclo su tutti gli oggetti della vecchia lista** > per ogni oggetto di lista vecchia, lo inserisco in ordine in quella nuova (*si ha il caso particolare in cui il primo elemento della lista nuova sia più grande di quello da inserire > in quel caso si inserisce direttamente in testa poiché è elemento minimo quello preso in carico, sennò si cerca la posizione esatta*)

Liste concatenate con indici

Posso anche realizzare liste concatenate con l'utilizzo di un vettore > i dati non sono ordinati effettivamente per un campo ma per l'ordine di inserimento

- È un vettore di struct
- Le struct non avranno un *link next* ma un semplice *int next*, dove *int next* è la posizione del prossimo nodo.
 - Un *next = -1* vuol dire NULL

RIASSUMENDO

Con le liste, posso agire in diversi modi. L'importante è però *provare a sfruttare sempre* l'accesso diretto tipico dei vettori >

In un caso in cui devo assegnare degli elementi di una lista ad alcune variabili, sarà molto più conveniente invece che creare una lista per ciascuna delle variabili, usare direttamente un vettore sia per *la lista* che per *ciascuna delle variabili* >

- *Inserisco direttamente gli indici del vettore delle canzoni dentro ogni variabile, così ho un costo unitario O(1) e in più ho una corrispondenza biunivoca tra vettore di ogni variabile e vettore di elementi*

Complessità riassunte

- **Liste non ordinate** > Inserimento $O(1)$, Ricerca e cancellazione $O(N)$, Selezione non ha senso (non c'è ordine)
- **Liste ordinate** > Inserimento $O(N)$, Ricerca e cancellazione $O(N)$, Selezione $O(N)$

Tabelle di simboli

Insieme di diversi possibili ADT che permettono funzioni di

- Insert
- Search

- Delete
- Count, print, init..
- *Ordinamento per campo*
- Selezione

Proprietà fondamentale: *possibilità di definire una relazione d'ordine* >> rango r

- La selezione si basa infatti sulla scelta di un oggetto dato il suo rango (*voglio il k-esimo oggetto*)

Quali ADT implementano le tabelle di simboli?

- Tabella accesso diretto
- Strutture lineari > *array, linked lists ordinate / non*
- Alberi
- Tabelle di Hashing

Esempio di applicazione di una tabella di simboli >> quando scriviamo un filepath ad un file, stiamo semplicemente facendo una ricerca su diverse tabelle di simboli, in cerca di un file con quel valore preciso. Tabelle di simboli quindi usate per passare da una chiave ad un valore.

TABELLE DI SIMBOLI – COMPLESSITÀ'

Caso peggiore

	caso peggiore		
	inserim.	ricerca	selezione
Tabelle ad accesso diretto	1	1	maxN
Array non ordinato	inserisco in fondo 1	devi passarli tutti n	
Array ordinato e ricerca lineare	n	n	1
Array ordinato e ricerca binaria	n	logn	1
Lista non ordinata	inserisco in testa 1	n	
Lista ordinata	n	n	n
BST <small>binary search tree (albero binario di ricerca) > albero in cui ogni nodo ha un right e un left - caso limite è un solo ramo e l'altro vuoto > oppure ogni nodo ha un solo figlio</small>	n	n	n
RB-tree <small>red-black tree > albero binario bilanciato > un albero binario ha un'altezza (profondità max) logaritmica nel numero di nodi</small>	logn	logn	logn
Hashing <small>generalizzazione delle tabelle ad accesso diretto > inserimento O(1)</small>	1	poco probabile (tutti stessa casella) n	

Note:

- Generalmente tutte le liste non ordinate hanno $O(1)$ di inserimento e $O(N)$ di ricerca, mentre la selezione non ha senso perché non ho un ordinamento per rango.
- In lista ordinata, non ho accesso diretto > sempre $O(n)$ per accesso, ricerca e selezione

- I BST nel caso peggiore sono una semplice lista, e quindi l'inserimento (che è ordinato) è $O(N)$, e la ricerca nel peggiore dei casi deve andare fino alla fine della lista $O(N)$.
- RB-tree è un albero binario bilanciato
- Hash Table hanno $O(1)$ come complessità, ma è *approssimativa* > c'è un ciclo for() dentro la funzione di hashing, ma che è comunque molto basso.

Caso medio

	caso medio		
	inserim.	ricerca	selezione
Tabelle ad accesso diretto	1	1	$\max N/2$
Array non ordinato	1	$n/2$	
Array ordinato e ricerca lineare	$n/2$	$n/2$	$n/2$
Array ordinato e ricerca binaria	$n/2$	$\log n$	$\log n$
Lista non ordinata	1	$n/2$	
Lista ordinata	$n/2$	$n/2$	$n/2$
BST <small>come quicksort > caso peggiore molto improbabile, caso medio logn</small>	$\log n$	$\log n$	$\log n$
RB-tree	$\log n$	$\log n$	$\log n$
Hashing	1	1	

Nella maggior parte dei casi, la complessità del caso medio è la metà (/2) > dovrò attraversare mediamente metà degli elementi presenti.

Tabelle di simboli – diverse possibili implementazioni

1) Tabelle ad accesso diretto

Corrispondenza biunivoca chiave <> indice: la dimensione della tabella è pari al numero più grande dell'insieme preso in considerazione

- Indice accessibile tramite funzione getIndex(Chiave k)
 - Elementi accessibili quindi tramite st[getIndex(k)]
- Per numeri molto grandi, eccessivo spreco di memoria

È un semplice vettore – l'inserimento è quindi $O(1)$ e anche la ricerca $O(1)$ (*basta usare come indice il numero*) – selezione è $O(n)$ > nel peggiore dei casi devo prendere l'ultimo elemento

La complessità maggiore si ha nell'*inizializzazione* nel momento in cui $\max N$ è molto grande.

Esempi di getIndex > chiavi sono lettere maiuscole dell'alfabeto

```
int GETindex(Key k) {  
    int i;  
    i = k - 'A';  
    return i;  
}
```

Oppure se sono interi, ritorno semplicemente il valore ricercato.

getIndex(Key k){return (int) k;}

2) Vettore non ordinato

Semplice vettore

- Inserimento in fondo, (elements – 1), O(1)
- Ricerca O(N) > devo scorrere tutto vettore
- Non c'è selezione > elementi non sono ordinati, non ha senso
 - Possibile spreco di memoria per vettori sovraallocati

3) Vettore ordinato

Semplice vettore ordinato

- Inserimento **O(N)** > devo trovare posizione esatta (e tenere conto di eventuale ri-allocazione)
 - *Ordinamento* può essere fatto una volta sola se i dati sono letti da un file una volta sola > in questo caso inserimento ha O(1) di complessità
 - *Ordinamento* può anche essere fatto ogni volta dopo inserimento > in questo caso inserimento ha O(N) di complessità
- Ricerca / cancellazione O(logN) > *ricerca dicotomica*
- Selezione O(1) >> **rango e indice coincidono**

4) Liste

La struttura sarà una struct wrapper con dentro lista + numero elementi

Lista concatenata non ordinata >

- Inserimento in testa O(1)
- Ricerca O(N)
- Selezione non ha senso

Lista ordinata >

- Inserimento ordinato O(N)
- Ricerca O(N) (*non ho accesso diretto*)
- Selezione O(N) >> non guarda la chiave ma conta solo a quanti elementi è arrivato – quando arriva al numero scelto, ritorna l'Item corrente

Continua con altre implementazioni, guarda BST – IBST ecc.

Cosa fare se ho elementi duplicati nelle tabelle di simboli?

- 1) Ignoro l'elemento aggiunto duplicato / elimino quello vecchio e inserisco quello nuovo
- 2) Ammetto elementi duplicati > elementi duplicati saranno un vettore contenenti tutte le occorrenze, oppure utente sceglie in base a cosa visualizzare gli elementi (magari su campi che non sono identici di due elementi che magari hanno lo stesso nome)

Heap (ADT)

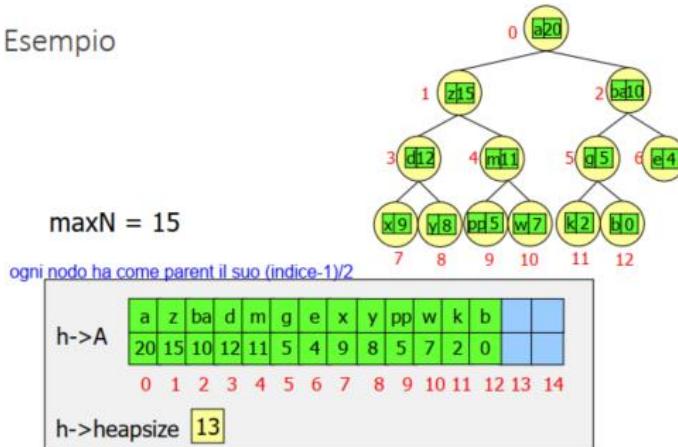
È un albero binario

- **Struttura** > quasi completo, tranne ultimo livello – è rappresentato tramite un vettore dinamico, ma concettualmente è un albero binario
 - *I'utilizzo di un vettore fa sì che non vi sia alcuno spreco di memoria – nessuno spazio inutilizzato*
- **Proprietà funzionale** > qualsiasi nodo appartenente alla struttura, non può avere valore maggiore del suo elemento padre.
 - *Radice = chiave con valore max*
 - *DIVERSO DA BST > non ci sono relazioni tra rami disgiunti*

Appunti sull'implementazione

- **H** heap
 - **H->A** vettore di **maxN-1** elementi
 - **H->heapsize** numero di elementi
 - **Root > A[0]**
- *Essendo un albero ma rappresentato come un array, gli elementi dell'array avranno una posizione precisa. Dato un elemento in posizione i*
 - *Figlio **sx** > A[**2i+1**] (funzione **Left(i)**)*
 - *Figlio **dx** > A[**2i+2**] (funzione **Right(i)**)*
 - *Padre > A[**(i-1)/2**] (funzione **Parent(i)**)*

Esempio



ADT di classe Heap

Heap.h

```
typedef struct heap *Heap;  
  
Heap  HEAPinit(int maxN);  
Void  HEAPfree(Heap h);  
void  HEAPfill(Heap h, Item val);  
void  HEAPsort(Heap h);  
void  HEAPdisplay(Heap h);
```

Alcune implementazioni

- HEAPinit() >> semplice malloc della struttura dati + malloc array usando maxN
- HEAPfree() >> libero array e poi sd
- HEAPfill() >> inserimento diretto in array in posizione $h\rightarrow\text{heapsize}++$ ($O(1)$)
 - Questo inserimento non rispetta la proprietà fondamentale di uno heap >> funzione **HEAPify()**
- HEAPdisplay() >> ciclo su array e stampa
- **HEAPify()** >> trasforma in HEAP un albero binario che non rispetta le proprietà di uno HEAP (ad esempio dopo semplice inserimento con HEAPfill)
 - Data una posizione i e un LEFT(i) e RIGHT(i) già esistenti, assegna alla posizione i il massimo tra i tre valori (uno da inserire e due già inseriti)

- Se un valore è scambiato così facendo tra A[i] e LEFT/RIGHT(i), applica ricorsivamente a quel sottoalbero HEAPify()

```
void HEAPify(Heap h, int i) {
    int l, r, largest;
    l = LEFT(i);
    r = RIGHT(i);
    if ((l < h->heapsize) && inizia a guardare il nodo e il suo figlio sinistro
        KEYcmp(KEYget(h->A[l]), KEYget(h->A[i]))==1)
        largest = l; in largest salva l'indice del valore massimo
    else
        largest = i;
    if ((r < h->heapsize) && confronta poi il largest con il figlio di dx
        KEYcmp(KEYget(h->A[r]), KEYget(h->A[largest]))==1)
        largest = r;
    if (largest != i) { se i = largest allora hai finito - sennò scambiali e ricorri
        swap(h, i, largest);
        HEAPify(h, largest);
    }
}
```

- **HEAPbuild()** >> trasforma un albero binario in HEAP
 - Parte dal *parent()* delle ultime foglie, e risale fino ad arrivare ad usare *HEAPify()* sul root (*obiettivo*)

```
void HEAPbuild (Heap h) {
    int i;
    for (i=PARENT(h->heapsize-1); i >= 0; i--)
        HEAPify(h, i);
}
```

- **O(N)** > sarebbe O(n log N) ma equazione ricorrenze si riduce ad O(n) (HEAPify ha costo logaritmico, e viene eseguito N/2 volte)

Sostituendo in T(n):

$$\begin{aligned}
 T(n) &= \sum_{i=0}^{\log_2 n} 2^i \log_2(n/2^i) \\
 &= \log_2 n \sum_{i=0}^{\log_2 n} 2^i - \sum_{i=0}^{\log_2 n} i 2^i \\
 &= \log_2 n (2n-1) - 2(1-(\log_2 n+1)n + 2n \log_2 n) \\
 &= 2n - \log_2 n - 2 \\
 &= \textcolor{red}{O(n)}
 \end{aligned}$$

$$\sum_{i=0}^k i x^i = x (1-(k+1)x^k+kx^{k+1})/(1-x)^2$$

- **HEAPSORT()** >> **O(n logN)** >> come?
 - Trasforma vettore in HEAP con HEAPbuild (O(n))
 - Scambia primo e ultimo elemento > riduce dimensione di 1 e poi usa HEAPify() su vettore di dimensione N-1 >> *HEAPify() stavolta parte dalla radice*
 - *In loco, non stabile*

- Vengono quindi messi in fondo gli elementi più grandi uno ad uno, e man mano che diminuisco la dimensione dell'HEAP opero solo più sugli elementi non ancora trattati.

```
void HEAPsort(Heap h) {
    int i, j;
    HEAPbuild(h);
    j = h->heapsize;
    for (i = h->heapsize-1; i > 0; i--) {
        Swap (h,0,i);
        h->heapsize--;
        HEAPify(h,0);
    }
    h->heapsize = j;
}
```

quicksort() sempre preferito...ma HEAPsort() usato in coda a priorità

Code a priorità – priority queue – PQ - HEAP

Obiettivo > mantenere un set di elementi secondo un ordine di priorità

- Inserzione
- Estrazione massimo
- Lettura massimo
- Cambio priorità

Una coda a priorità può essere implementata tramite un vettore / lista ordinato e no, oppure un HEAP di dati

ADT di I classe Coda a Priorità

PQ.h
<pre>typedef struct pqueue *PQ; PQ PQinit(int maxN); void PQfree(PQ pq); int PQempty(PQ pq); void PQinsert(PQ pq, Item val); Item PQextractMax(PQ pq); Item PQshowMax(PQ pq); void PQdisplay(PQ pq); int PQsize(PQ pq); void PQchange(PQ pq, Item val);</pre>

discussion a parte

Complessità >>

Ricorda sempre che l'insert costa $O(1)$, la ricerca di elementi costerà N / non $O(1)$

	PQinsert	PQshowMax	PQextractMax
Vettore non ordinato	1	N	N
Lista non ordinata	1	N	N
Vettore ordinato	N	1	1
Lista ordinata	N	1	1
Heap di item/indici	logN	1	logN

max in coda

max in testa

extractMax è logaritmico perchè quando togli l'elemento in testa dovrà mettere a posto tutto

Struttura dati in sé >

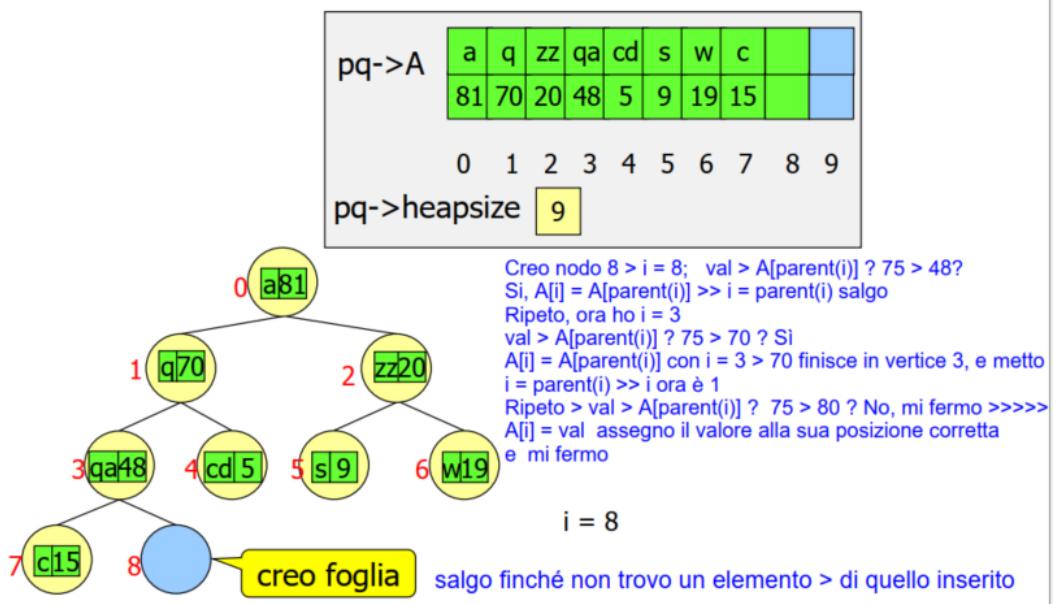
- Implementazione classica HEAP + funzioni per PQ priority queue

```
struct pqueue { Item *A; int heapsize; };
```

Alcune implementazioni delle funzioni base

- PQinit() >> alloca struttura dati + vettore per HEAP di dimensione maxN
- PQfree() >> libera vettore + sd
- PQempty >> return 1 se pq->heapsize == 0
- PQshowMax() >> ritorna il valore in posizione 0 A[0] (*valore massimo, proprietà HEAP*)
- PQdisplay() >> stampa tutti elementi
- **PQinsert()** >> devo aggiungere foglia all'albero > se il padre ha un valore minore, allora li scambio > faccio lo stesso fino alla radice se necessario > $O(\log N)$

```
void PQinsert (PQ pq, Item val) {
    int i;
    i = pq->heapsize++;
    cerca la prima casella che va bene >
    casella in cui valore è più grande di quello da inserire
    while((i>=1) &&
          (KEYcmp(KEYget(pq->A[PARENT(i)]),KEYget(val))==-1)){
        pq->A[i] = pq->A[PARENT(i)];
        i = PARENT(i); fa salire la i e fa scendere il valore che sta sopra
    } (il parent scende e crea posizione libera man mano
    pq->A[i] = val; che salgo
    return;
}
```



- **PQextractMAX()** >> estrae elemento in radice, scambiandolo prima con ultima foglia e poi chiamando HEAPify() dalla radice per ripristinare proprietà HEAP > **O(logN)**
 - Praticamente prima togli l'elemento e poi ordini l'heap (stesso procedimento di HEAPSORT, solo che non riduco size dell'HEAP ecc. e HEAPify parte dalla radice stavolta)
- **PQchange()** >> modificare la priorità di un elemento >> come?
- **2 modi > in base a come cambia il valore**
 - Se aumenta, potrà salire e quindi confronto con elemento superiore (come per PQinsert) e scambio eventualmente
 - Se diminuisce, sarà sicuramente più piccolo, quindi uso HEAPify()
 - **O(n) + O(logn) = O(N)**
 - Quindi prima ricerco il valore in sé e mi salvo il suo indice – applico poi lo stesso procedimento che per la pqInsert > confronto con parent finché elemento non è al posto corretto > una volta al posto corretto, uso l'indice salvato per salvare il vecchio valore al suo nuovo posto e chiamo HEAPify()

```

void PQchange (PQ pq, Item val) {
  int i, found = 0, pos;
  for (i = 0; i < pq->heapsize && found == 0; i++)
    if (NAMEcmp(NAMEget(&(pq->A[i])), NAMEget(&val))==0) {
      found = 1;    ricerca del nodo O(N)
      pos = i;
    }

  if (found==1) {
    while(pos>=1 &&
          PRIoget(pq->A[PARENT(pos)])<PRIoget(val)){
      pq->A[pos] = pq->A[PARENT(pos)];
      pos = PARENT(pos);
    }
    pq->A[pos] = val;  processo analogo al PQInsert O(lg)
    HEAPify(pq, pos);
  }
  else
    printf("key not found!\n");
  return;
}
  
```

Si può ridurre la complessità di PQchange:

- 1) Inserire nella coda solo *riferimenti* ad Item (vettore di posizioni) / corrispondenza dato-indice nel vettore, è così possibile ottenere la posizione di un Item in coda con tempo O(1)
- 2) Si usa una chiave univoca, gestita ad esempio con una tabella di simboli efficiente (*hash table O(1)* / *BST bilanciato O(logn)*)

1) Coda prioritaria di indici

Non si inseriscono in coda gli item ma coppie (indice,priorità), quindi si adotta la versione di *chiave affiancata al dato* (la priorità è un parametro aggiuntivo) invece che *chiave parte del dato*.

