

## → LISTA SEMPLICE

Ogni elemento è implementato attraverso una struct:

```
struct elem {
    int inf;        // o qualsiasi tipo semplice
    elem* pun;      // definizione ricorsiva
};

elem *testa; // puntatore alla testa della
              lista

// oppure:
typedef elem* lista; → utilizzato solo per una maggiore leggibilità
lista testa;
```

### - ACCESSO AD UNA LISTA

- restituisce la testa di l (valore contenuto in inf)  
tipo\_inf **head**(lista l){  
 return l→inf; // testa di l  
}
- restituisce la coda di l (indirizzo in pun)  
lista **tail**(lista l){  
 return l→pun; // coda di l  
}

```
void stampalista(lista p){
    while (p != NULL) {
        cout<<head(p)<<" "; // stampa valore
        p = tail(p);         // spostamento sul
                               // prossimo elemento di p
    }
    cout<<endl ;
}
```

### - AGGIORNAMENTO DI UNA LISTA

- aggiunge un elemento e ad l e ritorna la lista aggiornata  
lista **insert\_elem**(lista l, elem\* e){  
 e→pun = l; /\* il puntatore del nuovo elemento non punta a NULL ma alla testa della  
 lista in cui lo andremo ad inserire. \*/  
 return e; /\* se avessi scritto return l, al prossimo insert\_elem la testa delle lista è l e  
 non 'e' che, invece, andremo a perdere. \*/  
}

// prima di poter essere passato ad insert\_elem, e deve essere allocato

```
lista crealista(int n){
    lista testa = NULL ;
    for (int i = 1 ; i <= n ; i++) {
        elem* p = new elem ;
        cout<<"Valore elemento "<<i<<": " ;
        cin>>p→inf ; // manca p→pun = NULL;
        testa=insert_elem(testa,p) ;
    }
    return testa ;
}
```

- elimina e da l e ritorna la lista aggiornata

```

lista delete_elem(lista l, elem* e){
    if (l == e) // controllo se e è la testa della lista
        l = tail(l);
    else{
        // localizzo l'elemento che punta ad e
        lista l1 = l; /* devo utilizzare una lista d'appoggio l1 perché altrimenti arrivato
                        alla fine ritorno come testa della lista la fine, mentre con l1
                        scorro / lasciandola invariata */

        while (tail(l1) != e && l1 != NULL) // localizzo l'elemento che punta ad e
            l1 = tail(l1);

        // aggiorno l'elemento che punta ad e
        l1->pun = tail(e);
    }
    delete e;
    return l; /* è necessario ritornare la lista aggiornata perché potrei dover cancellare il
               primo elemento della lista */
}

```

```

void eliminalista(lista& testa){
    while (testa != NULL)
        testa=delete_elem(testa,testa);
}

```

## - RICERCA DI UN VALORE INFORMATIVO

- cerca in l il valore v e, se esiste, restituisce il puntatore all'elemento che contiene v, altrimenti restituisce NULL.

```

lista search(lista l, tipo_inf v){
    while (l != NULL){ /* scorro la lista l, attraverso le primitive head e tail, fino ad
                        individuare l'elemento cercato, se esiste */

        if (head(l) == v)
            return l;
        else
            l = tail(l); /* posso modificare la lista senza problemi in quanto non la
                           ritorno e non la passo come riferimento, quindi continuerà a
                           puntare alla testa della lista */
    }
    return NULL;
}

```

```

int conta(lista l, int v){
    int occ = 0;
    while((l=search(l,v))!=NULL){
        l=tail(l);
        occ++;
    }
    return occ;
}

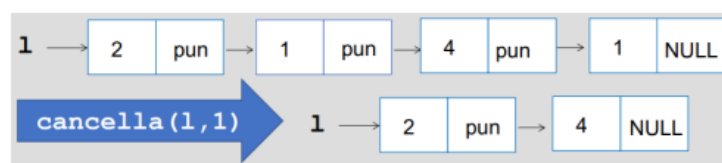
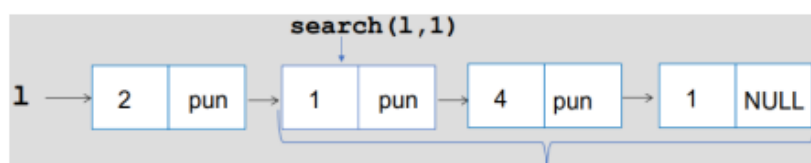
```

→ effettuata una search  
otteniamo la testa di una  
sottolista di /.  
re-iteriamo la ricerca sulla  
coda della sottolista (usando  
tail)

```

lista cancella(lista l, int v){
    elem* e;
    while((e=search(l,v))!=NULL)
        l=delete_elem(l,e);
    return l;
}