// implementazione NON obbligatoria per esame

```
PQ.h
typedef struct pqueue *PQ;
PQ PQinit(int maxN);
void PQfree(PQ pq);
int PQempty(PQ pq);
int PQsize(PQ pq);
void PQinsert(PQ pq, int index, int prio);
int PQshowMax(PQ pq);
int PQextractMax(PQ pq);
void PQdisplay(PQ pq);
void PQchange(PQ pq, int index, int prio);
```

Scompare il tipo Item
Si gestiscono indici e priorità

```
typedef struct {int index; int prio} heapItem;
struct pqueue {heapItem *A; int heapsize; int *qp};
```

Alcuni dettagli sull'implementazione con vettore di indici...

- PQinit() >> inizializza vettore di oggetti e vettore indici
- PQfree() >> libera vettori + sd
- PQempty() >> ritorna 1 se heapsize == 0
- PQinsert() >> *uguale a precedenti implementazioni* > invece di controllare il valore dell'Item confronto le due *prio* > se quella del figlio è maggiore del padre, li scambio e continuo
- Swap() > funzione per scambiare due valori nella coda a priorità *heapItem*

```
static void Swap(PQ pq, int pos1, int pos2){
    heapItem temp;
    int index1, index2;
    temp = pq->A[pos1];
    pq->A[pos1] = pq->A[pos2];
    pq->A[pos2] = temp;
    // update correspondence index-pos
    index1 = pq->A[pos1].index;
    index2 = pq->A[pos2].index;
    pq->qp[index1] = pos1;
    pq->qp[index2] = pos2;
}
```

- **HEAPIfy()** > uguale a precedenti implementazioni > confronta prio invece di altri elementi
- **PQextractMAX()** > uguale a precedenti implementazioni
- **PQchange()** >

```
void PQchange (PQ pq, int index, int prio) {
    int pos = pq->qp[index];
    heapItem temp = pq->A[pos];
    temp.prio = prio; // new prio

    while ((pos>=1) && (pq->A[PARENT(pos)].index < prio) {
        pq->A[pos] = pq->A[PARENT(pos)];
        pq->qp[pq->A[pos].index] = pos;
        pos = PARENT(pos);
    }
    pq->A[pos] = temp;
    pq->qp[index] = pos;

    HEAPify(pq, pos);
}
```

il ciclo while() per mantenere la proprietà dell'HEAP rimane, ma il ciclo for() per cercare l'elemento è eliminato

Alberi binari di ricerca (BST)

RICORDA > un albero è un grafo non orientato, connesso e aciclico.

Non sono altro che una lista, ma invece di next -> left & right

- Accesso tramite root (head)
- Si usano sempre funzioni ricorsive per l'attraversamento

Calcolo di parametri

```
int count(link root) { numero di nodi
    if (root == NULL)
        return 0; conta va dal basso verso alto
    return count(root->left) + count(root->right) + 1;
}

int height(link root) { max profondità di una foglia
    int u, v;
    if (root == NULL)
        return -1;
    u = height(root->left); v = height(root->right);
    if (u>v)
        return u+1;
    return v+1;
}
```

esempio di funzione che attraversa l'albero per contare elementi

Visite su alberi binari avvengono in 3 modi:

- **Pre-ordine** >> radice, sx, dx >> prima chiamo funzioni, poi ricorro sui rami
- **In-ordine** >> sx, radice, dx >> ricorro sx, chiamo funzioni, ricorro dx
- **Post-ordine** >> sx, dx, radice >> ricorro, poi chiamo funzioni

*****fanno riferimento a quando viene stampato il root >>> pre-ordine viene stampato prima, in in mezzo, post dopo le due foglie.*****

Attraversamento **albero completo** (foglie con stessa altezza)

- $O(n)$
- Eq. Ricorrenze >> $T(n) = 1 + 2T(n/2)$ divide and conquer

Attraversamento **albero completamente sbilanciato** (tutto su un ramo, è una lista)

- $O(n)$
- Eq. Ricorrenze >> $T(n) = 1 + T(n-1)$ decrease and conquer

Come possono essere usati?

Ad esempio per rappresentare espressioni algebriche: root è operatore, left & right sono operandi

Es. A + B >>> "+" Root, A & B foglie sx e dx

Alberi binari di ricerca (BST)

La radice fa da separatore tra i valori di sx (< Radice) e dx (> Radice)

- Ogni radice crea un albero i cui valori a sx sono più piccoli, e a dx sono più grandi >> vale per ogni sottoalbero – ricorda che però, laddove io abbia un sottoalbero ad esempio con valore 6, e la foglia dx di valore 10, questa foglia non deve comunque superare la radice assoluta – (ad esempio fosse 11) //// nel caso la superasse, l'albero non è considerato bilanciato.
- **Ogni nodo ha chiavi distinte**

```
BST.h typedef struct binarysearchtree *BST;

BST  BSTinit();
void BSTfree(BST bst);
int  BSTcount(BST bst);
int  BSTempty(BST bst);
Item BSTsearch(BST bst, Key k);
void BSTinsert_leafI(BST bst, Item x);
void BSTinsert_leafR(BST bst, Item x);
void BSTinsert_root(BST bst, Item x);
Item BSTmin(BST bst);
Item BSTmax(BST bst);
void BSTvisit(BST bst, int strategy);
```

funzioni ADT / classe BST

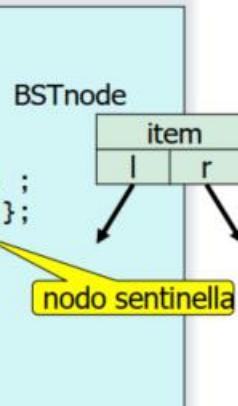
```

BST.c #include <stdlib.h>
#include "Item.h"
#include "BST.h"

typedef struct BSTnode* link;
struct BSTnode { Item item; link l; link r; };
struct binarysearchtree { link root; link z; };

static link NEW(Item item, link l, link r) {
    link x = malloc(sizeof *x);
    x->item = item; x->l = l; x->r = r;
    return x;
}

```



Semplice definizione di un adt di l classe >> + implementazione di nodi (BSTnode) e di un typedef che crea un tipo *link* che non è altro che un *puntatore a nodo* (i parametri di bynarysearchtree potevano semplicemente essere BSTnode *root, BSTnode *r)

APPUNTI SULLE IMPLEMENTAZIONI

- BSTinit() fa malloc() di ADT di l classe (puntatore a struct *vera*), root è null
- BSTfree() libera il puntatore alla struct, DOPO aver prima liberato tutte le foglie + root
 - Questo viene fatto tramite funzione ricorsiva in post-ordine, liberando prima le foglie e poi il root.
 - Il caso di terminazione è in cui il nodo == z (**z è nodo sentinella e viene settato a null all'inizio nell'init**)
- BSTcount() aggiorna un contatore sommando tra loro le chiamate ricorsive su foglia.left e foglia.right – caso terminale return 0
- BSTempty() attraversa tutto l'albero per la conta > O(N), se vuoto O(1)
- BSTsearch() ricerca dicotomica > prende head e z; se valore cercato < head, cerca a sx, sennò dx (*se invece uguale a head, ritorna nodo*)
- BSTmin() e BSTmax() sfruttano il bilanciamento dell'albero -> **tutto a sx** ho elemento min, **tutto a dx** ho elemento max (scorro fino a che ho sentinella a sx, idem dx)
- **BSTinsert()** > ricorsivo o iterativo >> inserimento in base al valore...mantenere bilanciato l'albero
 - **Ricorsiva**

```

static link insertR(link h, int x, link z)
{
    if(h == z)
        return NEW( valore: x, l: z, r: z );
    if(h->valore > x)
        h->left = insertR( h: h->left, x, z );
    if(h->valore < x)
        h->right = insertR( h: h->right, x, z );
    return h; // return h si assicura che il resto dell'albero rimanga invariato > ritorna il nodo
}
void BSTinsert(bst_p bst, int x)
{
    bst->head = insertR( h: bst->head, x, z: bst->z ); // return h >>> torna utile qua
}
      
```
 - **Iterativa** si basa su semplice algoritmo, sempre tramite ricerca dicotomica, per arrivare al nodo cercato con doppio puntatore. Una volta arrivato, creo il nuovo nodo e con confronto sul root (< o >, sx o dx) assegno nuova foglia
 - **COMPLESSITA' >>> Bilanciato O(logn) – Sbilanciato O(n)**
- BSTvisit() è una print dell'albero ma decidendo in che order (0,1,2 pre, in , post) – metti solo degli if dove potrebbe esserci la printf in base a 0,1,2 (prima, in mezzo o dopo chiamate ricorsive)
- **Attenzione >> esistono due inserzioni per i BST – inserzione in foglia ed inserzione in radice. L'inserzione in foglia opera senza rotazioni, e quindi può generare un albero binario di ricerca degenereato in lista. Un inserimento in radice invece permette di usare le rotazioni e quindi creare un albero binario bilanciato a dx e sx.**
- **Rotazione** rispetto ad un punto h
 - Dx > h diventa figlio dx di vecchio ramo sx > i suoi prossimi figli saranno a sx i figli DESTRI (quindi maggiori del vecchio figlio) del vecchio figlio, e a dx i suoi figli normali (i figli del vecchio figlio saranno per forza più piccoli del nodo attuale). Il ramo sx del nuovo padre rimane invariato (sempre più piccolo)

di 10, 10 diventa figlio dx di 5

```
link rotR(link h) {
    link x = h->l;
    h->l = x->r;
    x->r = h;
    return x;
}
```

- Sx > h diventa figlio del vecchio ramo dx > il nuovo padre avrà a sx il vecchio padre, con ramo sx il suo vecchio ramo sx originario e a dx il vecchio ramo sx del nuovo padre. Il ramo dx invece del nuovo padre rimane invariato (sempre più grande)

```
link rotL(link h) {
    link x = h->r;
    h->r = x->l;
    x->l = h;
    return x;
}
```

- Rotazioni servono a bilanciare l'albero > Utilizzata dopo inserimento
- **BSTinsert()** > inserisce al fondo, e poi ruota finché nodo non è al posto giusto (mantiene albero bilanciato)

```
static link insertT(link h, Item x, link z) {
    if (h == z)
        return NEW(x, z, z);
    if (KEYcmp(KEYget(x), KEYget(h->item)) == -1) {
        h->l = insertT(h->l, x, z);
        h = rotR(h); stessa funzione che per inserimento in foglia >  
Ovvero, scendi fino alla foglia e inseriscilo -> una volta che sono lì,  
appena ritorni, ruota subito  
> così ordino albero (ruoterò in ogni chiamata ricorsiva)
    } else {
        h->r = insertT(h->r, x, z);
        h = rotL(h);
    }
    return h;
}

void BSTinsert_root(BST bst, Item x) {
    bst->root = insertT(bst->root, x, bst->z);
}
```

percorre tutto l'albero cercando il posto giusto per x, e poi ruota tutto dopo aver finito di inserire

- L'inserimento in foglia con la **BSTinsert()** permette di rendere l'albero più bilanciato >>>

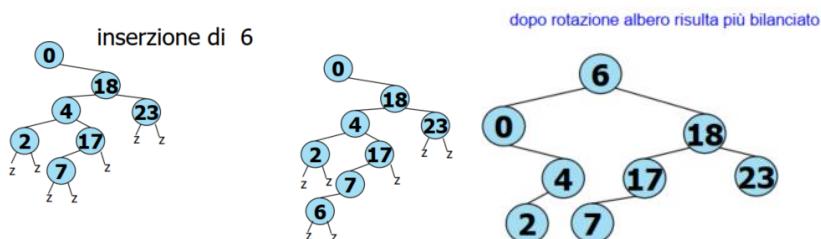


Figura 1 inizialmente >> ad ogni passo scelgo dove scendere e una volta finito, ruoto a sx / dx

Estensioni BST elementari

- Puntatore al padre
- Ogni nodo tiene conto della dimensione del suo sottoalbero (numero di nodi)
- Ne segue che la struct che rappresenta la struttura dati avrà campi aggiuntivi per
 - *Link p* >>> elemento padre
 - *Int cnt* >>> dimensione sottoalbero

Anche le funzioni come **BSTinit()** e **NEW()** saranno modificate per prendere come parametri aggiuntivi l'elemento padre e la dimensione del sottoalbero.

- **Nodo successore** > un nodo è successore di un nodo h se >
 - Se h ha un figlio destro, il *successore* è il nodo con valore più piccolo tra tutti i successori dx di h
 - Se h non ha figli dx, il *successore* è il primo nodo parente il cui figlio sx, dà origine come figlio dx ad h
- **Nodo predecessore** > un nodo è *predecessore* di un nodo h se >
 - Se h ha figli sx, il *predecessore* è il più grande tra i figli dx (il max)
 - Se h non ha figli sx, il *predecessore* è il primo nodo parente il cui figlio dx, dà origine come figlio sx ad h

```
Item searchSucc(link h, key k, link z) {
    link p;
    if (h == z) return ITEMsetNull();
    if (KEYcmp(k, KEYget(h->item))==0) { se ho trovato il nodo padre che cercavo
        if (h->r != z) return minR(h->r, z); se figlio dx di h!=NULL>>> successore è min() dei suoi figli dx
        else { se figlio dx di h!=NULL>>> successore è min() dei suoi figli dx
            p = h->p; nodo padre non ha elementi a dx > cerca in su
            while (p != z && h == p->r) { risali finché il nodo padre ha figli dx,
                h = p; p = p->p; salvando il nodo attuale e il suo padre
            } Quando avrò finito mi rimarrà solo che prendere
            return p->item; il padre, e quello sarà il successore
        }
    }
    if (KEYcmp(k, KEYget(h->item))==-1) { se nò, in base valore, scendi a dx o sx e cerca
        return searchSucc(h->l, k, z); nodo padre
        return searchSucc(h->r, k, z);
    }
}
Item BSTsucc(BST bst, Key k) {
    return searchSucc(bst->root, k, bst->z);
}
```

successore

```
if (KEYcmp(k, KEYget(h->item))==0) {
    if (h->l != z) return maxR(h->l, z);
    else {
        p = h->p;
        while (p != z && h == p->l) {h = p; p = p->p;}
        return p->item;
    }
}
```

procedimento uguale per predecessore > cambia solo selezione del predecessore (che viene fatta con max) e il ciclo while() per selezionarlo tra i nodi parenti

```

int searchSuccessor(link h, link z, int k) {
    link p;
    if (h == z)
        return -1;
    if(h->valore == k)
    {
        if(h->right != z)
            return BSTminR( h: h->left,z);
        else
        {
            /*
             * p = h->p
             * while( p != z && h == p->r)
             *     h = p
             *     p = h->p
             * return p->valore
             */
        }
    }
    else
    {
        if(h->valore < k)
            return searchSuccessor( h: h->right,z,k);
        if(h->valore > k)
            return searchSuccessor( h: h->left,z,k);
    }
}

```

successore

```

int searchSuccessor(link h, link z, int k) {
    link p;
    if (h == z)
        return -1;
    if(h->valore == k)
    {
        if(h->left != z)
            return BSTmaxR( h: h->left,z);
        else
        {
            /*
             * p = h->p
             * while( p != z && h == p->l)
             *     h = p
             *     p = h->p
             * return p->valore
             */
        }
    }
    else
    {
        if(h->valore < k)
            return searchSuccessor( h: h->right,z,k);
        if(h->valore > k)
            return searchSuccessor( h: h->left,z,k);
    }
}

```

predecessore

Come cambiano le funzioni implementate dai BST elementari?

- BSTinsert() > ad ogni chiamata ricorsiva, salva dopo aver ricorso $h \rightarrow l / r \rightarrow p = h$ (se vado a left salvo $h \rightarrow l \rightarrow p = h$ sennò r), e aumento contatore di h dei figli
- **BSTselect()** > Devo cercare la r -esima chiave > come? Sfrutto il contatore della foglia attuale>>
- **La chiave r -esima è la chiave in posizione r nell'ordinamento**

- Se cerco r -esima chiave più piccola, allora cerco in sottoalbero \rightarrow sennò dx
 - $T = \text{numero foglie}$
 - Se r -esima chiave più piccola $>>$ se $t(sx) > r$, scendo in albero sx
 - Se $t(sx) < r$, aggiorno $r = r - t(dx) - 1$ e scendo in albero dx
 - Se $t(sx) == r$, ritorno nodo corrente

```

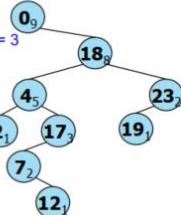
int BSTselectR(link h, link z, int r)
{
    int t;
    if(h == z)
        return -1;
    /*
     * t = h->l->N // contatore dimensione sottalbero totale
     *
     * if(t > r)
     *     return BSTselectR(h->left,z,r)
     * else if(t < r)
     *     return BSTselectR(h->right,z, r - t - 1)
     * else
     *     return h;
    */
}

```

- BSTrotate() >> devo scambiare i nodi padre + aggiornare il padre di uno dei due figli, in base rotazione (dx o sx)
 - Se a dx >>> $x \rightarrow r \rightarrow p = h$
 - Se a sx >>> $x \rightarrow l \rightarrow p = h$
 - **Poi, In entrambi i casi, $x \rightarrow p = h \rightarrow p$**
- BSTinsert() >> ad ogni passo aggiorno semplicemente il contatore, aumentando il numero di foglie totali di 1(ne sto aggiungendo uno)
- **BSTpartition()** > riorganizza l'albero, avendo l'item con la r -esima chiave più piccola in radice
 - Se $t(sx) > r$, ricorro sottoalbero sx e ruoto a dx
 - Se $t(sx) < r$, ricorro sottoalbero dx con $r = r - t(sx) - 1$ e ruoto a sx

Partizionamento rispetto alla 5a chiave più piccola (12, k=4)

```
Inizio da 0> r = 4
t(sx) = 0, 0 > 4, scendo dx con r = 4 - 0 - 1 >> r = 3
t(sx) = 5, 5 > 3 >> scendo sx
t(sx) = 1, 1 < 3 >>> r = 3 - 1 - 1, scendo a dx
t(sx) = 2, 2 > 1 >>> scendo a sx
t(sx) = 0, 0 < 1 >>> r = 1 - 0 - 1   r = 0
scendo a dx >> t(sx) = 0, r = 0
Il valore cercato è 12
Nel frattempo avrò allo stesso tempo segnato quali rotazioni fare, e avrò albero bilanciato
```



```
link partR(link h, int r) {
    int t = h->l->N;
    if (t > r) {
        h->l = partR(h->l, r);
        h = rotR(h);
    }
    if (t < r) {
        h->r = partR(h->r, r-t-1);
        h = rotL(h);
    }
    return h;
}
```

- In un certo senso è lo stesso funzionamento della BSTinsert() – qui però non confrontiamo i valori dei nodi ma la dimensione dei sottoalberi che generano
- **BSTdelete()** >> importante mantenere la proprietà dei BST e la struttura ad albero binario
 - Ricerca del nodo scendendo nel sottoalbero opportuno
 - Quando devo eliminare, elimino il nodo e ricombino i sottoalberi >> nuova radice è successore o predecessore del nodo cancellato
 - Prendo quindi il predecessore /successore, e faccio una partizione rispetto a quel valore > ottengo un albero che ha quel valore in radice
 - Unisco i due alberi

```
link deleteR(link h, Key k, link z) {
    link y, p;
    if (h == z) return z;
    if (KEYcmp(k, KEYget(h->item)) == -1)
        h->l = deleteR(h->l, k, z);
    if (KEYcmp(k, KEYget(h->item)) == 1) ricerca del nodo da eliminare
        h->r = deleteR(h->r, k, z);
    (h->N)--;
    if (KEYcmp(k, KEYget(h->item)) == 0) {nodo trovato
        y = h; p = h->p;salvo il parent
        h = joinLR(h->l, h->r, z); h->p = p;
        free(y);
    }
    return h;
}
void BSTdelete(BST bst, Key k) {
    bst->root = deleteR(bst->root, k, bst->z);
}
```

```
link joinLR(link a, link b, link z) {
    if (b == z) prendi l'elemento più piccolo e mettilo come radice
    return a;
    b = partR(b, 0);
    b->l = a;
    a->p = b;
    b->N = a->N + b->r->N + 1;
    return b;
}
```

aggiornamento puntatore al padre

aggiornamento dimensione sottoalberi

in questo caso ho messo 0 come r per la partition perché l'elemento più piccolo del sottoramo dx sarà sempre più grande di qualsiasi del sottoramo sx > mantengo così la proprietà del BST

- Quindi >> cerco l'elemento, e una volta trovato unisco i due alberi suoi figli >> come ?
 - prendo una partizione del sottoramo dx con r = 0, quindi prendendo l'elemento più piccolo come nuova radice > ho quindi questo nuovo item che ha già un ramo dx, e ci assegno come sx il vecchio ramo sx.