```



## - COPIA DI UNA LISTA

- copia un lista e ritorna una lista uguale alla stessa passata in ingresso

lista **copy**(lista l1){

```
    lista l=NULL;
    elem* curr;
    elem* prev=NULL;
    while(l1!=NULL){
        curr = new elem ;
        curr->inf = head(l1);
        curr->pun=NULL;
        if(prev==NULL) /* sto creando la testa */
            l=curr;
        else
            prev->pun=curr;
        prev=curr;
        l1=tail(l1);
    }
    return l;
}
```

Puntatore ausiliario: punta all'ultimo elemento inserito

Inserimento in fondo alla lista

Aggiornamento di prev



## → LISTA DOPPIA

Sia le liste doppie che le liste semplici implementano lo stesso tipo di dato, in modo, però, differente, il che permette alle liste doppie di ottimizzare il tutto.

```
struct elem {  
    int inf;  
    elem* pun; // punt. al prossimo elem  
    elem* prec; // punt. al precedente elem  
};  
  
typedef elem* lista;
```

## - LISTE SEMPLICI vs LISTE DOPPIE

Rispetto all'implementazione delle liste semplici cambia...

- il tipo di dato, in quanto è necessario AGGIUNGERE UN PUNTATORE all'elemento che precede l'elemento della lista considerato;
- L'IMPLEMENTAZIONE (NON l'interfaccia) delle PRIMITIVE che AGISCONO SUI PUNTATORI:  
*insert elem* / *delete elem* / *copy ...*  
(la *search* non cambia perché in ogni caso devo scorrere tutta la lista)
- aggiungiamo la primitiva *lista prev(lista)*, la quale restituisce la lista corrispondente all'elemento che precede l'elemento in input.

## - ACCESSO AD UNA LISTA

- restituisce la testa di *l* (valore contenuto in *inf*)  
tipo\_inf **head**(lista *l*){...}
- restituisce la coda di *l* (indirizzo in *pun*)  
lista **tail**(lista *l*){...}

## - AGGIORNAMENTO DI UNA LISTA

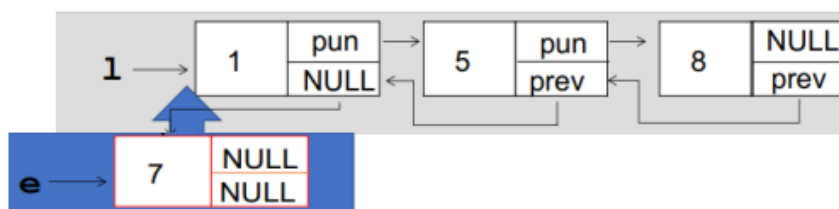
- aggiunge un elemento e ad *l* e ritorna la lista aggiornata.  
RISPETTO ALLE LISTE SEMPLICI... è necessario aggiornare anche i puntatori agli elementi precedenti

lista **insert\_elem**(lista *l*, elem\* *e*){

```
    e->pun=l;  
    if(l!=NULL)  
        l->prec=e;  
    e->prec=NULL;  
    return e;  
}
```

Dobbiamo aggiornare anche i puntatori *prev*

[ "if l != NULL" → perchè nelle liste VUOTE sia il *pun* che il *prev* sono NULL ]




- elimina *e* da *l* e ritorna la lista aggiornata.  
RISPETTO ALLE LISTE SEMPLICI... è possibile accedere direttamente al predecessore dell'elemento desiderato

```

lista delete_elem(lista l, elem* e){
  if(l==e)
    l=tail(l); // e è la testa della lista
  else // e non è la testa della lista
    (e->prev)->pun = e->pun;
    if(e->pun!=NULL)
      (e->pun)->prev=e->prev;
  delete e;
  return l;}

```

**LISTA DOPPIA**



Accedo direttamente a *e*  
e posso modificare  
precedente e successivo:  
Costo  $O(1)$

### - RICERCA DI UN VALORE INFORMATIVO

- cerca in *l* il valore *v* e, se esiste, restituisce il puntatore all'elemento che contiene *v*, altrimenti restituisce NULL.