Bilanciamento di un albero

Può essere fatto

- ricorsivamente, intorno chiave mediana inferiore
- algoritmo *Day & Stout & Warren, O(n)* > costruisce albero quasi completo tranne ultimo livello

Ricorsivamente >>> vado a fare una partizione intorno alla mediana > poi ricorro sia su albero sx che dx

```

static link balanceR(link h, link z) {
    int r;
    if (h == z)
        return z;
    r = (h->N+1)/2-1;
    h = partR(h, r);
    h->l = balanceR(h->l, z);
    h->r = balanceR(h->r, z);
    return h;
}

```

Approfondimento > Estensione di una struttura dati

Prima di creare una struttura dati *ex novo*, è bene valutare se è possibile estendere o migliorare una struttura dati già esistente.

- Identificare la struttura dati candidata
- Identificare le informazioni supplementari (da aggiungere)
- Verificare di poter inserire le informazioni supplementari senza compromettere la complessità delle funzioni già esistenti
- Sviluppare nuove operazioni

Ad esempio, il BST Order-Statistic ha come

- Struttura dati candidata > BST
- Info suppl. > Dimensione sottoalbero
- Complessità > O(1)
- Nuova operazione > BSTselect() (partition ecc.)

Interval BST

Bst in cui la chiave è un intervallo chiuso > coppia di numeri reali T1 e T2

- T1 <= T2

Si realizza con una struct semplice, campi low & high (t1 e t2)

Due intervalli t1 e t2 si intersecano se:

- Low(t1) <= high(t2) && low(t2) <= high(t1) (*se l'inizio di uno è minore della fine dell'altro, ovvero rientra in quell'intervallo*)

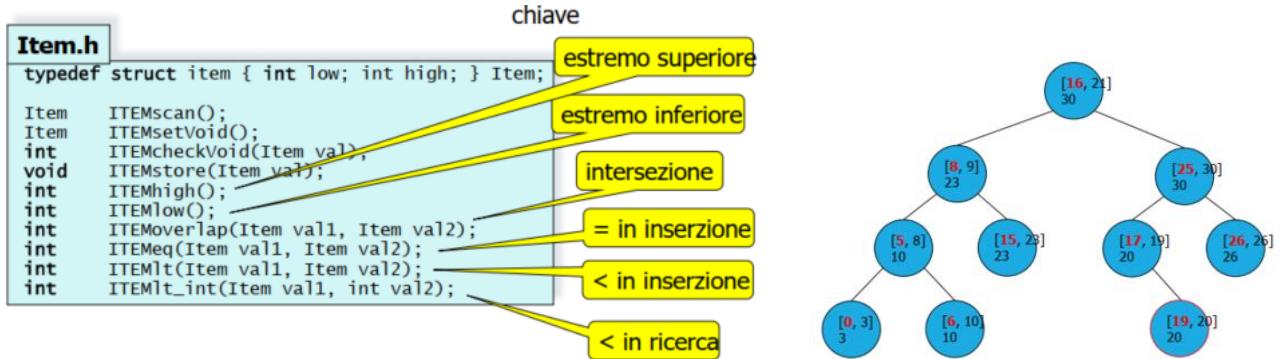
Vale la **tricotomia** seguente dati due intervalli A e B

- 1) A e B si intersecano
- 2) High(A) <= low(B)
- 3) High(B) <= low(A)

Ovvero, o due intervalli si intersecano, o uno inizia prima, o l'altro inizia prima.

Rispetto a un BST normale, propone un parametro supplementare: **max**

- Max propone l'intervallo con estremo maggiore del sottoalbero
- Complessità del calcolo del **max = O(1)**
- Nuova operazione **IBSTsearch(BST, ITEM)**



ADT di I classe Interval BST

IBST.h

```

typedef struct intervalbinarysearchtree *IBST;
void IBSTinit(IBST ibst);
void IBSTfree(IBST ibst);
void BSTinsert(IBST ibst, Item x);
void IBSTdelete(IBST ibst, Item x);
Item IBSTsearch(IBST ibst, Item x);
int IBSTcount(IBST ibst);
int IBSTempty(IBST ibst);
void IBSTvisit(IBST ibst, int strategy);

```

funzioni standard - le inserzioni vengono fatte in base all'intervallo low

Alcune implementazioni delle funzioni classiche degli IBST (ripasso di BST ma con intervalli invece di singoli valori)

- **IBSTinit()** >> crea istanza adt I classe, setto N = 0 ed eventuali parametri aggiuntivi (come un count = 0)
- **IBSTfree()** >> wrapper che richiama funzione ricorsiva su due rami dell'albero, e chiama poi la free()
- **IBSTcount()** >> conta quanti nodi presenti sull'albero – se ho parametro size faccio un semplice return della size
- **IBSTempty()** >> usa IBSTcount() per controllare se vuoto > se == 0, è vuoto e ritorna 1.
- **IBSTvisit()** >> identico a BSTvisit() > *funzione wrapper* per funzione ricorsiva, che in base alla tipologia di visit (pre,in,post) stampa i contenuti dell'albero.
- **IBSTinsert()** >> controllo dicotomico per dove inserirlo (sx o dx) – dopo chiamata ricorsiva aggiorna il valore massimo del nodo corrente nella ricorsione (chiama max nel caso il nuovo valore inserito superi tutti gli altri di quel ramo)
- **IBSTrotR() / IBSTrotL()** >> *implementazioni uguali a BST normali* – con eccezione che *vengono ricalcolati max per i due nodi scambiati* (vengono anche scambiati i contatori dei figli)

```

link rotL(link h)
{
    link x = h->right; // rotR > link x = h->left
    h->right = x->left; // rotR > h->left = x->right
    x->left = h; // rotR > x->right = h
    // h->max = max(ITEMhigh(h->item), h->l->max, h->r->max);
    // x->max = max(ITEMhigh(x->item), x->l->max, x->r->max);
    // ITEMhigh ritorna il valore contenuto nel campo HIGH
    return x;
}

```

- **partR()/joinLR()** >> *implementazione uguale a BST normale* – metto solo prima di fare la *return* un ricalcolo del max del risultato (in questo caso b->max viene ricalcolato)
- **IBSTdelete()** >> *implementazione uguale a BST normale* – prima ricocco il nodo attuale (e dopo ogni chiamata ricorsiva ricalcolo max) e poi faccio la *joinLR*, e poi diminuisco il size del risultato di 1.
- **IBSTsearch()** >> *cerco un nodo H che interseca l'intervallo i (complessità sarà sempre logn)*

- Controllo se i due intervalli si intersecano > **se sì, search hit, sennò >>**
- Se max ramo sx \geq low[i] allora vado in sottoalbero sinistro
- Se max ramo dx $<$ low[i] allora vado in sottoalbero dx
- La complessità è **R*logN** nel caso in cui voglia trovare tutti gli R intervalli che ne intersecano uno dato

+ Funzioni specifiche

- **IBSTsearch()** > cerca il primo item che interseca l'intervallo cercato
 - **IBSTinsert()** > inserisci un item (intervallo) nell'IBST
 - **IBSTdelete()** > cancella un item dal IBST
-

TABELLE DI HASH

Tutti algoritmi di ricerca finora si basano sul confronto (*eccezione per tabelle ad accesso diretto in cui ho relazione univoca tra chiave e indice – però questo comporta un grande utilizzo di memoria in caso di numeri o valori molto grandi, maxN*)

- Discorso già affrontato, per un universo di valori molto grandi, ho bisogno di molta memoria e potrei anche sprecarne poiché basta un solo valore grande *maxN* per allocare un vettore di tale grandezza.

Le tabelle di Hash sono un tentativo di riprodurre le tabelle ad accesso diretto, ma riducendo l'occupazione di spazio.

Questo viene fatto tramite un vettore, e una funzione di Hash, che ricava dal valore attuale l'indice della tabella in cui verrà poi inserito.

- Le tabelle di hash permettono quindi di ottenere una complessità molto bassa, al costo però di una possibile **collisione nell'inserimento di valori**
- Questo perché la funzione di hash potrebbe assegnare ad un valore un indice che è già occupato da un altro valore.

La funzione di hash viene usata per *inserzione, ricerca, cancellazione*.

Non viene usata per *l'ordinamento e selezione*.

ADT di I classe ST

```
ST.h    typedef struct symboltable *ST;  
  
ST      STinit(int maxN, float r) ;  
void   STinsert(ST st, Item val);  
Item   STsearch(ST st, Key k) ;  
void   STdelete(ST st, Key k) ;  
void   STdisplay(ST st) ;  
void   STfree(ST st);  
int    STcount(ST st);  
int    STempty(ST st);
```

Le funzioni di Hash

Le funzioni di hash permettono di non basarsi più sul confronto per trovare un indice valido ad un elemento, ma lo ricavano direttamente grazie ad una funzione.

- Questa funzione dovrà produrre un indice tra gli intervalli 0 e M-1, dove M è la *dimensione della tabella*.
- è preferibile come dimensione della tabella un numero primo
- La funzione di hash, o almeno una buona, deve *distribuire in modo uniforme* un insieme di chiavi su un vettore.
 - **gli indici devono essere equiprobabili** > non risulta facile poiché diverse chiavi potrebbero essere comunque correlate tra loro > la funzione di hash deve quindi prima scorrelarle (magari moltiplicando un numero per due numeri primi – **aumentando le differenze con altre possibili chiavi**) e poi calcolare l'indice corrispettivo.

Come stabilisco una funzione di hashing?

- 1) **Metodo moltiplicativo** > prendo due intervalli **s** e **t**, in cui è compreso il valore di cui fare l'hash **k**. **M** è la dimensione della tabella.

$$h(k) = (k - s) / (t - s) * M$$

- 2) **Metodo modulare** > se ho un intervallo anche più grande di M, il risultato di N%M sarà sicuramente minore di M.

$$h(k) = k \% M$$

- *metodo efficace*
- *problema di hashing di stringhe non effettivo, numeri troppo grandi*
- *dimostrato in metodo di Horner per calcolare hash di una stringa, vista come una serie di caratteri > trasformo prima la stringa in numero, ma genera numeri troppo grossi...*

- **Soluzione > calcolo il modulo di M ad ogni iterazione**

Funzione di hash per chiavi stringa con base prima:

```
int hash (char *v, int M) {
    int h = 0, base = 127;
    for (; *v != '\0'; v++)
        h = (base * h + *v) % M;
    return h;
}
```

valutazione di horner in base 127 (%M)

Funzione di horner - hash stringhe

Vi è poi un'altra versione della valutazione di Horner, che permette di **cambiare la base ad ogni iterazione**, scorrelando ancora di più ogni indice dall'altro.

```
int hashu( char *v, int M) {
    int h, a = 31415, b = 27183;
    for ( h = 0; *v != '\0'; v++, a = a*b % (M-1))
        h = (a*h + *v) % M;
    return h;
}
```

RICORDA: per l'implementazione delle funzioni di hash, ci basiamo su questi due esempi, naturalmente *adattati alle rispettive versioni per int / float ecc.*

- *Le diamo quindi per definite nelle implementazioni dell'ADT hash table*

COLLISIONI

Collisioni sono inevitabili – 2 soluzioni

- 1) **Linear chaining** >> ogni indice contiene un vettore
- 2) **Open addressing** >> funzione di hash calcola un indice finché non ne trova uno libero

Linear chaining – più elementi in una posizione

Il linear chaining permette di inserire, all'interno di una posizione di una Hashing table, una lista di oggetti > *lista concatenata*

- Inserzione in testa, ricerca, cancellazione

La lunghezza delle liste va stabilita a priori – devono comunque rimanere *corte ed equilibrate tra loro* (una buona lunghezza è nelle unità, max 5 ad esempio)

```
struct symbtab { link *heads; int N; int M; link z; };
```

La tabella di simboli non è quindi altro che una lista di liste > *heads* è una lista di puntatori a liste, ovvero a ogni *head* di ogni lista

Alcune delle implementazioni delle hashing table per il linear chaining

- STInit() >> alloca la struct wrapper + alloca vettore di *liste*, *heads*. Inizializza poi ogni *heads*[*i*] a NULL;
 - **M** è assegnato tramite una funzione apposita, STsetSize
- **STsetSize()** >> restituisce una possibile dimensione **M** della tabella >> questo viene fatto dividendo il massimo valore da contenere nella tabella per R (dimensione di ogni lista) > il primo numero primo più grande di questo numero, verrà usato come dimensione **M**.
- STfree() >> libera prima ogni elemento di ogni lista, poi l'insieme delle liste *heads*, e poi la struct wrapper.
- STcount() >> ritorna N
- STempty() >> se N == 0, ritorna 1
- STinsert() >> chiama funzione di hash (*una delle due definite prima*), poi inserisce in *heads*[*i*], dove *i* è l'indice ottenuto dalla tabella di hash. L'inserzione è fatta inserendo in testa, come nelle liste semplici. (*sposto in avanti il vecchio nodo e lo metto come next a quello nuovo, poi aggiorno head*)
- **STsearch()** >> richiama funzione ricorsiva **searchR()**, a cui passo direttamente *heads* in posizione *i* > la posizione è il risultato dell'*hashing della chiave ricercata*
 - **searchR()** >> semplice ricerca su lista, O(*R*) in peggio dei casi
- **STdelete()** > richiama funzione ricorsiva **deleteR()** che ricerca in *heads* in posizione *i* (come ricerca) ed elimina. La funzione STdelete() riassegna a *heads*[*i*] il valore di ritorno di deleteR(), che ritorna il nodo corrente, così che se elimino il primo nodo non vado a perdere la lista.

```
void STdelete(ST st, Key k) {
    int i = hash(k, st->M);
    st->heads[i] = deleteR(st->heads[i], k); riassegno perché è possibile
} che venga cancellata la prima

link deleteR(link x, Key k) {
    if (x == NULL) return NULL;
    if ((KEYcmp(KEYget(&x->item), k)) == 0) {
        link t = x->next; free(x); return t;
    }
    x->next = deleteR(x->next, k);
    return x;
}
```

- STdisplay() >> per ogni *heads*, ovvero fino a **M**, richiama funzione ricorsiva di visitR() su ogni *heads* > semplice percorrimento lista

Complessità

L'hashing è quasi unitario (piccolo ciclo for() per calcolare l'hash), ciò che conta davvero è *la dimensione delle liste*.

- Inserimento > **O(1)**, inserimento in testa + costo quasi unitario di hashing
- Ricerca > **O(N)** caso peggiore, tutti gli elementi sono in un'unica lista
 - Caso medio >> **O(1+a)** dove **a** è il fattore di carico, che indica la dimensione massima di una lista.
- Cancellazione > **O(1)**, simile a ricerca

Open addressing – un elemento per posizione (*vettore di struct*)

Ogni cella può contenere un solo elemento

- **N <= M**
- **Probing >>** gestisco la collisione generando un'altra possibile casella per il valore da inserire, finché non ne trova una valida.

```
...  
struct symboltable { Item *a; int N; int M;};
```

La struttura dati è quindi un semplice vettore di item, o struct qualsiasi.

Alcune delle implementazioni...

- STinit() > alloca struct wrapper + calcola **M** con stessa funzione STsizeset(), e alloca *vettore di Item* con dimensione *M*

Funzioni di probing

- Linear probing
- Quadratic probing
- Double hashing

Per quanto riguarda il probing, ovvero il tentativo di trovare una cella non occupata per la funzione di hashing, si può ricorrere nel problema di *clustering* >

- *Raggruppamento di posizioni occupate contigue*

1) *Linear probing*

Calcola $h(k)$ >> se quella posizione è occupata, allora nuovo indice sarà $i = (i+1)\%M$

Continua finché non trovi posizione valida

- **Inserzione >>** fa uso di funzione full(i), che controlla se posizione è occupata. Continua finché non ne trova una libera.
- **Search >>** compara valore con $h(k)$ iniziale, e poi lo aumenta e continua a comparare finché non trova valore giusto (questo sempre se la cella attuale è piena, usando full())
- **Delete() >>** più problematica, se elimino oggetto a metà di un cluster, rischio di non poter raggiungere più quelli dopo che magari stavano in posizione precedentemente occupata (ricerca si ferma nelle celle vuote)

La delete() non si eseguirebbe normalmente nelle tabelle di hashing con *open addressing* perché causa problemi soprattutto ai cluster. Si può però

- **Usare un flag per le celle “eliminate”, e la funzione full() controlla quel flag**

- Funzioni `checkFull()`, `checkDeleted()` >> controllano se `flag == 0` (vuota) oppure `(-1) > se sì, inserisci lì, sennò ritorna 1`
 - La `delete()` settnerà semplicemente la struct in quell'indice a -1
- Ad ogni eliminazione, re-inserire tutte le chiavi successive nei rispettivi indici. Ricreto quindi ogni cluster ad ogni eliminazione.
 - STdelete() >> prima cerca valore, una volta trovato, lo elimina. Poi fa partire un ciclo for() che, finché gli elementi **dopo quello eliminato non sono vuoti, li salva, li elimina, e li re-inserisce.**

LINEAR PROBING >> COMPLESSITÀ'

search hit: $1/2(1 + 1/(1-\alpha))$ Hit si ferma naturalmente prima

search miss: $1/2(1 + 1/(1-\alpha)^2)$ Miss avviene quando arrivo alla fine di un cluster

Linear probing non adatto a riempimento totale della tabella >> inserimento a tabella piena per il 90% ha complessità eccessiva

2) Quadratic Probing

Quadratic probing prova ad evitare di creare cluster troppo grossi – scopo è ridurre complessità a tabella riempita maggiormente

Quando trovo casella piena, invece di cercarne una accanto, mi allontano.

- Tende quindi a scorrelare un po di più maggiore è la quantità di tentativi fatti
- Come?
 - Prendo due numeri **c1** e **c2**, e ad ogni tentativo l'indice diventa $(j + c1*j + c2*j*j)\%M$, dove **J** è il numero di tentativi
 - Inserzione > calcolo hash – se è pieno, aumento numero tentativi e aggiorno l'indice.
 - Ricerca > identico
 - Delete() > prima cerco elemento, poi lo elimino e inserisco tutti gli altri > uso però come indice il nuovo metodo
- Come scelgo **c1** e **c2** per il quadratic probing?
 - Se M divisibile per 2 >> $c1 = c2 = \frac{1}{2}$
 - Se M primo >> $c1 = c2 = \frac{1}{2}$, oppure $c1 = 0$ e $c2 = 1$, oppure 1 per entrambi

3) Double hashing

Aumenta ancora di più la scorrelazione maggiore è il numero di tentativi

Attenzione – i due hash $h(k)$ e $h'(k)$ vengono calcolati solo una volta, vengono poi riutilizzati varie volte.

- Inserzione > prova con $h(k)$ – se indice è occupato, calcola secondo indice $h'(k)$ con altra funzione di hashing, e lo somma ad $h(k)$ e poi fa $\%M$
 - $h(k) = i; h'(k) = j; i = (i+j)\%M$

- Attenzione! Il nuovo valore DEVE essere diverso dal vecchio i. sennò loop infinito + $h'(k)$ NON DEVE essere 0.

```
static int hash2(key k, int M) {
    int h = 0, base = 127;
    for ( ; *k != '\0'; k++) h = (base * h + *k);
    h = ((h % 97) + 1)%M;
    if (h==0) h=1; il secondo h non deve essere 0
    return h;
}
```

si può facilmente controllare tale caso mettendo un controllo nel caso il valore sia 0 - e ritornare 1. Così il vecchio valore di i non sarà neanche mai uguale al precedente

- **Ricerca > calcolo i due hash** – controlla il primo valore con il primo hash, sennò aumentalo con $(i+j)\%M$ e continua a comparare valori.
- **Delete()** > procedimento analogo – cambia solo l'indice e come viene calcolato, ovvero con due funzioni di hash invece di una.

Complessità – double hashing

Il double hashing, come previsto, permette di scorrerare molto di più gli indici assegnati a ciascuna chiave. Da questo risultano quindi meno cluster, e quindi una maggiore efficienza per quanto riguarda tabelle di hashing più piene.

- search miss: $1/(1-\alpha)$
- search hit: $1/\alpha \ln (1/(1-\alpha))$

α	1/2	2/3	3/4	9/10
hit	1.4	1.6	1.8	2.6
miss	1.5	2.0	3.0	5.5

Tabelle di Hashing o Alberi binari (BST ecc.)?

Tabelle di hashing >>>

- unica soluzione per chiavi senza relazione d'ordine,
- facili da realizzare
- più veloci per chiavi semplici.

Alberi binari >>>

- alberi bilanciati hanno prestazioni più stabili (*anche per chiavi meno semplici*)
 - permettono operazioni su insiemi con relazioni d'ordine.
-

ADT GRAFO

Grafi trovano moltissime applicazioni pratiche, centinaia di algoritmi – astrazione utilizzabile in molti domini diversi

Problemi tipici di grafi:

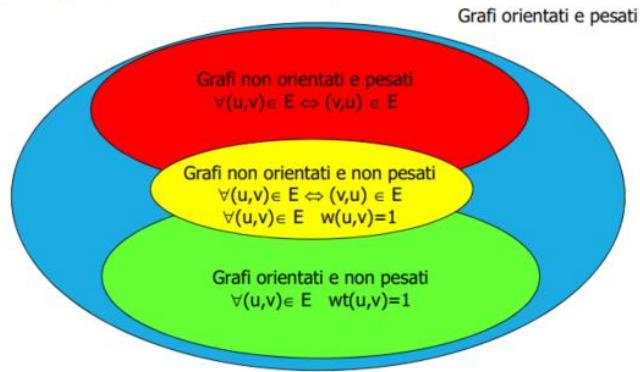
- stabilire se *un grafo è connesso*
- stabilire la *presenza di un ciclo*
- individuare *componenti fortemente connesse*
- individuare *alberi ricoprenti minimi*
- *cammini minimi*
- determinare se un grafo è *bipartito*
- *cammino di Eulero*

Problemi intrattabili >

- cammino massimo
- colorabilità > dato un grafo non orientato, qual è il minimo numero di colori K affinchè nessun vertice abbia lo stesso colore di un vertice ad esso adiacente (es. mappe geografiche)
 - introdotto **grafo planare** > grafo i cui archi non si intersecano tra loro.
 - è dimostrato che per i grafi planari servono al massimo 4 colori.
- ciclo di Hamilton > dato un grafo non orientato, esiste un ciclo semplice che visita ogni vertice una sola volta
- problema del commesso viaggiatore > passare per tutti i vertici e tornare in origine, con cammino minimo

Un grafo può essere

- **orientato** > gli archi hanno una direzione di percorimento
- **pesato** > gli archi hanno tutti un peso
 - *i grafi non pesati sono semplicemente grafi i cui pesi di tutti gli archi è 1*



- 1) **Statico** > non si aggiungono né cancellano vertici / archi
- 2) **Semi-statico** > si possono aggiungere / cancellare archi (si segnano eliminati con un flag)
- 3) **Dinamico** > si possono aggiungere / cancellare archi e vertici

RICORDA > grafo è completo se i suoi vertici sono adiacenti a due a due. Ovvero se ogni vertice ha un arco che lo connette a tutti gli altri vertici

Come rappresento i vertici?

- Interi, per identificarli ai fini dell'algoritmo > agli interi posso far corrispondere gli indici di un vettore
 - Anche se memorizzati come stringhe, negli algoritmi userò interi, magari creo una tabella di simboli esterne al grafo per memorizzare i nomi
 - Tabella di simboli funzioni STsearch(nome) -> mi ritorna indice
 - STsearchByIndex(indice) -> mi ritorna nome
 - Posso usare anche BST / Hash table
- Si può anche usare una tabella di simboli formata da un vettore non ordinato di stringhe > **l'indice del vettore coincide con l'indice del vertice**
- **BST / Hash table** > l'indice del vertice è memorizzato esplicitamente. STsearchByindex() scansione lineare di un vettore in corrispondenza indice – chiave.

Lista archi

ST
A B
B C
B D
A E
C D
B E
D E

ST
0 A
1 B
2 C
3 D
4 E

ST = tabella di simboli con corrispondenza vertice = posizione e come valore il suo nome

Numero di vertici >

- Letto da file
- Al massimo è $2 * N$ archi
- I vertici sono letti con **GRAPHload / GRAPHstore**

ADT di classe Grafo

Graph.h

```

typedef struct edge { int v; int w; int wt; } Edge; archi
typedef struct graph *Graph; contiene tabella di simboli
Graph GRAPHinit(int V);
void GRAPHfree(Graph G);
void GRAPHload(FILE *fin);
void GRAPHstore(Graph G, FILE *fout);
int GRAPHgetIndex(Graph G, char *label);
void GRAPHinsertE(Graph G, int id1, int id2; int wt; );
void GRAPHremoveE(Graph G, int id1, int id2);
void GRAPHedges(Graph G, Edge *a);
int GRAPHpath(Graph G, int id1, int id2);
void GRAPHpathH(Graph G, int id1, int id2);
void GRAPHbfs(Graph G, int id);
void GRAPHdfs(Graph G, int id);
int GRAPHcc(Graph G);
int GRAPHscc(Graph G);

```

breadth-first-search
depth-first-search

grafo pesati

indice dato nome

grafo pesati

grafo non orientato

grafo orientato

Come rappresento un grafo?

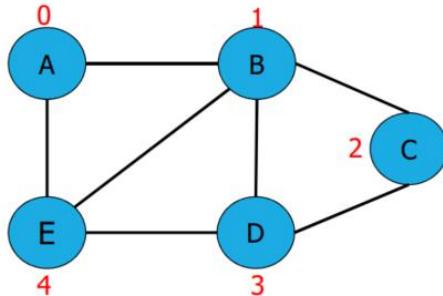
- Matrice delle adiacenze
- Lista delle adiacenze
- Elenco di archi

1) Matrice di adiacenza

Rappresento come matrice > se il vertice **i** è adiacente al vertice **j**, allora in posizione matrice[i][j] avrà 1 (*o il peso dell'arco*)

- Grafi non orientati >> matrice simmetrica
- Grafi pesati > matrice[i][j] ha come valore il peso di un arco che tende tra *i* e *j*

Non orientato non pesato



Lista archi

	ST
0	A
1	B
2	C
3	D
4	E

G->adj

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

matrice di adiacenze per
grafi non orientati è una
matrice simmetrica

	0	1	2	3	4
0	0	3	0	0	5
1	3	0	5	4	6
2	0	5	0	7	0
3	0	4	7	0	1
4	5	6	0	1	0

matrice simmetrica ma
al posto di 1 per
segnare presenza di
arco ho direttamente il
peso degli archi che li
collegano

	0	1	2	3	4
0	0	3	0	0	0
1	0	0	5	4	0
2	0	0	7	0	0
3	0	0	0	0	1
4	5	6	0	0	0

matrice non
simmetrica, al posto di
1 per rappresentare la
presenza di un arco ho
direttamente il suo
peso

grafo non orientato, pesato

grafo orientato e pesato

```

Graph.c numero di archi
          ... matrice di adiacenza
          numero di vertici tabella di simboli
          struct graph {int v; int E; int **madj; ST tab;};
static int **MATRIXint(int r, int c, int val) {
    int i, j;
    int **t = malloc(r * sizeof(int *));
    for (i=0; i < r; i++) t[i] = malloc(c * sizeof(int));
    for (i=0; i < r; i++)
        for (j=0; j < c; j++)
            t[i][j] = val; inizializza matrice di adiacenza e mette tutti i valori ad un val scelto
    return t;
} crea l'arco, ma per inserirlo dentro grafo serve funzione che aggiorni la matrice di adiacenze
static Edge EDGEcreate(int v, int w, int wt) {
    Edge e;
    e.v = v; e.w = w; e.wt = wt;
    return e;
}

```

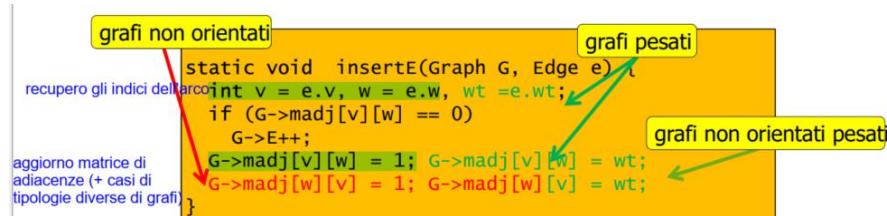
MATRIXint() inizializza la matrice di adiacenza dell'ADT di I classe graph

`MatrixINT()` verrà usata da `GRAPHinit()` laddove io scelga di utilizzare una matrice di adiacenza.

```
Graph GRAPHinit(int v) {
    Graph G = malloc(sizeof *G);
    G->V = V;
    G->E = 0;
    G->madj = MATRIXint(V, V, 0);
    G->tab = STinit(V);
    return G;
}
```

Come inserisco gli archi nella matrice di adiacenza?

- Scorro su tutti gli archi, e metto in posizione `matrice[vertice1][vertice2] = 1` (se non pesato), sennò metto il peso dell'arco.
- Se il grafo è non orientato, metterò il valore anche in `matrice[vertice2][vertice1]`
- Per rimuoverli, prima controllo che `matrice[vertice1][vertice2] != 0` poi setto a 0 e diminuisco il numero di archi (`G->E--`)



GRAPHgetIndex > prendi indice di un vertice data una stringa s

- Cerca in tabella di simboli con `STsearch()` >> se non c'è, conta quanti vertici ci sono e inseriscilo
- Ritorna quindi l'id trovato – oppure l'id appena inserito

```
int GRAPHgetIndex(Graph G, char *label) {
    int id;
    id = STsearch(G->tab, label);
    if (id == -1) {
        id = STcount(G->tab);
        STinsert(G->tab, label, id);
    }
    return id;
}
```

GRAPHedges > inserisce in un vettore tutti gli archi di un grafo

```
graph TD
    subgraph Right [Categorization]
        direction TB
        R1[grafi orientati]
        R2[grafi pesati]
        R3[grafi non orientati pesati]
    end
    Center[void GRAPHedges(Graph G, Edge *a)] --> R1
    Center --> R2
    Center --> R3
    R3 --> Center
```

```
void GRAPHedges(Graph G, Edge *a) {
    int v, w, E = 0;
    for (v=0; v < G->V; v++)
        for (w=v+1; w < G->V; w++)
            if (G->madj[v][w] != 0)
                a[E++] = EDGEcreate(v, w, G->madj[v][w]);
    return;
}
```

COMPLESSITÀ'

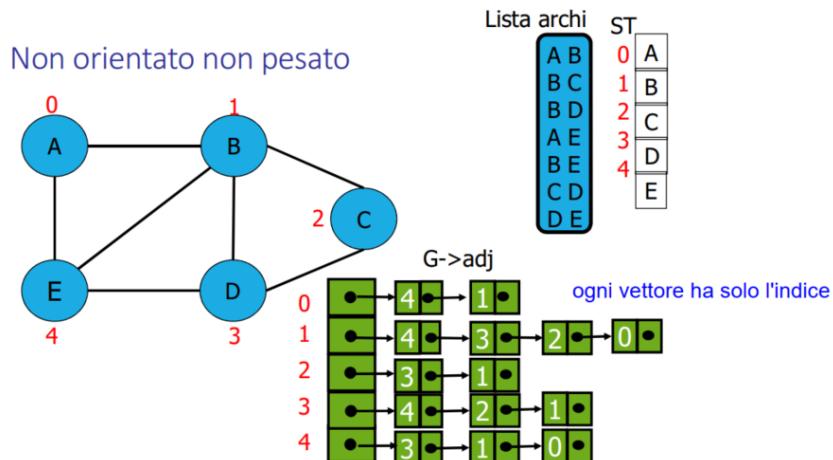
Le matrice di adiacenze sono vantaggiose per grafi densi > $O(V^2)$

- No costi aggiuntivi per grafi pesati
- Accesso efficiente $O(1)$ per controllare se esiste un arco o meno
 - $\text{Matrice}[i][j] \neq 0 ?$
-

2) LISTE DI ADIACENZA

Dato infatti un grafo $G = (V, E)$ con V numero di vertici ed E numero di archi

- **Lista di adiacenza** $A = |V|$ elementi
 - $A[i]$ contiene puntatore alla lista dei vertici adiacenti ad i
 - La lista di adiacenza è una vettore di liste
 - Ho quindi bisogno di una struct wrapper per il grafo + struct nodo



```
for(int i = 0; i < G->V; i++)
{
    link tmp = G->adj[i];
    printf("\n Stampo tutti gli archi per il vertice %d",i)
    while(tmp != NULL)
    {
        printf("\n %d-%d", i , tmp->val);
        tmp = tmp->next;
    }
}
```

esempio di stampa di tutti gli archi per tutti i vertici con lista di adiacenza

```

Graph.c
typedef struct node {
    int v; int wt; link next; } ;
struct graph{int V; int E; link *ladj; ST tab; link z;} ;
crea vertice lo mette in testa -> a lista contenente tutti vertici adiacenti a quello corrente
static link NEW(int v, int wt, link next) {
    link x = malloc(sizeof *x);
    x->v = v; x->wt = wt; x->next = next;
    return x;
}
static Edge EDGEcreate(int v, int w, int wt) {
    Edge e;
    e.v = v; e.w = w; e.wt = wt;
    return e;
}

```

numero di vertici
numero di archi
lista di adiacenza
tabella di simboli
sentinella
grafo pesati

Nella GRAPHinit() avrò naturalmente una malloc per **ladj**, di dimensione **V**.

- [sizeof\(link\) \(puntatore a struct\)](#)
- [setto tutto inizialmente a NULL](#)

GRAPHedges() diventa un semplice attraversamento di tutte le liste contenute dentro **ladj**

```

void GRAPHedges(Graph G, Edge *a) {
    int v, E = 0;
    link t;
    for (v=0; v < G->V; v++)
        for (t=G->ladj[v]; t != G->z; t = t->next)
            if (v < t->v) a[E++] = EDGEcreate(v, t->v, t->wt);
}

```

grafo pesati
grafo non orientato

GRAPHinsertE() semplice inserimento in testa in una lista >> O(1)

```

static void insertE(Graph G, Edge e) {
    int v = e.v, w = e.w, wt = e.wt;
    G->ladj[v] = NEW(w, wt, G->ladj[v]);
    G->ladj[w] = NEW(v, wt, G->ladj[w]);
    G->E++;
}

```

grafo pesati
grafo non orientato (pesati)

GRAPHdeleteE() semplice eliminazione da lista > O(N) in caso peggiore > scorro con doppio puntatore, quando l'ho trovato, se sono in testa, aggiorno direttamente la testa al prossimo di quello da eliminare

- sennò, aggiorno il valore del puntatore a quello corrente con $p->next = current->next$

```

static void removeE(Graph G, Edge e) {
    int v = e.v, w = e.w;  link x, p;
    for (x = G->ladj[v], p = NULL; x != G->z; p = x, x = x->next) {
        if (x->v == w) {
            if (x == G->ladj[v]) G->ladj[v] = x->next; se sono in testa, aggiorna direttamente
            else p->next = x->next; la head della lista adiacenza per quel
                                         vertice
            break; sennò aggiorna il doppio puntatore
        }
    } cerco l'altro arco -> se cerco v-w devo cercare anche w-v
    for (x = G->ladj[w], p = NULL; x != G->z; p = x, x = x->next) {
        if (x->v == v) {
            if (x == G->ladj[w]) G->ladj[w] = x->next;
            else p->next = x->next;
            break;
        }
    }
    G->E--; free(x);
}

```

grafo non orientato

VANTAGGI

- *grafo non orientato > elementi $2|E|$*
- *grafo orientati > elementi $|E|$*
- *Complessità spaziale > $|V + E| >> \underline{\text{vantaggioso per grafi sparsi}}$*

Svantaggioso verificare l'adiacenza di 2 vertici tramite scansione di lista di adiacenza > $O(N)$, normale scansione di lista + uso di memoria per i pesi dei grafi pesati

Generazione di grafi – e perché

Generalmente, i grafi sono utilizzati come modelli di situazioni reali e vengono forniti come dati in ingresso.

- In alternativa, si può generare:
 - un grafo casuale con vertici tra 0 e $V-1$, Archi come coppie casuali di interi tra 0 e $V-1$
 - Archi con probabilità $p >$ data una probabilità P , si calcola un numero di archi $E = p*(V*(V-1)/2)$

PROBLEMA COMUNE – CAMMINO SEMPLICE

Dato un grafo non orientato $G = (V, E)$ e 2 suoi vertici v e w , esiste un cammino semplice che li connette?

- Se connesso >> basta trovarne uno, senza backtrack
- Se non connesso >> cammino esiste se vertici sono nella stessa componente连通 > sennò non esiste.

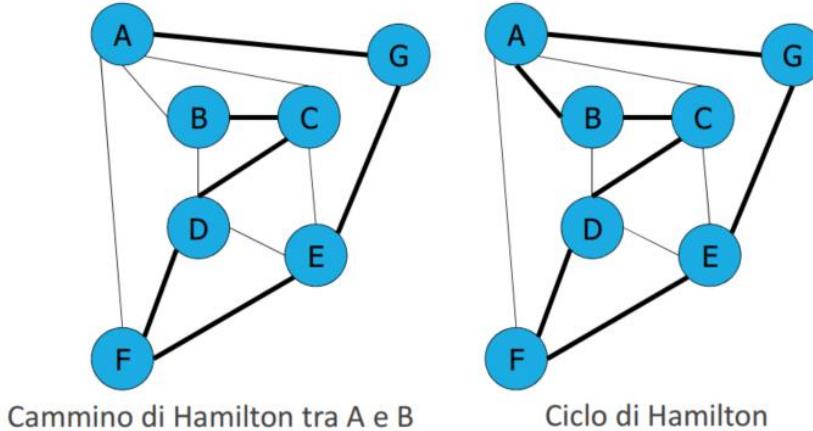
Come? Determino se esiste un cammino semplice ricorsivamente da t -> w

- Uso un array per segnare i nodi già visitati
- **O(V+E)**
- **Funzione ricorsiva** a cui passo nodo iniziale > guarda i nodi adiacenti non ancora visitati, e ne visita uno > chiama ricorsivamente la funzione con nodo iniziale il nodo adiacente che sto prendendo in considerazione
 - **Caso di terminazione** > ho raggiunto il nodo w

CAMMINO HAMILTONIANO

Dato un grafo non orientato $G = (V, E)$ e due vertici v,w determinare se esiste un cammino semplice che li connette visitando *ogni vertice una sola volta*

- Se $v == w$, allora *ciclo di hamilton* (deve tornare a punto di partenza dopo aver visitato tutti nodi una sola volta)



Complessità esponenziale

Come?

- Uso vettore mark[] per segnare nodi già visitati, procedimento simile a cammino semplice
- Risultato accettabile solo se lunghezza del cammino == $V-1$

```

void GRAPHpath/GRAphpathH(Graph G, int id1, int id2) {
    int t, found, *visited;
    visited = malloc(G->v*sizeof(int));
    for (t=0; t<G->v; t++)
        visited[t]=0;
    if (id1 < 0 || id2 < 0)
        return;
    found = pathR/pathRH(G, id1, id2, G->v-1, visited);
    if (found == 0)
        printf("\n Path not found!\n");
}

```

Attenzione: per sinteticità si viola la sintassi del C

funzione wrapper - inizializza mark[] e chiama funzione principale – id1 e id2 sono i vertici da cui stabilire il cammino hamiltoniano

```

static int pathR(Graph G, int v, int w, int *visited) {
    int t;
    if (v == w)
        return 1;
    visited[v] = 1;
    for (t = 0 ; t < G->v ; t++)
        if (G->madj[v][t] == 1) se c'è arco tra due
            vertici e quel vertice
            if (visited[t] == 0) non è visitato
                if (pathR(G, t, w, visited)) {ricorro su quel vertice
                    printf("(%s, %s) in path\n",
                        STsearchByIndex(G->tab, v),
                        STsearchByIndex(G->tab, t));
                }
    return 1;
}
return 0;
}

```

matrice delle
adiacenze

stampo gli archi
del cammino in
ordine inverso

```

static int pathRH(Graph G, int v, int w, int d, int *visited) {
    int t;
    if (v == w) {
        if (d == 0) return 1;
        else return 0;
    }
    visited[v] = 1;
    for (t = 0 ; t < G->V ; t++)
        if (G->adj[v][t] == 1) se vertice attuale è connesso ad
            if (visited[t] == 0) altro e non è visitato,
                if (pathRH(G, t, w, d-1, visited)) {
                    printf("(%s, %s) in path \n", STsearchByIndex(G->tab, v),
                           STsearchByIndex(G->tab, t));
                    return 1;
                }
    visited[v] = 0;
    return 0;      backtrack
}

```

intero che indica quanto manca a un cammino lungo $|V|-1$

matrice delle adiacenze

se vertice attuale è connesso ad altro e non è visitato, ricorri e diminisci la distanza

stampo gli archi del cammino in ordine inverso

backtrack

CAMMINO DI EULERO

Dato un grafo non orientato $G = (V, E)$ e 2 suoi vertici v, w si dice *cammino di Eulero* un cammino anche non semplice che li connette, attraversando *ogni arco una sola volta*.

Se $v == w$, si dice ciclo di eulero.

Lemmi >

- Un grafo non orientato ha un *ciclo di Eulero* solo se è connesso e tutti i suoi vertici sono di grado pari
- Un grafo non orientato ha un *cammino di Eulero* solo se è connesso e se esattamente due vertici hanno grado dispari.