lista **search**(lista *l*, tipo\_inf *v*){...}

### - COPIA DI UNA LISTA

- copia un lista e ritorna una lista uguale alla stessa passata in ingresso

lista **copy**(lista *l1*){...}

## → ALBERO

```
struct node {
    tipo_inf inf;
    node* parent; //opzionale
    node* firstChild;
    node* nextSibling;
};

typedef node* tree; //punta alla radice
                    //dell'albero
tree root; //variabile di tipo tree
```

```
int compare(tipo_inf s1, tipo_inf s2) {
    return strcmp(s1, s2);
}

void copy(tipo_inf* dest, tipo_inf source) {
    dest = new char[strlen(source)];
    strcpy(dest, source);
}

void print(tipo_inf inf) {
    cout << inf;
}
```

## - CREAZIONE DI UN ALBERO

- crea un nuovo nodo con valore informativo *i*

```
node* new_node(tipo_inf i) {
    node* n = new node;
    copy(n->inf, i);
    n->nextSibling = n->firstChild = n->parent = NULL;
    return n;
}
```

- aggiorna *p* inserendo il sottoalbero radicato in *c* come primo figlio di *p*

```
void insert_child(tree p, tree c) {
    c->nextSibling = p->firstChild;
    c->parent = p;
    p->firstChild = c;
}
```

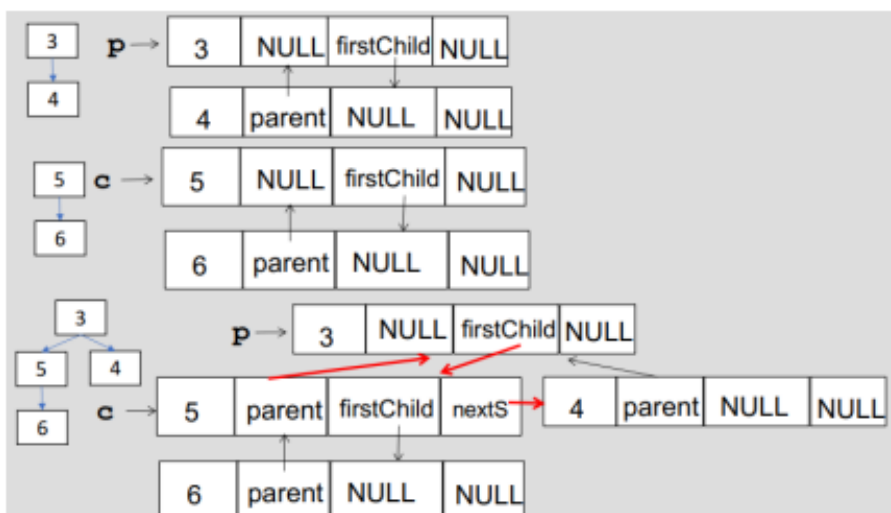
1. **aggiorno *c->nextSibling***

2. **aggiorno *c->parent***

3. **aggiorno *p->firstChild***

...dove modifico *p->firstChild* per ultimo altrimenti lo perdo.

**esempio:** (le operazioni sono rappresentate dalle frecce rosse)

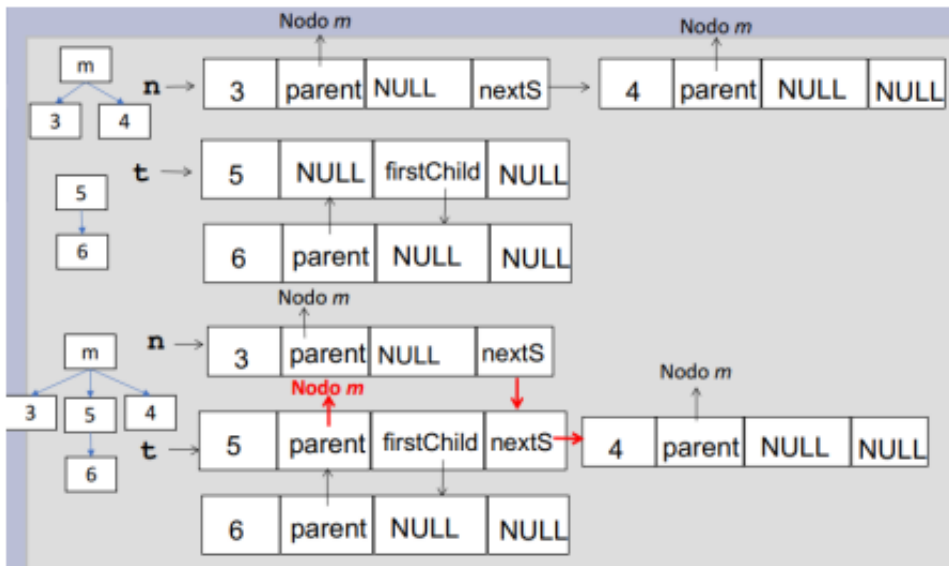


- aggiorna  $n$  inserendo il sottoalbero radicato in  $t$  come fratello successivo di  $n$

```
void insert_sibling(node* n, tree t){
    t→nextSibling = n→nextSibling;
    t→parent = n→parent;
    n→nextSibling = t;
}
```

1. aggiorno  $t \rightarrow \text{nextSibling}$
2. aggiorno  $t \rightarrow \text{parent}$
3. aggiorno  $n \rightarrow \text{nextSibling}$

esempio: (le operazioni sono rappresentate dalle frecce rosse)



## - ACCESSO AD UN ALBERO

- restituisce il contenuto informativo del nodo  $n$   
 tipo\_inf **get\_info**(node\*  $n$ ){  
     return  $n \rightarrow \text{inf}$ ;  
 }
- restituisce il padre del nodo  $n$   
 node\* **get\_parent**(node\*  $n$ ){  
     return  $n \rightarrow \text{parent}$ ;  
 }
- restituisce il primo figlio del nodo  $n$ , se esiste  
 node\* **get\_firstChild**(node\*  $n$ ){  
     return  $n \rightarrow \text{firstChild}$ ;  
 }
- restituisce il fratello successivo del nodo  $n$ , se esiste  
 node\* **get\_nextSibling**(node\*  $n$ ){  
     return  $n \rightarrow \text{nextSibling}$ ;  
 }

## → VISITA DFS RICORSIVA // ALBERO

### Algoritmo visitaDFSRicorsiva(nodo n) PRE-ORDINE !!!

1. visita il nodo  $n$  /\* la visita di un nodo può essere una qualsiasi operazione su di esso cout, somma ecc... \*/
2. for each child  $n'$  of  $n$   
    visitaDFSRicorsiva( $n'$ )

### N.B. !!!

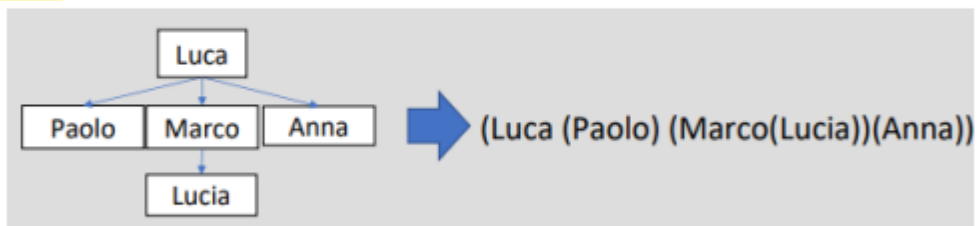
Sia  $T$  un albero non vuoto con radice  $n$  e  $k$  figli  $T_1, \dots, T_k$ , allora nella visita in...

- in **PRE-ORDINE** visito  $n$  e poi nell'ordine  $T_1, \dots, T_k$ ;
- in **POST-ORDINE** visito  $T_1, \dots, T_k$  e poi  $n$ ;
- in **IN-ORDINE** visito  $T_1, \dots, T_i$ , visito  $n$ , visito  $T_{i+1}, \dots, T_k$  per un prefissato  $i \geq 1$ .

### FORMALMENTE

Sia  $T$  un albero non vuoto con radice  $n$  e  $k$  figli  $T_1, \dots, T_k$   
 serializzazione( $n$ ): ( $n$  serializzazione( $T_1$ ) ... serializzazione( $T_k$ ))

esempio:



```

void serialize(tree t){
    cout<<" (";
    print(get_info(t));
    tree t1 = get_firstChild(t);
    while(t1!=NULL){
        serialize(t1);
        t1 = get_nextSibling(t1);
    }
    cout<<" )";
}
    
```

Visita del nodo

→ *pre-ordine*

Chiamata  
ricorsiva ai figli

(continuo fino a quando ne ho uno)

```

int dimensione(tree t){
    int dim=0,dim1;
    tree t1 = get_firstChild(t);
    while(t1!=NULL){
        dim1=dimensione(t1);
        dim+=dim1;
        t1 = get_nextSibling(t1);
    }
    return dim+1;
}
    
```

```

int altezza(tree t){
    if(get_firstChild(t)==NULL)
        return 0;
    int max=0,max_loc;
    tree t1 = get_firstChild(t);
    while(t1!=NULL){
        max_loc=altezza(t1);
        if(max_loc>max)
            max=max_loc;
        t1 = get_nextSibling(t1);
    }
    return max+1;
}
    
```