ALGORITMI DI VISITA DEI GRAFI

Due tipologie di algoritmi:

- In profondità (depth-first search, **DFS**)
- In ampiezza (breadth-first search, **BFS**)

1) Visita in profondità > DFS

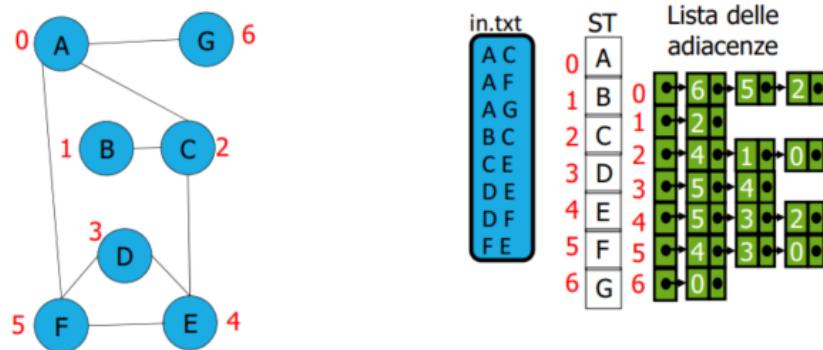
Principio base > espandere l'ultimo vertice scoperto che ha ancora vertici non scoperti adiacenti.

Mi posiziono su un vertice, e ricorsivamente esploro tutti i suoi vertici adiacenti >> se non sono ancora stati visitati, li visito, e li segno come visitati nel vettore *pre[]* (prima era *mark[]*)

Implementazione

Funzione wrapper GRAPHsimpleDFS > crea un vertice fittizio che ha come adiacente il primo vertice del grafo, così da poter applicare subito la funzione ricorsiva di esplorazione.

SimpleDfsR > funzione ricorsiva di esplorazione di vertici – ci passo tra altri parametri il vettore *pre[]*



Parto da 0, guardando la lista di adiacenze > ha come vertici adiacenti 6,5,2 > visito prima 6 non ha altri vertici adiacenti non ancora visitati > prendo 5 (adiacente a 0) > chi ha come adiacenti non visitati?

Guardo lista adiacenze > 4,3,0 (0 già visitato) > prendo 4 > 4 ha 5,3,2 (5 già visitato) prendo 3 > 3 ha 5,4 (5 e 4 già visitati) > torno indietro, prendo 2 > 4,1,0 - unico non visitato è 1, prendo 1 > ho finito

Ad ogni passo, avrò un valore *cnt* che verrà incrementato ogni volta, e che verrà inserito nel vettore *pre[]* in posizione corrispondente al vertice attuale, indicando così il tempo di scoperta di quel vertice.

Strutture dati per l'implementazione SEMPLICE

- Lista delle adiacenze di grafo non pesato
- Vettore *pre[]* > registra tempo di scoperta di un vertice
- *Cnt* > tiene tempi di scoperta (lo aumento ogni iterazione)

```

void GRAPHsimpleDfs(Graph G, int id) { <<< WRAPPER
    int v, cnt=0, *pre;
    pre = malloc(G->V * sizeof(int)); alloco vettore *pre (ovvero mark)
    if ((pre == NULL)) return; setto tutti i valori di pre a -1 (nessun vertice visitato ancora)
    for (v=0; v<G->V; v++) pre[v]=-1; visita a partire da id
        creo nodo finto come punto di partenza
    simpleDfsR(G, EDGEcreate(id,id), &cnt, pre); una volta che termina
        avrò alcuni nodi visitati, ma non tutti
    for (v=0; v < G->V; v++)
        if (pre[v]== -1) visita dei nodi non ancora visitati
            simpleDfsR(G, EDGEcreate(v,v), &cnt, pre); controllo tutti i vertici > se uno
                non è visitato, chiama funzione
                ricorsiva su di esso
    printf("discovery time labels \n");
    for (v=0; v < G->V; v++)
        printf("vertex %s : %d \n", STsearchByIndex(G->tab, v), pre[v]);
}

```

la seconda parte dopo la chiamata ricorsiva è fondamentale per quanto riguarda grafi orientati > poiché si potrebbero avere vertici non raggiungibili dalla direzione di quello da cui siamo partiti

Come funziona quindi?

- Creo in funzione wrapper contatore e vettore *pre[]*, ed inizializzo tutti i suoi valori a -1. Poi applico la funzione ricorsiva al primo nodo del grafo, applicandoci un nodo fittizio come punto di partenza.
- Una volta che ha finito la chiamata ricorsiva, non avrò per forza esplorato tutti i vertici >> ciclo su vettore *pre[]*, e se ho ancora valori negativi (cioè vertici non esplorati) chiamo funzione ricorsiva su di essi

arrivo su vertice w > esploro quindi con il ciclo for() tutti i vertici a lui adiacenti - come?
Uso lista di adiacenze >> è una lista, la attraverso finché non è NULL; se il vertice così trovato non è ancora visitato, allora visitalo

```

static void simpleDfsR(Graph G, Edge e, int *cnt, int *pre) {
    link t; int w = e.w;
    pre[w] = (*cnt)++;
    for (t = G->adj[w]; t != G->z; t = t->next) terminazione implicita
        if (pre[t->v] == -1) della ricorsione
            simpleDfsR(G, EDGEcreate(w, t->v), cnt, pre);
}

```

Struttura dati per l'implementazione ESTESA

Non teniamo conto solo del tempo di scoperta, ma anche del tempo di terminazione di visita di un determinato vertice.

- Segno quindi prima di chiamata ricorsiva su un vertice il suo tempo come prima *pre[i]* = *cnt++*; dopo la chiamata ricorsiva inserisco poi il tempo di visita nel vettore *post[i]* = *cnt++*
- Variabile **time** che incrementa ad ogni chiamata ricorsiva > serve per segnare *pre[]* e *post[]* (sostituisce variabile *cnt*)
- Vettore **st[]** per indicare nodo predecessore nella visita del grafo >> **albero della visita in profondità**

st	0	-1	-1	-1	0	-1
	0	1	2	3	4	5

vettore in cui 0 è nodo di partenza (ha sé stesso come padre) e 4 ha padre 0, quindi 0 -> 4 nell'attraversamento

```

void GRAPHextendedDfs(Graph G, int id) { wrapper
    int v, time=0, *pre, *post, *st;
    pre/post/st = malloc(G->V * sizeof(int));
    for (v=0;v<G->V;v++) {
        pre[v]=-1; post[v]=-1; st[v]=-1; } discesa ricorsiva
    extendedDfsR(G, EDGEcreate(id,id), &time, pre, post, st);
    for (v=0; v < G->V; v++)
        if (pre[v]==-1) secondo controllo con discesa ricorsiva su lista adiacenze
            extendedDfsR(G,EDGEcreate(v,v),&time,pre,post,st);
}

```

```

static void ExtendedDfsR(Graph G, Edge e, int *time, int *pre,
                           int *post, int *st) {
    link t;
    int w = e.w;
    st[e.w] = e.v, salva per il vertice che sto visitando, come padre il vertice da cui sono arrivato
    pre[w] = (*time)++; salva tempo di scoperta
    for (t = G->ladj[w]; t != G->z; t = t->next) terminazione implicita
        if (pre[t->v] == -1) della ricorsione
            ExtendedDfsR(G, EDGEcreate(w, t->v), time, pre, post, st);
    post[w] = (*time)++; dopo ciclo su lista adiacenze, segna tempo di fine scoperta
}

```

Visita in profondità – versione COMPLETA

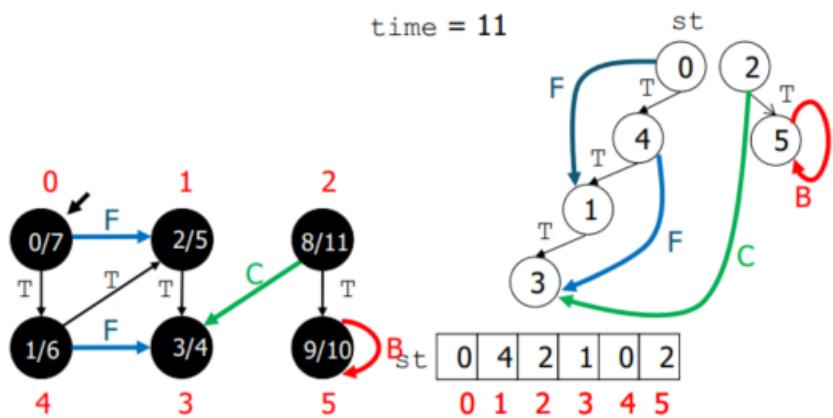
Si etichetta ogni arco, non solo *pre[]* e *post[]* >>

- *Grafi orientati* >> Tree, Backward, Forward, Cross
- *Grafi non orientate* >> Tree, Backward

Nei grafi orientati si definiscono archi:

- **T** > archi dell’albero della visita in profondità
- **B** > archi che connettono un vertice *w* ad un suo antenato *v* >> esistono se *pre[v] > post[w]* e se quando ho terminato di visitare *w*, *post[v]* è ancora -1
- **F** > connettono un vertice *w* ad un suo descendente *v* > *w->v* porta “in avanti”, esiste se *pre[w]* è maggiore di *pre[v]* ; questo quando scopro l’arco *w->v* (quindi se ho già visitato i due vertici e scopro un arco *w->v*, e *pre[v] > pre[w]* allora *w->v* è un arco forward)
- **C** > quando arco non è né T,B o F, se quando scopro arco *w->v* *pre[w] > pre[v]* allora è arco cross

Nei grafi non orientati abbiamo solo archi Tree e Backwards.

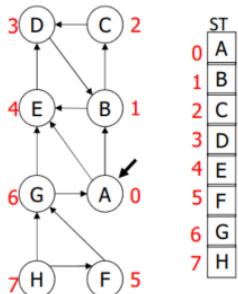


esempio arco Forward 0->1; nel corso della ricorsione, non visito subito 0->1 ma lo visito verso la fine...anche se scopro arco che porta a vertice c'è visitato, vedo che pre[1] > pre[0] e quindi l'arco che connette 0-> 1 è un arco di forward

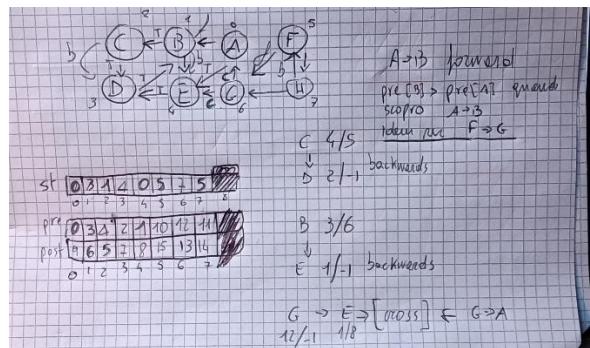
esercizio >>

Esempio

Lista archi	
A	B
B	C
C	D
A	E
F	G
F	H
D	B
D	E
E	B
B	E
G	E
H	G
H	F
F	G



Lista delle adiacenze	
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H



Implementazione – visita in profondità COMPLETA, tenendo conto di presenza di archi T-B-F-C

```

void dfsR(Graph G, Edge e, int *time,
          int *pre, int *post, int *st){
    link t;
    int v, w = e.w;
    Edge x;
    if (e.v != e.w) escludi arco fittizio
        printf("(%s, %s): T \n", STsearchByIndex(G->tab, e.v),
               STsearchByIndex(G->tab, e.w));
    st[e.w] = e.v; salva come padre del vertice attuale
    pre[w] = (*time); quello da cui arrivo
    for (t = G->adj[w]; t != G->z; t = t->next)
        if (pre[t->v] == -1) visita tutti vicini, se non ancora visitati
            dfsR(G, EDGEcreate(w, t->v), time, pre, post, st);
        else {
            v = t->v;
            x = EDGEcreate(w, v);

```

```

test per non considerare gli archi 2 volte
    if (pre[w] < pre[v])
        printf("(%s, %s): B\n", STsearchByIndex(G->tab, x.v),
               STsearchByIndex(G->tab, x.w));
    if (post[v] == -1)
        printf("(%s, %s): B\n", STsearchByIndex(G->tab, x.v),
               STsearchByIndex(G->tab, x.w));
    else
        if (pre[v] > pre[w])
            printf("(%s,%s):F\n", STsearchByIndex(G->tab, x.v),
                   STsearchByIndex(G->tab, x.w));
        else
            printf("(%s,%s):C\n", STsearchByIndex(G->tab, x.v),
                   STsearchByIndex(G->tab, x.w));
    }
    post[w] = (*time)++;
}

```

Complessità di visita in profondità

Lista adiacenze >> **O(V+E)**

Matrice adiacenze >> **O(V^2)**

Stiamo comunque considerando tutti i vertici adiacenti ad uno di partenza + tutte possibili liste di archi.



Altro esempio >> FLOOD FILL

Colorare un'intera area di pixel connessi con lo stesso colore (secchiello) – si usa una depth-first search partendo da un pixel sorgente, e si termina quando si incontra una frontiera

- Insieme di pixel rappresentati come lista di adiacenze / matrice (forse meglio matrice) > quando incontro frontiera, parto quindi poi da un pixel non ancora colorato (non visitato, -1)

2) Visita in ampiezza > BFS

La visita in ampiezza può essere considerata come un caso particolare del calcolo di cammino minimo

Dato un vertice S >

- Determina tutti i vertici raggiungibili da S > non visita tutti i vertici (*non ci sarà un ciclo for() dopo la chiamata ricorsiva partendo dal vertice S*)
- **Calcola anche la distanza minima da S di TUTTI i vertici da esso raggiungibili.**
 - *Per grafi non pesati, calcola il numero di archi per raggiungere ogni vertice raggiungibile*
- *In un certo senso, nella Depth-first research consideravamo i vertici uno alla volta – in quella in ampiezza, espandiamo tutta la frontiera, dal vertice corrente a quelli adiacenti.*

Struttura dati

- **Pre[]** registra tempo di visita di V
- **St[]** registra padre di V
- *Scoperta in parallelo avviene tramite utilizzo di una Queue >> simula esplorazione “in parallelo” anche se in realtà è un processo seriale*
- Contatore *time*

Algoritmo >

- Parti da arco fittizio in coda > ripeti il seguente passo finché coda non è vuota
 - Estrai da coda un arco e(v,w) >> se w non ancora visitato
 - Indica che st[e.w] = e.v, e marca e.w come scoperto (*pre[e.w] = time*)
 - Metti in coda tutti gli archi che portano a vertici non ancora scoperti, partendo da w

Implementazione visita in ampiezza >> BFS

Funzione wrapper >> inizializza vettore pre[] a -1, time = 0, st[] a -1 e richiama funzione bfs()

Funzione **BFS()** >> inizializza una coda, e ci inserisce il nodo iniziale. Poi inizia ciclo che continua finché coda non è vuota

```

void bfs(Graph G, Edge e, int *time, int *pre, int *st,
         int *dist) {
    int x;
    Q q = Qinit();
    Qput(q, e);
    dist[e.v]=-1;
    while (!Qempty(q)) finché coda non è vuota
        if (pre[(e = Qget(q)).w] == -1) {se il vertice del prossimo elemento
            pre[e.w] = (*time)++;
            st[e.w] = e.v;salvo parent
            dist[e.w] = dist[e.v]+1salvo distanza
            for (x = 0; x < G->V; x++) prendo tutti gli elementi adiacenti a quello corrente preso dalla coda
                if (G->adj[e.w][x] == 1) if (pre[x] == -1)non ancora visitato, metti in coda
                    Qput(q, EDGEcreate(e.w, x));
    }
}

```

matrice delle
adiacenze

ricordati: matrice di
adiacenze presenta 1
dove c'è un arco / vertice

Complessità – visita in ampiezza

Come per visita in profondità

- Matrice adiacenze >> $O(V^2)$
- Lista adiacenze >> $O(V+E)$

Tramite la visita in ampiezza, come accennato, si trova il cammino minimo da un vertice rispetto a tutti gli altri vertici > per ogni vertice abbiamo infatti segnato la distanza corrente, che è semplicemente la distanza dal punto di partenza che abbiamo inserito, dove la distanza era inizialmente 0.

Es. **Numeri di Erdos** >> algoritmo di visita in ampiezza usato per calcolare la *distanza* di ogni matematico (vertice) da Erdos – distanza intesa come persone tra quello corrente ed Erdos stesso.

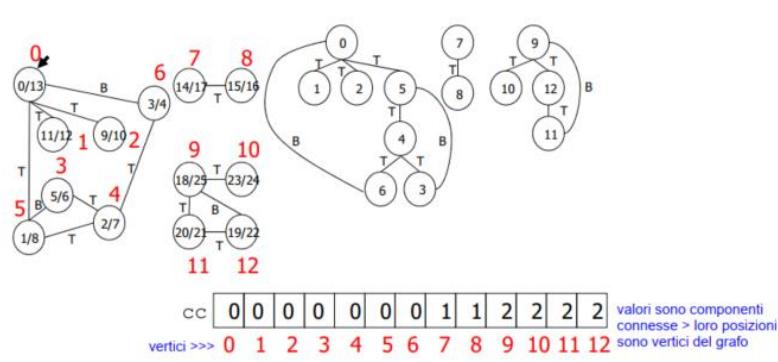
Applicazioni degli algoritmi di visita dei grafi

CICLI > un grafo è aciclico se non ci sono archi *backwards* > *non solo*

Altre proprietà dei grafi >>>

Componenti connesse > *in una visita in profondità, ogni albero della foresta è una componente connessa* >

- Funzione wrapper ha vettore **cc[v]** che identifica, per ogni posizione e quindi vertice, a quale componente connessa si ricollega quel vertice.



il vettore CC mostra, ad esempio valori 0 per tutti i vertici connessi al vertice 0.

Come calcolo le componenti connesse?

Funzione per calcolare componenti connesse > non è altro che una visita in profondità - in questo caso però devo trattare tutti i vertici, e quindi la funzione wrapper ciclerà su tutti i vertici, calcolando a quale componente connessa appartiene tramite una visita in profondità

```
void dfsRcc(Graph G, int v, int id, int *cc) { FUNZIONE RICORSIVA DI CALCOLO COMPONENTI CONNESSE
    link t;
    cc[v] = id; cc è vettore componenti connesse
    for (t = G->adj[v]; t != G->z; t = t->next)
        if (cc[t->v] == -1)
            dfsRcc(G, t->v, id, cc);
}
int GRAPHcc(Graph G) { WRAPPER
    int v, id = 0, *cc;
    cc = malloc(G->V * sizeof(int)); inizializzo vettore componenti connesse > vettore di lunghezza n
    for (v = 0; v < G->V; v++) cc[v] = -1;
    for (v = 0; v < G->V; v++)
        if (cc[v] == -1) dfsRcc(G, v, id++, cc);
    printf("Connected component(s) \n");
    for (v = 0; v < G->V; v++)
        printf("node %s in cc %d\n", STsearchByIndex(G->tab, v), cc[v]);
    return id;
}
```

Non faccio altro che fare una visita in profondità, ma invece di partire da un vertice preciso, parto da ciascun vertice, e ci passo il vettore **CC** che tiene traccia delle componenti connesse.

Connettività >>

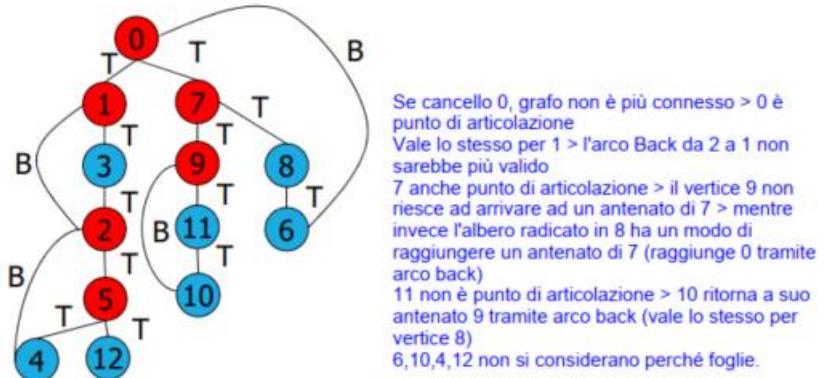
Grafo orientato e connesso >> se perdo proprietà di connessione in seguito a rimozione di:

- Arco >> **ponte**
- Nodo >> **punto di articolazione**
 - Rimuovendo il vertice si rimuovono anche gli archi che tendono su esso

1) Punto di articolazione

Dato grafo connesso e non orientato G, l'albero della visita in profondità **G**

- La radice di **G** è punto di articolazione se ha due figli (rimuovendolo sconnetto i figli)
- Ogni altro vertice V è punto di articolazione solo se ha un figlio tale che non vi è alcun arco che connette il figlio / un suo discendente ad un antenato di V.



2) Bridge

Archi back non sono mai bridge >> questo perché per prima definire un arco bridge devo essere già passato per i due vertici, e quindi esiste già un percorso alternativo.

Un arco TREE (v,w) è un ponte solo se non esistono archi BACK che riconnettono un discendente di w ad un antenato di v

Come verifico esistenza di ponti?