## → VISITA BFS ITERATIVA // CODA-BFS

### Algoritmo visitaBFSIterativa(nodo n)

1. coda C
2. enqueue(C,n)
3. while (C!=NULL) // fino a quando ho almeno un nodo da visitare
  - a. n=dequeue(C)
  - b. visita nodo n
  - c. for each child  $n'$  of  $n$   
enqueue( $n'$ )

La politica di accesso della coda implementata è di tipo **FIFO** ←

```
struct elemBFS
{
    node* inf;
    elemBFS* pun ;
};

typedef elemBFS* lista;

typedef struct{
    lista head;
    elemBFS* tail;} codaBFS;
```

## - CREAZIONE DI UNA CODA BFS / ELEMENTO DELLA CODA BFS

- *inizializza la coda settando a NULL i campi inf e pun*  
`codaBFS new_queue(){  
 codaBFS c = {NULL, NULL};  
 return c;  
}`
- *crea un nuovo elemento con valore informativo i (utilizzata in enqueue())*  
`static elemBFS* new_elem(tipo_inf i){  
 elemBFS* p = new elemBFS;  
 p→inf = i;  
 p→pun = NULL;  
 return p;  
}`

## - AGGIORNAMENTO DI UNA CODA BFS

- *crea ed inserisce un nuovo elemento i in fondo alla coda, ritornando, poi, la coda aggiornata*  
`codaBFS enqueue(codaBFS c, node* i){  
 elemBFS* e = new_elem(i);  
 if (c.tail != NULL)  
 c.tail→pun = e;  
 c.tail = e;  
 if (c.head == NULL)  
 c.head = c.tail;  
 return c;  
}`
- *rimuove e ritorna l'elemento in testa alla coda*  
`node* dequeue(codaBFS& c){  
 node* ris = (c.head)→inf;  
 c.head = (c.head)→pun;  
 return ris;  
}`

## - RICERCA DI UN INFORMAZIONE

- *restituisce il valore inf dell'elemento in testa*  
`node* first(codaBFS c){  
 return (c.head)→inf;  
}`
- *verifica se la coda è vuota (TRUE = vuota)*  
`bool isEmpty(codaBFS c){  
 if (c.head == NULL)  
 return true;  
 return false;  
}`

```

#ifdef DFS
int dimensione(tree t){
    int dim=0,dim1;
    tree t1 = get_firstChild(t);
    while(t1!=NULL){
        dim1=dimensione(t1);
        dim+=dim1;
        t1 = get_nextSibling(t1);
    }
    return dim+1;
}
#else
int dimensione(tree t){
    int count=0;
    codaBFS c = newQueue();
    c=enqueue(c,t);
    while(!isEmpty(c)){
        node* n=dequeue(c);
        count++;
        tree t1 = get_firstChild(n);
        while(t1!=NULL){
            c=enqueue(c,t1);
            t1 = get_nextSibling(t1);
        }
    }
    return count;
}

```

## → ALBERO BINARIO DI RICERCA (BST)

```
typedef int tipo_key;
typedef char* tipo_inf;
struct bnode {
    tipo_key key;
    tipo_inf inf;
    bnode* left;
    bnode* right;
    bnode* parent;};
typedef bnode* bst;
```

-----

### - DEFINIZIONE ALBERO BINARIO DI RICERCA

Un albero binario di ricerca (binary search tree BST) è un albero binario che soddisfa le seguenti PROPRIETA':

- ogni **nodo  $n$**  ha...
  - un **CONTENUTO INFORMATIVO**  $value(n)$ ;
  - una **CHIAVE**  $key(n)$  presa da un dominio totalmente ordinato, ovvero **su cui è definita una relazione d'ordine totale  $<$** ;
- sia  $n'$  un nodo nel **sottoalbero sinistro** di  $n$  allora...  
 $key(n') \leq key(n)$   
(oppure  $key(n') < key(n)$ )
- sia  $n'$  un nodo nel **sottoalbero destro** di  $n$  allora...  
 $key(n') > key(n)$   
(oppure  $key(n') \geq key(n)$ )

### - CREAZIONE E ACCESSO AD UN NODO DI UN BST

- crea un nuovo nodo con chiave e valore informativo dati in ingresso  
**bnode\* bst\_newNode(){**  
    bnode\* n = new bnode;  
    copy(n→inf, i);  
    copy\_key(n→key, k);  
    n→right = n→left = n→parent = NULL;  
    return n;  
}
- restituisce la chiave del nodo in ingresso  
    **tipo\_key get\_key(){**  
        return n→key;  
    }
- restituisce il valore del nodo in ingresso  
    **tipo\_inf get\_value(){**  
        return n→inf;  
    }
- restituisce il sottoalbero sinistro dell'albero in ingresso  
    **bst get\_left(bst t){**  
        return t→left;  
    }
- restituisce il sottoalbero destro dell'albero in ingresso  
    **bst get\_right(bst t){**  
        return t→right;  
    }
- restituisce il padre dell'albero in ingresso  
    **bnode\* get\_parent(bnode\* n){**  
        return n→parent;  
    }

## - AGGIORNAMENTO DI UN BST

- aggiunge un nodo all'albero di ricerca, mantenendo la proprietà di ricerca

```
void bst_insert(bst& b, bnode* n){
    bnode* x;
    bnode* y=NULL;
    if(b==NULL){
        b=n;
    }
    else{
        x=b;
        while (x != NULL) {
            y=x;
            if (compare_key(get_key(n),get_key(x))<0) {
                x = get_left(x);
            } else {
                x = get_right(x);
            }
        }
        n->parent = y;
        if (compare_key(get_key(n), get_key(y))<0) {
            y->left = n;
        } else {
            y->right = n;
        }
    }
}
```

Partendo dalla radice, il percorso di scansione dipende dall'esito del confronto tra il nodo corrente z e il nodo da inserire n:

- se  $key(n) < key(z)$  la scansione prosegue nel sottoalbero **sinistro**;
- se  $key(z) < key(n)$  la scansione prosegue nel sottoalbero **destro**.

La scansione termina quando il sottoalbero selezionato è vuoto, ovvero ha valore NULL.  
Al termine della scansione si inserisce il nuovo nodo.

```
void bst_insert(bst& b, bnode* n){
    if(b==NULL){ b=n;
                return;}
    if (compare_key(get_key(n),get_key(b))<0)
        if(get_left(b)!=NULL)
            bst_insert(b->left,n);
        else { b->left=n;
              n->parent=b;}
    else
        if(get_right(b)!=NULL)
            bst_insert(b->right,n);
        else { b->right=n;
              n->parent=b;}
}
```

### Algoritmo invisitaDFS-BST (nodo n)

1. If (get\_left(n)!=NULL)  
    invisitaDFS-BST(get\_left(n))
2. Visita nodo n
3. If (get\_right(n)!=NULL)  
    invisitaDFS-BST(get\_right(n))

→ per accedere a tutti i nodi di un BST in ORDINE CRESCENTE, implementiamo l'algoritmo di visita DFS in-order, il quale visita prima la parte sinistra dell'albero, il nodo padre ed infine la parte destra.

- cancella un nodo dall'albero di ricerca, mantenendo intatta la struttura dell'albero e la proprietà di ricerca

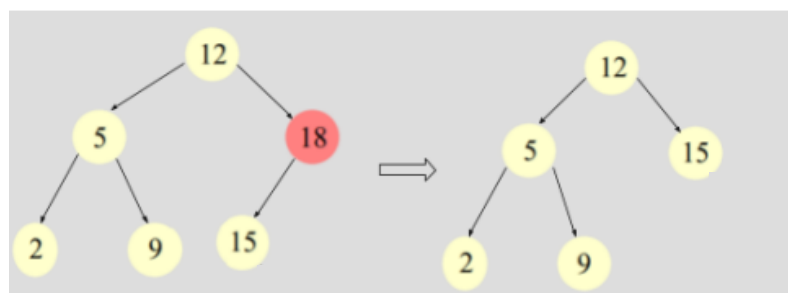
void **bst::delete**(bst& b. bnode\* n){

```

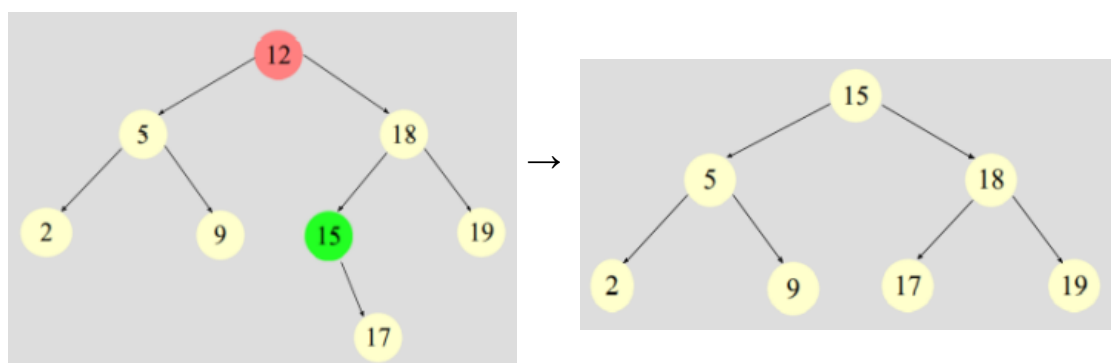
    bnode* new_child; // variabile d'appoggio che individua il nodo da sostituire a n come figlio
    if (get_left(n) == NULL) {
        if (get_right(n) == NULL) //Nodo foglia
            new_child=NULL;
        else {
            cout << "Nodo con solo figlio destro\n"; //Solo figlio destro
            new_child=get_right(n);
        }
    }
    else if (get_right(n) == NULL) { //Solo figlio sinistro
        cout << "Nodo con solo figlio sinistro\n";
        new_child=get_left(n);
    }
    else { //Entrambi i figli: cerco l'elemento più grande nel sottoalbero di sx
        cout << "Nodo con entrambi i figli\n";
        bnode* app = get_left(n);
        while (get_right(app) != NULL) //cerco l'elemento più a destra nel sottoalbero di sx
            app = get_right(app);
        if (get_left(app) == NULL) { //app è una foglia
            update_father(app, NULL);
        } else { //app ha il figlio sinistro
            app->parent->right = get_left(app);
            app->left->parent = get_parent(app);
        }
        // sostituisco app a n
        app->left = get_left(n);
        app->right = get_right(n);
        if (get_left(app) != NULL)
            (app->right)->parent = app;
        if (get_left(app) != NULL)
            (app->left)->parent = app;
        new_child=app;
    }
    if (new_child != NULL)
        new_child->parent = get_parent(n);
    if (n == b) // n è la radice
        b = new_child;
    else
        update_father(n, new_child);
    delete n;
}