- Visita in profondità calcolando le componenti connesse, tolgo un arco alla volta >> se ho sempre una sola componente连通, grafo è connesso. Sennò no.

Directed Acyclic Graph (DAG) – grafi connessi aciclici

Un albero ricordiamo che è un grafo, non orientato, connesso e aciclico.

Modelli DAG usati per scheduling di compiti >>> un compito prima di essere svolto richiede lo svolgimento di un compito precedente >>> allo stesso modo l'esplorazione di un albero, prima di giungere ad un dato nodo devo passare per tutti quelli prima. Si ha quindi un ordine di priorità (tipico degli alberi, bst...)

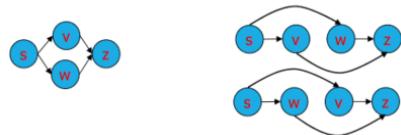
2 tipologie di nodi nei grafi connessi aciclici >

- **Nodi sorgente / source** > non hanno nodi in entrata
- **Nodi pozzo /sink** > non hanno nodi in uscita

Ordinamenti

1) Topologico > ordino i vertici secondo una linea orizzontale – se esiste un arco $u \rightarrow v$, u è a sx di v , e gli archi vanno da sx \rightarrow dx (ho tutti vertici disposti su linea orizzontale che vanno da sx \rightarrow dx)

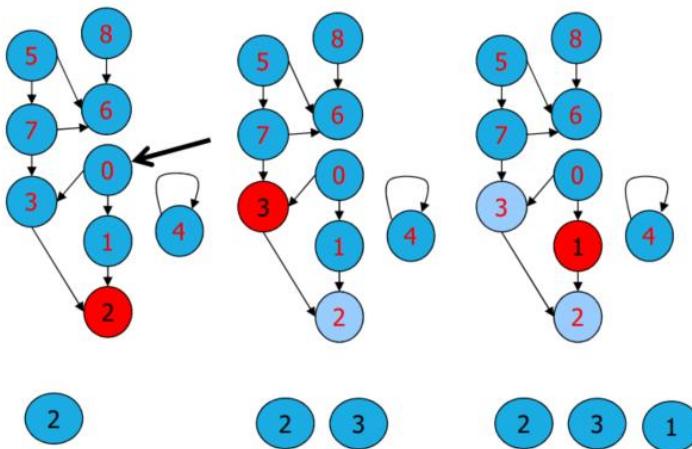
- Ogni DAG ha ALMENO un ordinamento topologico > a meno che esista un cammino Hamiltoniano orientato, in quel caso è unico.



Due diversi ordinamenti topologici dello stesso grafo

2) Topologico inverso > opposto di ordinamento topologico normale > da dx \rightarrow sx

- Man mano che esploro i diversi vertici, una volta che non posso più fare altro che tornare indietro, li inserisco in vettore ordine topologico.



i vertici rossi sono i vertici da cui non posso più proseguire con la funzione ricorsiva

DAG > STRUTTURA DATI

DAG trattato come un ADT di classe per rappresentare ordinamenti topologici e topologici inversi

- Uso liste adiacenze + vettori `pref[]`, `ts[]` -> ogni posizione è il tempo, e contiene il vertice scoperto a quel tempo
- Variabile `time` che aumenta solo quando ho terminato la visita di un vertice

ORDINAMENTO TOPOLOGICO INVERSO

```
void TSdfsR(Dag D, int v, int *ts, int *pre, int *time) {
    link t; pre[v] = 0;
    for (t = D->1adj[v]; t != D->z; t = t->next)
        if (pre[t->v] == -1) SE VERTICE NON ANCORA VISITATO, LANCIO VISITA IN PROFONDITA'
            TSdfsR(D, t->v, ts, pre, time);
        ts[(*time)++] = v; UNA VOLTA FINITO, INSERISCI VERTICE IN VETTORE (vertice è stato visitato)
    } IN POSIZIONE TEMPO++
}

void DAGrts(Dag D) {
    int v, time = 0, *pre, *ts; ALLOCÒ VETTORE VERTICI E TEMPI
    /* allocazione di pre e ts */
    for (v=0; v < D->V; v++) { pre[v] = -1; ts[v] = -1; }
    for (v=0; v < D->V; v++) SCANDISCO VERTICI E, SE NON ANCORA VISITATO, CHIAMA FUNZIONE RICORSIVA
        if (pre[v]== -1) TSdfsR(D, v, ts, pre, &time);
    printf("DAG nodes in reverse topological order \n");
    for (v=0; v < D->V; v++)
        printf("%s ", STsearchByIndex(D->tab, ts[v]));
    printf("\n");
}
```

praticamente visita tutti i vertici, e una volta che in quel vertice non può più andare da nessuna parte, lo inserisce nel vettore `ts[time]` e aumenta `time`

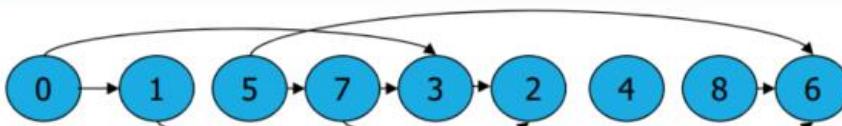
Il vettore `ts[]` rappresenterà quindi l'ordine topologico inverso.

ORDINAMENTO TOPOLOGICO

ordinamento topologico: con il DAG rappresentato da una matrice delle adiacenze, basta invertire i riferimenti riga-colonna (considerando gli archi incidenti):

```
void TSdfsR(Dag D, int v, int *ts, int *pre, int *time) {
    int w;
    pre[v] = 0;
    for (w = 0; w < D->V; w++)
        if (D->madj[w][v] != 0)
            if (pre[w] == -1)
                TSdfsR(D, w, ts, pre, time);
    ts[(*time)++] = v;
}
```

lavoro quindi con una matrice di incidenze e non di adiacenze



Componenti fortemente connesse

È un sotto-grafo massimale per cui vale la condizione di mutua raggiungibilità

Algoritmo di Kosaraju >

- Esegue visita in profondità su grafo trasposto (inverte direzione del grafo e calcola i tempi di scoperta)
- Torna poi sul grafo normale, e va avanti, partendo dal vertice che ha terminato con tempo massimo nell'esplorazione trasposta.
- Gli alberi dell'ultima visita in profondità sono le componenti fortemente connesse.

Perché calcolare le componenti fortemente connesse?

- ridurre un grafo da un insieme di vertici ad un insieme di componenti fortemente connesse >> non devo più rappresentare i singoli vertici, ma una sola componente per ogni insieme di vertici
 - versione condensata del mio grafo >> **KERNEL DAG**
 - Come faccio a sapere che è un DAG? Come faccio a dire che è aciclico?
 - **Se fosse ciclico** > ci sarebbe un ciclo che connette alcuni dei suoi vertici, e se ci fosse, sarebbero inclusi nella stessa classe – ma se non sono stati inclusi, allora *Componente fortemente connessa non sarebbe massimale* >> contraddico tesi iniziale, assurdo

Il calcolo delle componenti fortemente connesse (SGG) è quindi utilizzato per ridurre la complessità di operazioni, ad esempio, su macchine a stato finito.

- Usare una singola componente per rappresentare un grafo è infatti un grande risparmio in complessità e memoria

Implementazione – calcolo componenti fortemente connesse

Struttura dati >

- Vettore **sccG[]** memorizza componente fortemente connessa per ogni posizione (vertice) -> *sfrutto la corrispondenza posizione-vertice per usarlo anche come vettore mark[]*
- Vettore **sccR[]** memorizza vertici visitati di grafo trasposto
- **Time0** contatore del tempo, avanza solo a terminazione visita di un vertice
- Vettore **postR[]** > tempi di visita di tutti i vertici nel grafo trasposto > se lo percorro al contrario ho tempi in ordine decrescente
- Contatore delle componenti fortemente connesse **Time1**

1) Calcolo grafo trasposto

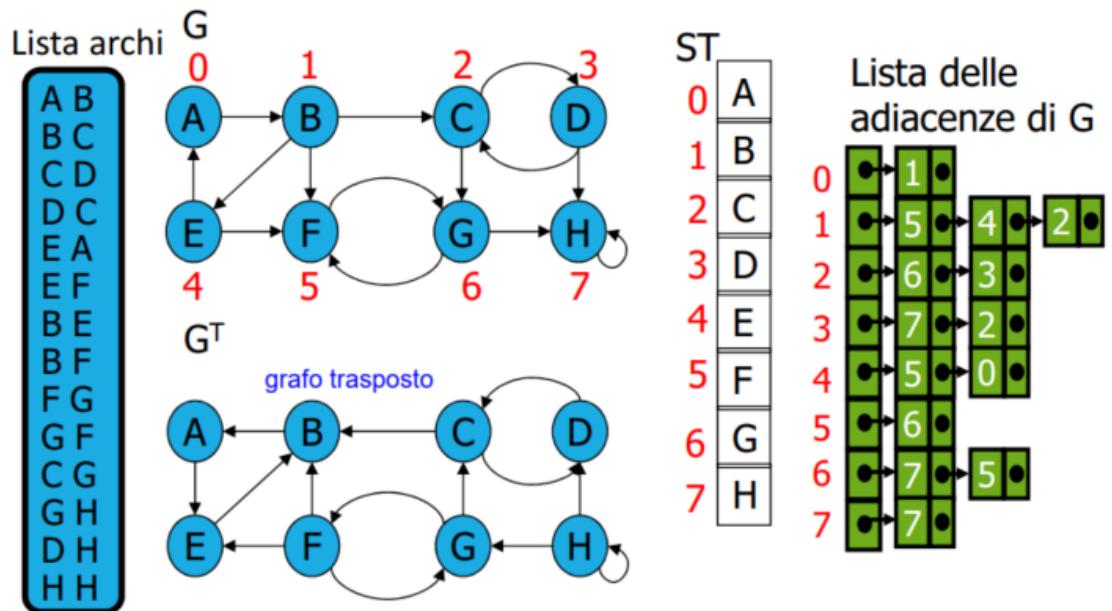
Grafo trasposto è grafo con direzione di ogni arco invertita

```

creo grafo trasposto
Graphreverse(Graph G) {
    int v;
    link t;
    Graph R = GRAPHinit(G->V);
    for (v=0; v < G->V; v++)
        for (t= G->ladj[v]; t != G->z; t = t->next)
            GRAPHinsertE(R, t->v, v);
    return R;
}

```

Esempio



postR

1	4	0	3	2	6	5	7
0	1	2	3	4	5	6	7

visita del grafo trasposto

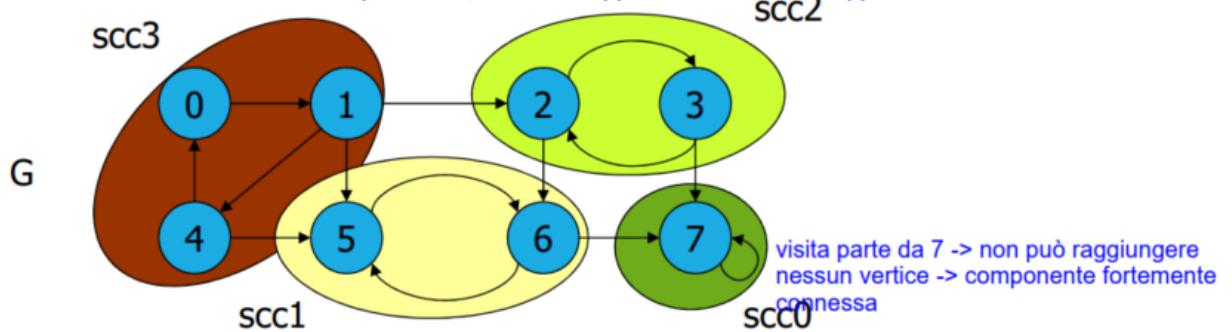
Visito quindi poi il grafo originale, partendo da postR[] all'ultima posizione >> calcolo le componenti fortemente connesse

sccG

0	1	2	3	4	5	6	7
3	3	2	2	3	1	1	0

componenti fortemente connesse, partendo dall'ultimo elemento di postR[] che è 7, e che ha come SGG quindi 0

Le componenti fortemente connesse saranno quindi 0,1,2 e 3 -> le raggrupperò poi quindi in classi di equivalenza, invece di rappresentare 8 vertici rappresenterò 4 classi di SCC



FUNZIONE PER DETERMINARE COMPONENTI FORTEMENTE CONNESSE

```
void SCCdfsR(Graph G,int w,int *scc,int *time0,int time1,int *post) {
    link t;
    scc[w] = time1;
    for (t = G->ladj[w]; t != G->z; t = t->next)
        if (scc[t->v] == -1)
            SCCdfsR(G, t->v, scc, time0, time1, post);
    post[(*time0)++]= w; algoritmo identico al precedente (che però opera sul grafo trasposto)
}
```

funzione di visita del grafo trasposto dato un vertice e il vettore postR

Questa funzione verrà usata sia con il grafo originale G che con il grafo trasposto R – ha lo scopo di determinare le componenti fortemente connesse.

ALGORITMO DI DETERMINAZIONE COMPONENTI FORTEMENTE CONNESSE

```
int GRAPHscC(Graph G) {
    int v, time0 = 0, time1 = 0, *sccG, *sccR, *postG, *postR;
    Graph R = GRAPHreverse(G); costruisco grafo trasposto

    sccG = malloc(G->V * sizeof(int));
    sccR = malloc(G->V * sizeof(int)); inizializzo 4 vettori
    postG = malloc(G->V * sizeof(int));
    postR = malloc(G->V * sizeof(int));
```

```

for (v=0; v < G->V; v++) {
    sccG[v]=-1; sccR[v]=-1; postG[v]=-1; postR[v]=-1;
}
for (v=0; v < G->V; v++) ciclo sui vertici, partendo da quello non ancora scoperto -> lancio algoritmo
    if (sccR[v] == -1) di visita in profondità (su grafo trasposto)
        SCCdfsR(R, v, sccR, &time0, time1, postR);
time0 = 0; time1 = 0; una volta finito, resettiamo i tempi >> Poi parto con visita su grafo originale partendo da postR[]
for (v = G->V-1; v >= 0; v--) parto con un'altra visita > si basa su tempi di elab. decrescenti
    if (sccG[postR[v]]== -1){ controllo se sccG a tempo postR con quel vertice non ancora visitato, e lo visito se -1
        SCCdfsR(G,postR[v], sccG, &time0, time1, postG);
        time1++;
    }
printf("strongly connected components \n");
for (v = 0; v < G->V; v++)
    printf("node %s in scc %d\n", STsearchByIndex(G->tab,v), sccG[v]);
return time1;
}

```

Alberi ricoprenti minimi (MST)

Minimum-weight spanning tree

Ricorda: un albero ricoprente ha sempre $V-1$ archi

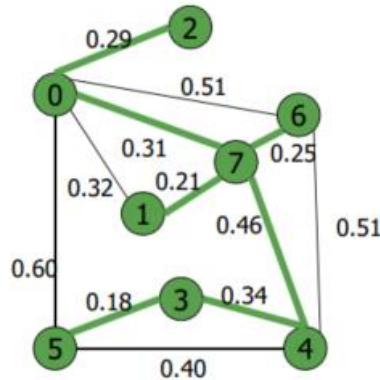
Se di più, ho cicli > se di meno, ho foresta di alberi

Dato un grafo $G = (V, E)$ non orientato e con pesi reali, connesso, *estrarre un albero ricoprente minimo (MST) > cosa vuol dire?*

- Trovare un sottografo $G'=(V,A)$ dove A è contenuto in E , G' è aciclico
- **G' minimizza la somma dei pesi degli archi che ho considerato** – non minimizzo il peso in sé (*prende solo archi con pesi minimi*)

G' è quindi un albero che copre TUTTI i vertici, e minimizza la somma dei pesi degli archi considerati.

- **L'MST G' è unico solo se tutti i pesi sono distinti.**



Rappresentazione

ADT grafo non orientato e pesato

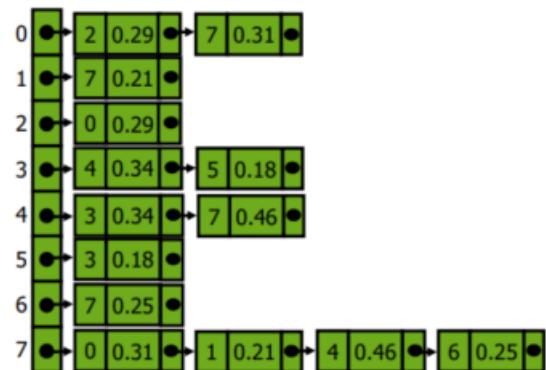
- Lista di adiacenze / matrice di adiacenze
- Si usa un valore sentinella per indicare l'assenza di un arco (peso inesistente)
 - Il valore può essere un valore molto alto (*quindi non sarà mai scelto come valore conveniente da attraversare rispetto agli altri*)
 - 0 se non sono ammessi archi a peso 0
 - -1 se non sono ammessi archi a peso negativo

Rappresentazioni utilizzate

- 1) Algoritmo Kruskal >** elenco di archi, memorizzato in un vettore di archi
 $mst[maxE]$

0-2 0.29
 4-3 0.34
 5-3 0.18
 7-4 0.46
 7-0 0.31
 7-6 0.25
 7-1 0.21

- 2) Grafo come lista di adiacenze >** soluzione
 molto adottata finora per altri grafi >
 ogni dato è una struct con indice + valore



3) Vettore st padri wt pesi > gli indici del vettore sono i vertici (soluzione usata da *algoritmo di Prim*)

Vettore st dei padri e wt dei pesi								
	0	1	2	3	4	5	6	7
st	0	7	0	4	7	3	7	0
wt	0	.21	.29	.34	.46	.18	.25	.31

Diversi approcci al calcolo di un albero ricoprente minimo

Approccio completo >

Un albero ricoprente avrà sempre $V-1$ archi.

Approccio *brute-force*, calcolo tutti i diversi modi di raggruppare i $V-1$ archi scelti dagli E archi totali.

- Non conta l'ordine, userò *combinazioni*
- *Soluzione è accettabile se non ci sono cicli*
- *Si deve scegliere la soluzione migliore*
- **Complessità $O(N^N)$**

Se quindi dato un grafo, devo trovare un MST, mi basta usare le combinazioni (no powerset) > k dimensione insieme finale sarà $V-1$ ($G \rightarrow V-1$)

Approccio Greedy

Generalmente, l'approccio greedy prende ogni volta la scelta *localmente ottima* – questo però non garantisce sempre una scelta globalmente ottima.

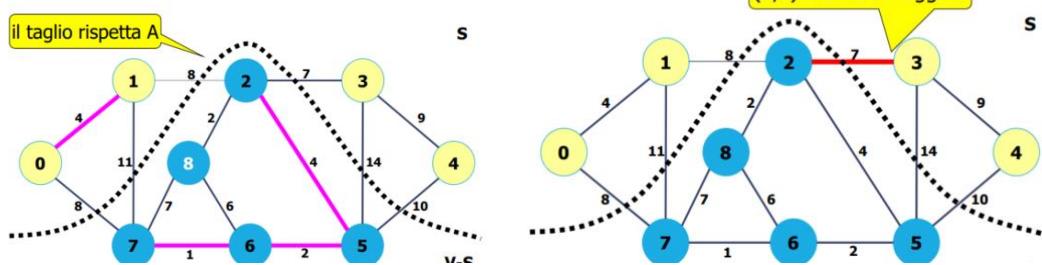
Per gli MST, si parte da un approccio *incrementale, generico e greedy* dove si prende la scelta *localmente ottima*, ed è dimostrabile che questa porterà ad una scelta globalmente ottima.

Inizialmente ho insieme vuoto, che è un sottoinsieme per definizione degli archi di un albero ricoprente minimo.

- **Ad ogni passo**, si aggiunge un arco *sicuro*
- **Invarianza** > un arco (u,v) è *sicuro* solo se, aggiunto ad un sottoinsieme di un albero ricoprente minimo, produce ancora un sottoinsieme di un albero ricoprente minimo.

Arco leggero > arco con *peso minimo* tra gli archi che attraversano un taglio.

Posto che A sia $A = \{(0,1), (2,5), (5,6), (6,7)\}$



Come definisco un arco *sicuro*?

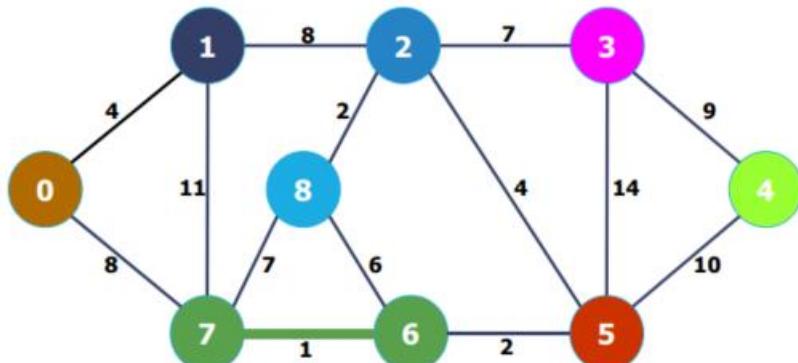
- 1) *Teorema* > Se $G = (V, E)$ grafo non orientato, connesso e pesato
 - un arco leggero rispetto ad un Taglio che rispetta A è considerato **sicuro per A** (ha valore minimo)
- 2) *Corollario* > Se $G = (V, E)$ grafo non orientato, connesso e pesato
 - dato un albero C nella foresta $G = (V, A)$ e un arco leggero (u, v) che connette G rispetto ad un altro albero sempre in $G = (V, A)$
 - allora (u, v) è sicuro per $A \rightarrow (u, v)$ è un arco leggero che connette due foreste
 - prendo quindi quegli archi sicuri e quindi minimi che permettono di connettere una foresta a nuovi vertici non ancora raggiunti da quella foresta

Algoritmo di Kruskal

Si basa sull'utilizzo del corollario:

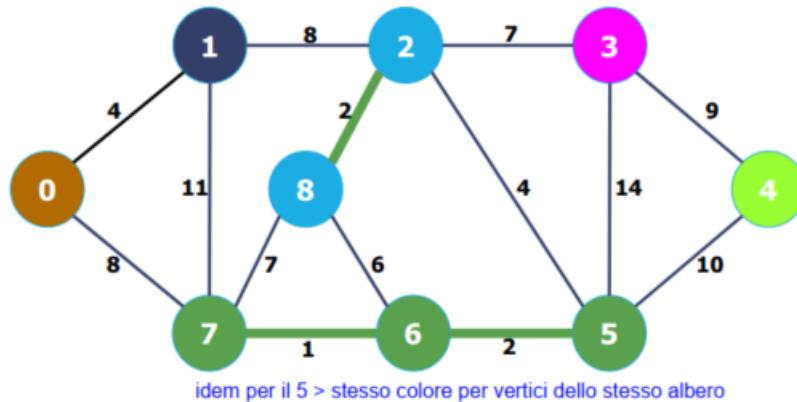
- all'inizio ho una foresta di alberi, che sono tutti i vertici a mia disposizione.
- Si ordinano gli archi per pesi crescenti, e prendo iterando l'arco sicuro (peso minimo) > **l'arco sicuro connetterà due alberi tra loro, generandone uno unico** (unisco albero precedente a nuovo vertice NON VISITATO)

Esempio > sono sul vertice 7, prendo l'arco con peso minimo > porta ad un vertice non ancora visitato? Se sì, allora collego i due alberi (in questo caso vertici) e continuo su quel vertice

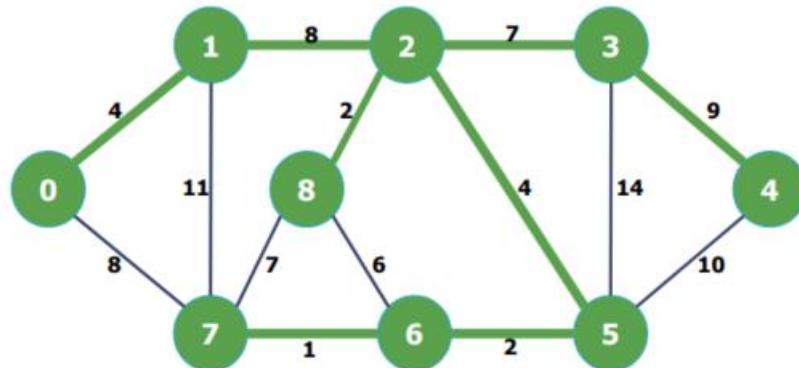


l'arco 7-6 unisce due alberi diversi ? sì, li unisco in un unico albero

Sono sul vertice 6 > ripeto stesso procedimento, arco sicuro è 6->5 di valore 2 > 5 non è ancora visitato, lo prendo e ripeto per vertice 5



Una serie di scelte localmente ottime mi porta così ad una scelta globalmente ottima > ho trovato un albero ricoprente minimo



IMPLEMENTAZIONE ALGORITMO DI KRUSKAL

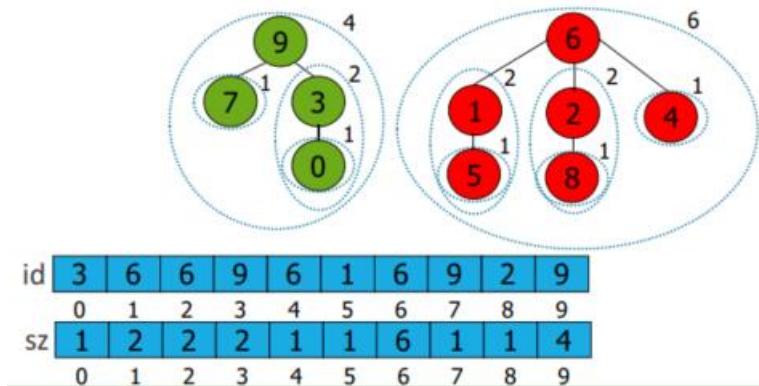
ADT UNION FIND

Disjoint-set data / merge-find set

Struttura dati usata per rappresentare diversi sottoinsiemi disgiunti tra loro – l'obiettivo è utilizzare le funzioni

- *UFunion* > unire due sottoinsiemi
- *UFfind* > verifica se 2 elementi appartengono o meno allo stesso sottoinsieme

- Vettore *id* di N-1 elementi > ad ogni posizione corrisponde un elemento (corrispondenza posizione – indice) >> il valore contenuto è l'indice del valore che lo rappresenta (*simile vettore st[] per grafi*)
 - Inizialmente ogni elemento rappresenta sé stesso
 - Quando poi avrà diversi elementi rappresentati da un sottoinsieme, *l'elemento che rappresenta il sottoinsieme è quell'elemento dove indice == elemento*
- Vettore *sz* contiene cardinalità del sottoinsieme a cui appartiene ogni elemento.



letteralmente uguale ad un vettore st – il vettore sz rappresenta il numero totale di elementi per quel sottoinsieme

Implementazione dell'ADT di I classe UNION FIND

```
UF.h
void UFinit(int N);
int UFFind(int p, int q);
void UFunion(int p, int q);
```

- **UFinit()** >> alloca id e sz – setta tutti i valori di sz a 1 (ogni elemento rappresenta minimo sé stesso)
- **quickUnion()** > $O(1)$, elemento punta a chi lo rappresenta – accesso diretto su vettore
- **find()** > $O(\log N)$ – unisco insieme con cardinalità minore con quello a cardinalità maggiore, genera un cammino di lunghezza logaritmica

```
static int find(int x) {
    int i = x;
    while (i != id[i]) i = id[i]; continua finché non trovi l'elemento che rappresenta il sottoinsieme
    return i;
} UFFind trova il punto di incontro dei due sottoinsiemi
int UFFind(int p, int q) { return(find(p) == find(q)); } unisco il sottoinsieme più piccolo al più grande - come?
void UFunion(int p, int q) {
    int i = find(p), j = find(q);
    if (i == j) return; se appartengono allo stesso insieme -> ritorna, non devi unire
    if (sz[i] < sz[j]) {
        id[i] = j; sz[j] += sz[i];
    } trovo quale dei due sottoinsiemi ha meno elementi (comparo sz[i] e sz[j]) > quello più piccolo, id[9] (elemento che rappresenta quel sottoinsieme) diventa elemento che rappresenta sottoinsieme più grande
    else {
        id[j] = i; sz[i] += sz[j];
    }
}
```

La funzione **find()** trova l'elemento che rappresenta un sottoinsieme, dato un vertice.

La funzione **UFind()** trova il punto d'incontro di due sottoinsiemi.

La funzione **UFunion()** prende, dati due sottoinsiemi, i loro rappresentanti. Se sono uguali, ritorna. se non lo sono, quello con la quantità di elementi più piccola viene integrato dal più grande (rappresentante con meno elementi diventa rappresentato da quello che ne ha di più, ovvero dall' $sz[]$ più grande)

Calcolo dell'MST in sé

```

void GRAPHmstK(Graph G) {
    int i, k, weight = 0; inizializzo vettore risultato > dimensione è V-1 obbligatoriamente
    Edge *mst = malloc((G->V-1) * sizeof(Edge));
    Edge *a = malloc(G->E * sizeof(Edge));

    k = mstE(G, mst, a);

    printf("\nEdges in the MST: \n");
    for (i=0; i < k; i++) {
        printf("%s - %s\n", STsearchByIndex(G->tab, mst[i].v),
               STsearchByIndex(G->tab, mst[i].w));
        weight += mst[i].wt;
    }
    printf("minimum weight: %d\n", weight);
}

```

funzione wrapper, calcola il vettore di archi MST e lo stampa

```

int mstE(Graph G, Edge *mst, Edge *a) {
    int i, k;

    GRAPHedges(G, a); prende tutti gli archi
    sort(a, 0, G->E-1); li ordina dal più piccolo
    UFinit(G->V); crea l'union find ADT
    for (i=0, k=0; i < G->E && k < G->V-1; i++ ) per tutti gli archi
        if (!UFFind(a[i].v, a[i].w)) {
            UFunion(a[i].v, a[i].w); se i due vertici non
            appartengono allo stesso
            sottoinsieme, uniscili e
            salva in MST l'arco corrente
            >> fa parte del risultato
            mst[k++]=a[i];
        }
    return k;
}

```

ù

Complessità > UF

$T(n) = O(E \cdot \lg E) = O(E \cdot \lg V)$ ma $E = V^2$ in grafo completo $\gg O(\log(V^2)) > O(2\log V) > O(\log V)$

Algoritmo di PRIMM

Offre una soluzione *brute-force*, si basa sul teorema per stabilire arco sicuro

- insieme della soluzione inizialmente vuoto, poi prende primo vertice
- Partendo da un vertice, si ha un ciclo annidato su tutti gli archi vicini, scegliendo il minimo > lo aggiungo alla soluzione e mi sposto su quel vertice, e aggiorno S

- È una versione semplice *ma non efficiente a causa del ciclo annidato*

Esiste però un algoritmo migliorato rispetto a quello originale di Prim ->

- *ad ogni passo aggiungo un vertice v a S*
- *ci interessa la distanza minima da ogni vertice ANCORA IN V-S ai vertici già in S*
- *quando si aggiunge un vertice ad S, un vertice in V-S vicino ad un vertice in S può già avvicinarsi*
- non serve memorizzare la distanza tra quel vertice e tutti i vertici in S > prendo quella minima e verifico se la sua aggiunta ad S la diminuisce, e se sì allora la aggiorno.

Algoritmo di PRIMM – implementazione

Strutture dati

- grafo come matrice di adiacenze
- vettore *st*
- vettore *fringe* > dato un vertice indicato dalla posizione, il suo valore è il vertice a cui è più vicino
- vettore *wt* > dimensione $V+1$ che registra
 - per elementi di S > peso di arco padre
 - per elementi di $V-S$ > peso arco verso vertice più vicino
- variabile *min* > vertice in $V-S$ più vicino ai vertici di S

Algoritmo

Inizialmente setto tutti i valori di *wt* a +inf (non conosco ancora distanze)

- ciclo esterno sui vertici prendendo ad ogni passo quello a minima distanza > lo aggiungo ad S
 - inizialmente $m = 0$, primo vertice > salvo in *st[min]* il valore in *fr[min]* (salvo il padre di quell'elemento)
- ciclo interno sui vertici w non ancora in S (ovvero dove *st[w] == -1*)
 - se l'arco dal vertice attuale (*min*) a $w \rightarrow$ cioè $\text{mat}[v][w] < \text{wt}[w]$ allora $\text{wt}[w] = \text{mat}[v][w]$ e **indico che il vertice più vicino a v è $w \rightarrow \text{fringe}[w] = v$**
- se w è diventato vertice più vicino a v (min corrente), allora $\text{min} = w$ e ripeti procedimento

Ciclo for() partendo da vertice = 0, e finché non ho finito i vertici. (finisco quando il vertice attuale è uguale al numero di vertici)

- ciclo poi sui vertici non ancora presi, cioè dove il vertice w è tale che *st[w] == -1*.
- All'inizio il vettore *wt* contiene tutti valori +inf. Questo perché non so ancora quale sia la distanza da un vertice all'altro. Controllo quindi se nella matrice di adiacenze, dato il vertice attuale e quello che sto controllando, esiste un arco che li collega più piccolo di infinito.
- Se esiste, aggiorno $\text{wt}[w] = \text{mat}[v][w]$ e segno che il vertice w è il più vicino al vertice $v \rightarrow \text{fringe}[w] = v$
- Se $\text{wt}[w] < \text{wt}[min]$, ovvero se w è diventato il vertice più vicino a S , allora $\text{min} = w$

```

void GRAPHmstP(Graph G) {
    int v, *st, *wt, weight = 0;
    st = malloc(G->V*sizeof(int));
    wt = malloc((G->V+1)*sizeof(int));

    mstV(G, st, wt);

    printf("\nEdges in the MST: \n");
    for (v=0; v < G->V; v++) {
        if (st[v] != v) {
            printf("(%s-%s)\n", STsearchByIndex(G->tab, st[v]),
                   STsearchByIndex(G->tab, v));
            weight += wt[v];
        }
    }
    printf("\nminimum weight: %d\n", weight);
}

```

funzione wrapper

```

void mstV(Graph G, int *st, int *wt) {
    int v, w, min, *fr = malloc(G->V*sizeof(int));
    for (v=0; v < G->V; v++) {
        st[v] = -1; fr[v] = v; wt[v] = maxWT;
    } inizializzazione st[], fr[], wt[]

    st[0] = 0; wt[0] = 0; wt[G->V] = maxWT;
    for (min = 0; min != G->V; ) {prendo vertice iniziale, continuo finché non sono
        v = min; st[min] = fr[min]; alla fine considero tutti altri vertici >
        for (w = 0, min = G->V; w < G->V; w++) inizialmente min = G->V ma poi
            if (st[w] == -1) {se non ancora visitato > cambia
                if (G->madj[v][w] < wt[w]) se esiste arco e suo valore è minore di
                    wt[w] = G->madj[v][w]; fr[w] = v;
                } salvo come arco minimo quello attuale, e salvo come vertice minimo min
                if (wt[w] < wt[min]) min = w; aggiorno min > se il peso è minore per quel vertice,
            } allora min = quel vertice
    }
}

```

Naturalmente, se un arco non esiste tra due vertici, il suo valore sarà +inf. Che sarà quindi uguale a $wt[w]$ per quel vertice, e quindi non andrà oltre e controllerà gli altri vertici.

SPIEGAZIONE PER QUANDO NON LO RICORDI O NON LO CAPISCI>>

Il primo ciclo for ha lo scopo di assegnare inizialmente il primo nodo, ovvero 0. Continua finché non ho finito i vertici.

L'assegnazione fatta dentro il ciclo $v = min$, $st[min] = fr[min]$ ha lo scopo di segnare i risultati del secondo ciclo for.

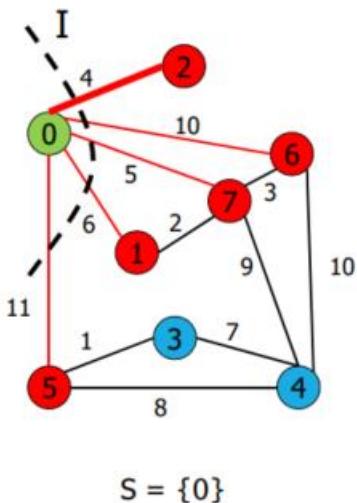
Il secondo ciclo for() >> controlla tutti i vertici del grafo, e assegna $min = G->V$, che è un valore il cui peso è infinito.

Controlla prima di tutto che ogni vertice w non sia stato ancora visitato > se è già stato visitato, va oltre.

- **se non è stato visitato** > controlla che il valore dal vertice attuale V e il vertice in considerazione nel secondo ciclo for w abbiano un arco esistente – come ? controlla matrice di adiacenze in $[v][w] < wt[w]$ (inizialmente $wt[]$ è tutto + infinito > se non esiste, valore è infinito, e quindi non va oltre e aumenta $w++$, controlla vertice dopo)
 - **se $madj[v][w]$ ha un valore valido** > ed è più piccolo di $wt[w]$, allora salva il nuovo $wt[w]$ come $madj[v][w]$ e salva il nuovo vertice di frontiera per w come $V \rightarrow fr[w] = V$
- una volta finito, se $wt[w] < wt[min]$, aggiorna il nuovo $min = w$ >>> inizialmente sarà sempre vero, poiché $wt[G->V]$ è infinito.

- Questa fase è importantissima >> quando arrivo ad un nodo che non ha più vicini se non un nodo visitato, il secondo ciclo for() si ripeterà inizialmente, anche se V attuale non ha vicini. Min verrà quindi aggiornato di nuovo al vertice precedente, e V tornerà al valore del vertice precedente dopo la conclusione del ciclo.
- Si ripeterà quindi il ciclo di controllo dei vicini per il vertice precedente, stavolta però escludendo i vertici già presi (e quindi quello che prima non aveva più vicini), e procedendo quindi con l'esplorazione del grafo.

Es.



	0	1	2	3	4	5	6	7	8
st	0	-1	-1	-1	-1	-1	-1	-1	
wt	0	6	4	∞	∞	11	10	5	∞
fr	0	0	0	3	4	0	0	0	

- min = 0, aggiorno st[0] e wt[0]
- fringe contiene 1, 2, 5, 6, 7
- aggiorno wt e fr in base agli archi che attraversano il taglio
- il vertice 2 è quello più vicino a S, in quanto 0-2 è l'arco a peso minimo che attraversa il taglio I
- min = 2

$$V-S = \{1, 2, 3, 4, 5, 6, 7\}$$

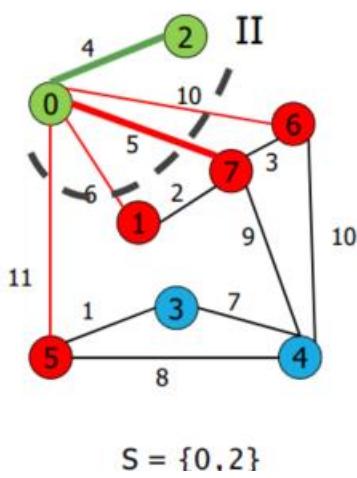
Analizzo tutti i vicini di 0 > (in realtà calcolo TUTTI i vertici con l'algoritmo, ma quando controlla se $G \rightarrow \text{madj}[v][w] < \text{wt}[w]$, se non esiste l'arco troverà infinito come valore di $\text{madj}[v][w]$, e quindi non andrà oltre)

Dopo aver terminato un'esecuzione, v viene settato a min

Quindi min = 2 -> v = 2, st[2] = fr[2] e procedo

IMPORTANTE > quando avrò v = 2, il ciclo for() che controlla ogni vertice assegnando w = 0 e min = G->V, riassegnerà min = 0, e quindi farà un giro a vuoto poiché nessun vertice oltre 0 avrà distanza da sé stesso minore.

Da qui in poi ripartirà un altro ciclo che avrà di nuovo come centro il vertice 0, ma stavolta escluderà il vertice 2, che è stato già visitato, continuando così l'esplorazione del grafo.



	0	1	2	3	4	5	6	7	8
st	0	-1	0	-1	-1	-1	-1	-1	
wt	0	6	4	∞	∞	11	10	5	∞
fr	0	0	0	3	4	0	0	0	

- aggiungo 2 alla soluzione e aggiorno st[2]
- fringe contiene 1, 5, 6, 7
- aggiorno wt e fr in base agli archi che attraversano il taglio
- il vertice 7 è quello più vicino a S, in quanto 0-7 è l'arco a peso minimo che attraversa il taglio II
- min = 7

$$V-S = \{1, 3, 4, 5, 6, 7\}$$

Complessità – algoritmo di Prim

Per grafi densi, $T(N) = O(V^2)$

Per grafi sparsi, si può migliorare la complessità usando una coda a priorità $> O(E \log V)$

*****CAMMINI MINIMI*****

Grafi orientati, pesati

Per definizione, dato un cammino $w(p)$ come la somma di tutti i pesi degli archi che lo compongono, un cammino minimo $S \rightarrow$ è il valore minimo di $w(p)$, purché S e V siano connessi

- Il cammino minimo tra due vertici sconnessi è infinito.

Algoritmi su cammini minimi

Da una sorgente singola >>

- **Dijkstra** > dato un sorgente, calcola distanza minima da tutti i vertici
 - Archi negativi > Non garantisce soluzione ottima
 - Se archi negativi creano ciclo < 0, risultato senza senso
- **Bellford**
 - Archi negativi > Garantisce soluzione ottima
 - Se archi negativi creano ciclo < 0, rileva ciclo negativo ma fornisce risultato insensato.