```

- nodo foglia → il nodo viene semplicemente cancellato
- nodo con un solo figlio → il nodo  $n$  viene cancellato creando un collegamento tra il padre di  $n$  ed il figlio, sempre di  $n$ .



- nodo con due figli → di  $n$  cerchiamo il minore dei suoi successori, ossia il nodo più a sinistra del sottoalbero destro di  $n$ . (tale nodo avrà al più un figlio destro)



## - ACCESSO AD UN BST

- restituisce il nodo associato alla chiave in ingresso, se esiste

```
bnode* bst_search(bst b, tipo_key k){
    while (b != NULL){
        if (compare_key(k, get_key(b)) == 0)
            return b;
        if (compare_key(k, get_key(b)) < 0){
            b = get_left(b);
        } else {
            b = get_right(b);
        }
    }
    return NULL;
}
```

Scansione dell'albero a partire DALLA RADICE VERSO IL BASSO, sempre in base all'ESITO del CONFRONTO tra la chiave del nodo corrente z e la chiave da cercare k:

- se  $k = \text{key}(z)$  restituisco z;
- se  $k < \text{key}(z)$  la scansione prosegue nel sottoalbero sinistro;
- se  $\text{key}(z) < k$  la scansione prosegue nel sottoalbero di destra.

## → GRAFO CON LISTA DI ADIACENZA

### Struttura Lista di adiacenza

```
struct adj_node {
    int node;
    float weight;
    adj_node* next;
};
typedef adj_node* adj_list;
```

Puntatore al prossimo elemento della lista di adiacenza

Lista di adiacenza

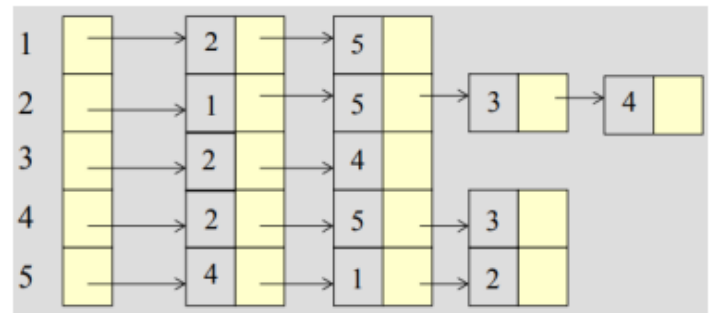
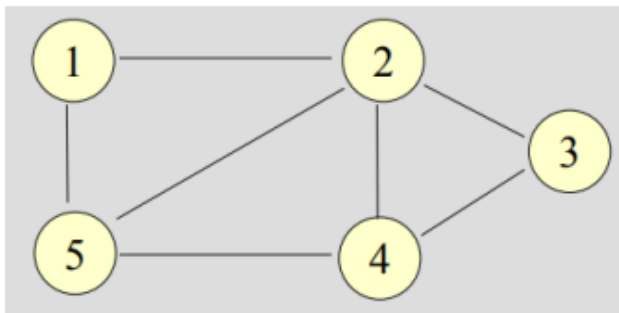
### Grafo

```
typedef struct {
    adj_list* nodes;
    int dim;
} graph;
```