Con destinazione singola

Tra coppie di vertici

Tra tutte le coppie di vertici > teoricamente basterebbe iterare per tutti i vertici l'algoritmo che calcola i cammini minimi verso tutti gli altri > esiste però una soluzione che riduce la complessità rispetto ad una semplice e *bruta* iterazione.

RICORDA: I cammini minimi sono cammini semplici, con massimo $V-1$ archi.

Non sono ammessi cicli >

- Non possono contenere cicli a peso negativo perché perderebbero senso
- Non possono contenere cicli, poiché se svolgo un ciclo ho aumentato il peso totale del cammino, e non è più semplice >> assurdo

Cammini minimi – Approcci

1) Brute force

Enumero tutti i sottoinsiemi di archi con dimensione k compreso tra 0 e $V-1$ – utilizzo un powerset con disposizioni semplici (*ordine conta*), e scelgo quello con peso minimo

Complessità > $O(N^N)$ esponenziale

2) Sottostruttura ottima di un cammino minimo > programmazione dinamica, approccio greedy

Un sottocammino minimo di un cammino minimo è un cammino minimo. (sottoproblema di problema ottimale è un sottoproblema ottimale)

Per assurdo > se un sottocammino non fosse minimo, esisterebbe un arco il cui peso sarebbe più piccolo di un altro arco > per cui il cammino minimo non sarebbe più minimo, e si contraddirrebbe l'ipotesi.

Rappresentazione dei cammini minimi

Come posso rappresentare un cammino minimo?

- Vettore predecessori $st[]$
- Sottografo predecessori
- Albero dei cammini > V' è l'insieme dei vertici raggiungibili da S ;
 - S è radice di albero > qualsiasi cammino minimo da $v \rightarrow S$ in G' è cammino minimo in G
 - Nei grafi non pesati basta fare una visita in ampiezza del grafo rappresentato come albero

3) Relaxation >> lasciare che una soluzione violi un vincolo temporaneamente, e poi aggiustare la violazione

Ovvero, sovrastimo il cammino minimo, e poi lo correggo. (*inizialmente metto le distanze di un nodo da tutti gli altri ad un numero altissimo, e poi quando li visiterò le correggerò con i valori effettivi – così sono sicuro di ottenere i valori minimi*)

Es. Dato un grafo e tre vertici $s \rightarrow u \rightarrow v \rightarrow$ Devo calcolare la distanza di un vertice s da un altro v . $s \rightarrow v$

- Sovrastimo inizialmente tutte le distanze da s ad infinito, tranne quella di s da sé stesso che è 0.
- Metto a confronto la distanza sovrastimata di $s \rightarrow u$ con il peso dell'arco $s \rightarrow u$ + la distanza di s dall'origine – se $w(s,u) + d(s) < d(u)$ allora aggiorno $d(u)$ ad un valore sensato $\rightarrow d(u) = d(s) + w(s,u)$
- **$D(u)$ è ora un valore sensato, ed è un cammino minimo da $u \rightarrow s$**

- Ripeto quindi lo stesso procedimento, ma ora tenendo come vertice u, e come arco $u \rightarrow v \ggg d(u) + w(u,v) < d(v)$?? **Ma $d(v)$ è sovrastimata ad infinito, quindi sicuramente sì $\rightarrow d(v) = d(u) + w(u,v)$**
- **Il cammino minimo da s-v sarà semplicemente $d(v)$.**

Relaxation – Lemmi

- 1) **Diseguaglianza triangolare** > un cammino minimo da s-v non può avere peso maggiore del cammino formato da un cammino minimo da s-u e da un arco u-v (se è così, aggiorno il valore della distanza dall'origine di v a cammino minimo s-u + peso(u-v))
- 2) **Proprietà del limite superiore** > una volta che la distanza di un vertice dall'origine assume un valore *minimo*, quello non cambia più. (una volta che ho finito di considerare un vertice e ho calcolato la distanza minima dall'origine, quel valore non cambia più)
- 3) **A seguito del rilassamento $d(v)$ non può aumentare** > al più diminuisce / rimane invariato -> comportamento che ci fa convergere verso la stima minima della distanza.
- 4) **Proprietà della convergenza** > dato un cammino minimo s-v, e un cammino minimo composto da cammino s-u + arco u-v, se prima del rilassamento $d(u) = \delta(s,u)$, dopo il rilassamento $d(v) = \delta(u,v)$
- 5) **Proprietà del rilassamento** > dopo tutti i passi di rilassamento sugli archi, $d(v) = \delta(s,v) \ggg d(v)$ non cambia più

Relaxation – Corollario

- 1) **Proprietà dell'assenza di cammino** > se non esiste un cammino S-V, allora $d(V) = \infty$

Relaxation – Applicazione

- **Dijkstra** > una volta per ogni arco
- **Bellman-Ford** > più volte

Algoritmo di Dijkstra

Ricordiamo che non si accettano archi a peso negativo e nessun ciclo – **strategia greedy**

- Utilizzo coda a priorità **PQ** con vertici ancora da stimare >> **perché?**
 - **Ad ogni passo**, prendo vertice con distanza minore rispetto a quello che stavo visitando. Questo viene fatto utilizzando la funzione PQextractmin()
- **PQempty == 1** >> algoritmo concluso.
- **Ad ogni passo, estraie un vertice da PQ, inserisce vertice in lista visited, e rilassa tutti gli archi uscenti da quel vertice.**

vertice iniziale s $\rightarrow d(s) = 0$, tutte le altre sono messe ad infinito.

Considero ora i suoi vertici vicini con i rispettivi archi e li rilasso.

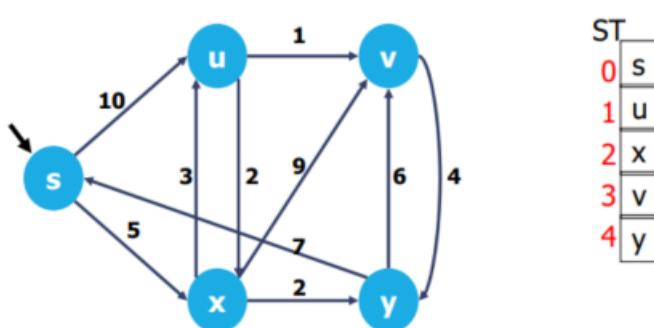
Vertice u $> d(s) + w(s,u) < d(u)$??? Si, $d(u)$ è inizialmente infinito. Aggiorno $d(u) = d(s) + w(s,u)$ e metto $st[u] = s$

Vertice x $> d(s) + w(s,x) < d(x)$??? Si, $d(x)$ è infinito. $d(x) = d(s) + w(s,x)$ e metto $st[x] = s$

Aggiungo il vertice attuale alla lista dei visitati S.

Procedo ora con il prossimo item nella PQ >> prendo x > rilasso tutti gli archi da lui uscenti, e aggiorno dove eventuali valori siano più piccoli di altri \rightarrow esempio con U >> $d(x) + w(x,u) < d(u)$??? Si, $5+3 < 10$. Aggiorno $d(u)$ e metto $st[u] = x$

Esempio



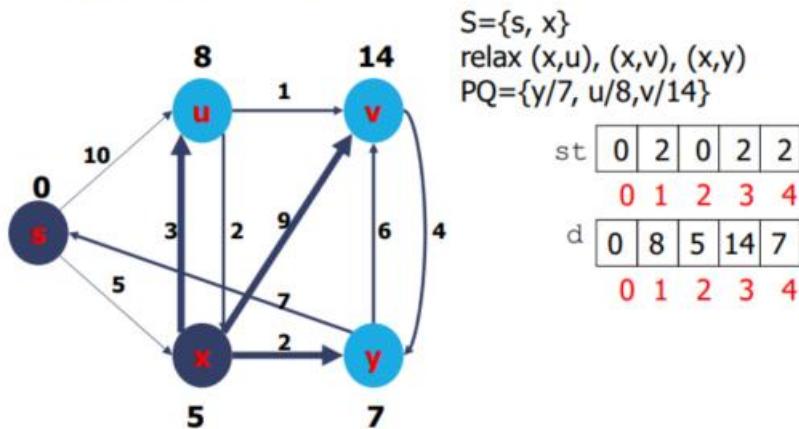
Prendo vertice x > prendo archi verso u,v e y

Verso u >>> applico relaxation, costo di raggiungimento di u + costo arco u > x è minore di distanza minima attuale di u da origine? Si, $(5+3) < 10$ >>> aggiorno distanza minima di u da origine al nuovo valore, e $st[u]$ diventa x

Applico poi stesso procedimento per v e y

$st[v]$ diventa x (potrebbe anche essere u, sono entrambi cammini di valore 9)

$st[y]$ diventa x, distanza 2



Implementazione – algoritmo di Dijkstra

```

void GRAPHspD(Graph G, int id) {
    int v;
    link t;
    PQ pq = PQinit(G->V);
    int *st, *d;
    st = malloc(G->V*sizeof(int));
    d = malloc(G->V*sizeof(int));

    for (v = 0; v < G->V; v++) { PQ con priorità in d
        st[v] = -1;
        d[v] = maxWT;
        PQinsert(pq, d, v); inizialmente tutti i nodi non hanno padri quindi metto -1 tutti distano infinito dalla sorgente tutti nodi vengono inseriti in coda a priorità
    }

    d[id] = 0; nodo di partenza è id > setto distanza da se stesso a 0, e il suo padre è sé stesso
    st[id] = id;
    PQchange(pq, d, id);
}

```

```

while (!PQempty(pq)) {procede finché coda a priorità non è vuota
    if (d[v = PQextractMin(pq, d)] != maxWT) prendi il primo elemento la cui distanza dall'attuale non è infinito (che è quindi raggiungibile)
        for (t=G->1adj[v]; t!=G->z ; t=t->next) se vertice è connesso, controllo sua lista adiacenze
            if (d[v] + t->wt < d[t->v]) { se distanza da vertice attuale + peso arco con vertice adiacente è minore di distanza del vertice preso in considerazione dall'origine, aggiorna
                d[t->v] = d[v] + t->wt;
                PQchange(pq, d, t->v); cambia il valore del vertice alla nuova distanza
                st[t->v] = v; cambiamo il padre dell'elemento adiacente preso in considerazione con il nodo attuale che stiamo visitando
            }
}

```

Complessità – Algoritmo di Dijkstra

Coda a priorità >>

- Estrazione minimo **O(logV)**
- Rilassamento di tutti gli archi (**O(E)**) uscenti da ogni vertice preso **O(logV)**

Quindi $T(n) =$

- **$O(V+E)*logV$**
- **$O(E*logV)$** >se tutti vertici sono raggiungibili da vertice di partenza S (*ho solo un vertice quindi da analizzare*)

Perché Dijkstra non accetta archi con peso negativo? (e anche cicli)

Anche se inizialmente un cammino minimo che tiene conto di pesi negativi potrebbe sembrare sensato, non risulta comunque ottimale – perché?

Assumiamo l'ipotetica situazione in cui arriviamo alla fine del calcolo di un cammino minimo, con un risultato – ma se potessimo continuare e considerare ancora un arco (introducendo così un ciclo), ad esempio un arco negativo, questo vorrebbe dire **diminuire il cammino minimo totale** rispetto al valore che abbiamo ->

- *Ciò vuol dire che quello che abbiamo ora allora non è un cammino minimo...*
- *Abbiamo quindi una soluzione non ottima.*

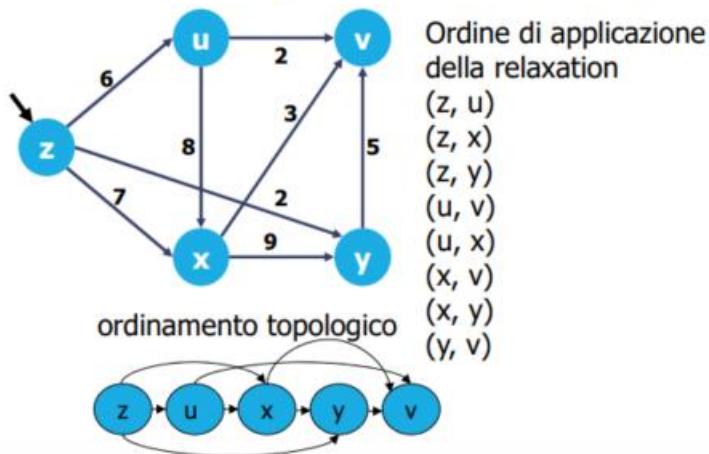
Cammini minimi – DAG

Grafi orientati pesati aciclici – DAG pesati

Dovrò eseguire quindi un ordinamento topologico del DAG, e poi per ogni vertice applicare la relaxation.

Esempio

I nodi compaiono con il loro nome originale per leggibilità



trovo quindi prima l'ordinamento topologico, e poi esegui per ogni vertice in ordine la relaxation

Complessità > $O(V+E)$

Applicabile anche a DAG con archi negativi

Applicazione reale – cammini minimi su DAG

Seam Carving, algoritmo di image resizing > si utilizza come modello un'immagine come DAG pesato di pixel.

Il peso di ogni arco è il contrasto tra 2 pixel.



Cammini MASSIMI – DAG pesati – implementazione

I cammini massimi non sono applicabili a qualsiasi tipo i DAG

La relaxation va applicata in modo *invertito* rispetto ai vertici >>

Semplicemente perché, se devo trovare un cammino massimo, allora qualsiasi distanza – se esistente- sarà maggiore di -inf.

```
if (d[v] < d[u] + w(u,v)) {    qui la relaxation ha effetto se AUMENTIAMO la stima
    d[v] = d[u] + w(u,v);
    st[v] = u;
}
```

cambiamo quindi il valore della distanza se quello nuovo è **maggior**e rispetto a quello precedente

Ne segue che inizialmente, non sovrastimeremo le distanze da ogni vertice a +inf ma a **-inf**

Algoritmo di Bellman-Ford

Bellman-Ford offre una soluzione anche nel caso di **archi negativi**

- Nel caso di cicli negativi però, li rileva semplicemente, e il risultato che offre non ha *senso*.

L'algoritmo di *Bellman-Ford* utilizza la **programmazione dinamica**.

- È applicabile? >> già dimostrato che i sotto problemi di un cammino minimo dovranno avere come soluzioni a loro volta cammini minimi
- la distanza minima da s->v con massimo V-1 archi è data da $d(v)$ che è il minimo tra $(d(v), \min(d(u) + w(u,v)))$, ovvero il valore minimo tra la distanza attuale da s di V e il minimo della distanza di un arco $u \rightarrow v$
- calcolo la distanza minima di un vertice rispetto agli altri prendendo prima in considerazione **1 arco, poi 2, poi 3...fino a V-1**

- anche qui includiamo solo cammini semplici e non cicli >> se ho cicli e pesi solo positivi, non mi serve ciclo perché aumenterei solo il peso che dovrebbe invece essere minimo; se invece ho pesi negativi, potrei trovare un cammino con peso sempre negativo, quindi il problema “*non avrebbe fine*” e quindi la soluzione non sarebbe sensata

Come? Spiegazione teorica

Eseguo V-1 iterazioni sul grafo, rilassando per ogni vertice tutti gli archi che escono da esso. (*dopo v-1 passi avrò sicuramente trovato cammino minimo, poiché avrò sfruttato ogni possibile distanza presente sul grafo*)

Ad ogni iterazione, si aggiorneranno eventuali valori delle distanze di vertici dall’origine – questo perché *potrei avere archi negativi, e quindi eventuali valori e distanze calcolate inizialmente potrebbero diminuire in una seconda o terza iterazione.*

Se, dopo aver completato una iterazione intera, e quindi aver visitato tutti i vertici, nessuna distanza cambia, allora posso interrompere l’iterazione.

- *Se invece non interrompo preventivamente, e arrivo all’ultimo passo (passo V) – eseguo ancora un altro passo aggiuntivo: se in questo passo aggiuntivo riesco ancora a diminuire le distanze di alcuni vertici, allora è presente un ciclo negativo, e il risultato non è un risultato ottimo. >> raggiungimento di un punto fisso”*

È quindi una versione “più completa” dell’algoritmo di Dijkstra, poiché invece di visitare una sola volta tutti i vertici, li visita al massimo v-1 volte.

Caso di terminazione e caso di ricorsione =>

- Caso terminale è in cui il vertice attuale è quello di origine, quindi $d = 0$
- Caso ricorsivo è in cui la distanza del vertice attuale dall’origine è data dalla distanza di un vertice intermedio dall’origine + distanza del vertice attuale dal vertice intermedio
 - Simile a Dijkstra in un certo senso – la distanza del nodo intermedio dall’origine sarà sempre la distanza minima calcolata fino ad ora, poiché applicherò ogni volta la relaxation
 - **Nota bene:** la versione ricorsiva in realtà utilizzerebbe la lista delle incidenze su ogni vertice, invece della lista di adiacenze / matrice - noi però non utilizzeremo tale implementazione nella soluzione **iterativa**

Calcolo bottom-up

- vettori ***st, d*** >> vettori distanze minime e predecessori
- V-1 passi
 - Ogni passo rilasso gli archi in avanti
- Al passo V (quindi successivo a ultimo) >> **se trovo miglioramento, e quindi ho una distanza nuova minore rispetto a prima >> c’è ciclo negativo**
 - Altrimenti ho distanza ottima e quindi soluzione ottima e accettabile
- **Attenzione!** Il numero di passi è obbligatoriamente *al massimo* V-1, come abbiamo già detto. Ma se notiamo che dopo un tot. Di passi, il valore complessivo del cammino minimo non cambia, possiamo interrompere.

Implementazione – Bellman-ford

```

void GRAPHspBF(Graph G, int id){
    int v, i, negcycfound; flag se abbiamo trovato un ciclo a peso negativo
    link t;
    int *st, *d;

    st = malloc(G->V*sizeof(int));
    d = malloc(G->V*sizeof(int));

    for (v = 0; v < G->V; v++) {
        st[v] = -1; inizializzo vettore parent ST e vettore distanze D
        d[v] = maxWT;
    }

    d[id] = 0; setto a 0 la distanza del nodo di partenza, e setto come suo
    st[id] = id; parent sé stesso

```

```

for (i=0; i<G->V-1; i++) ciclo su V-1 passi (w)
    for (v=0; v<G->V; v++) esaminiamo in ogni vertice
        if (d[v] < maxWT) se la sua distanza è < infinito
            for (t=G->adj[v]; t!=G->z ; t=t->next) cerchiamo dei vertici vicini ad esso
                if (d[t->v] > d[v] + t->wt) { per cui applicare o meno
                    d[t->v] = d[v] + t->wt;
                    st[t->v] = v; relaxation archi vicini
                }
        negcycfound = 0;
        for (v=0; v<G->V; v++)
            if (d[v] < maxWT)
                for (t=G->adj[v]; t!=G->z ; t=t->next)
                    if (d[t->v] > d[v] + t->wt)
                        negcycfound = 1;

```

se un ulteriore ciclo, una volta finito l'algoritmo, permette di trovare valori ancora più piccoli,
allora si ha un ciclo negativo, e il problema non ha una soluzione sensata.

notare come vi sia un ulteriore ciclo for() rispetto a Dijkstra - questo perché, poiché potrebbero esserci archi negativi, è necessario ripetere la visita più volte per trovare il cammino minimo.

Naturalmente questo comporterà una complessità nel caso peggiore maggiore rispetto a Dijkstra, che arriva al massimo ad un $O(E \cdot \log V)$

Complessità – Bellman-Ford

Al peggior dei casi, rilasso tutti gli archi V volte.

$$T(N) = O(|V| \cdot |E|)$$

Se uso un flag di terminazione in caso di raggiungimento di un punto fisso, posso avere una complessità di $O(|E|)$

Esempio di utilizzo dell'algoritmo di Bellman-ford >>> ARBITRAGE

Arbitrage definito in economia come l'acquisto e vendita in simultanea di beni su mercati diversi, per sfruttare piccole differenze nei prezzi.

Bellman-ford utilizzato nell'*'Arbitrage Trading program (ATP)*:

- Osserva prezzi su mercati, e rileva anomalie di prezzo in millisecondi >>
- Qui applica Bellman-ford, con l'obiettivo di rilevare *cicli a peso negativo*
 - Se non ne rileva, effettua transazioni

	VAL1	VAL2	VAL3	VAL4
VAL1	1.000	1.286	0.843	1.247
VAL2	0.777	1.000	0.655	0.966
VAL3	1.185	1.526	1.000	1.474
VAL4	0.802	1.035	0.678	1.000

Se ho quindi ad esempio 4 tassi di cambio, con bellman-ford trovo un cammino minimo > 1 (ciclo non negativo) che mi permette quindi di avere un guadagno positivo. Questo sarà il "percorso di valute" che dovrò fare per andare in positivo. (naturalmente non funziona a pieno poiché saranno presenti tasse di cambio su ogni valuta)