Array dinamico di *dim* liste di adiacenze, una per vertice

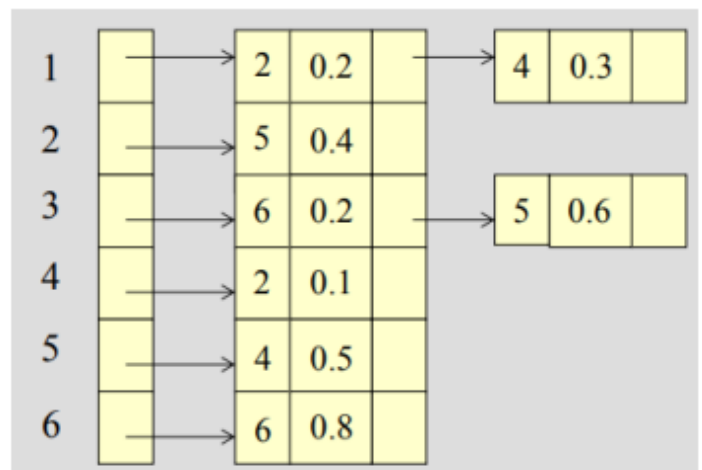
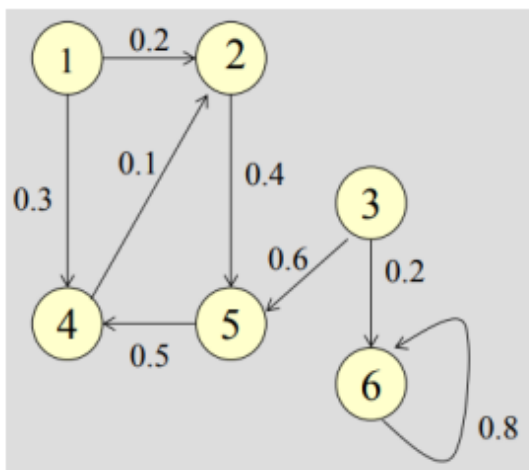
Numero dei vertici

### $G = (V, E)$ non orientato



Ogni arco  $(u, v)$  è memorizzato nella lista di adiacenza di  $u$  e nella lista di adiacenza di  $v$

### $G = (V, E)$ orientato e pesato



Il peso dell'arco  $(u, v)$  è memorizzato col vertice  $v$  nella lista di  $u$



(se il grafo non è pesato gli elementi della lista non presentano il campo centrale)

- le TESTE delle liste di adiacenza vengono memorizzate in un VETTORE DINAMICO della dimensione corrispondente al numero dei vertici graph:

```
graph g;
g.dim = ...;
g.nodes = new adl_list[g.dim];
```

- i VERTICI del grafo nell'implementazione a lista di adiacenza sono quindi identificati dagli indici  $0 \dots g.dim-1$ ;

## - CREAZIONE ED AGGIORNAMENTO DI UN GRAFO

- restituisce la rappresentazione di un grafo in  $n$  vertici identificati univocamente da 1 a  $n$  attraverso  $n$  liste di adiacenza

```
graph new_graph(int n){
    graph G;
    G.dim = n;
    G.nodes = new adj_list[n];
    for (int i=0; i<n; i++){
        G.nodes[i] = NULL;
    }
    return G;
}
```

- aggiunge l'arco orientato  $(s, d)$ , con peso  $w$ , in testa alla lista di adiacenza del nodo  $s$

```
void add_arc(graph& G, int u, int v, float w){
    adj_node* t = new adj_node;
    t->node = v-1;
    t->weight = w;
    t->next = G.nodes[u-1];
    G.nodes[u-1] = t;
}
```

- aggiunge l'arco NON orientato  $(s, d)$ , con peso  $w$ , alle liste di adiacenza dei nodi  $s$  e  $d$ .

```
void add_edge(graph& g, int u, int v, float w){
    add_arc(g, u, v, w);
    add_arc(g, v, u, w);
}
```

## - ACCESSO AD UN GRAFO

- restituisce il numero  $n$  dei nodi del grafo

```
int get_dim(graph g){
    return g.dim;
}
```

- restituisce l'identificativo del nodo contenuto nell'elemento della lista di adiacenza

```
int get_adjnode(adj_node* l){
    return (l->node+1);
}
```

- restituisce il peso

```
float get_adjweight(adj_node* l){
    return (l->weight);
}
```

- restituisce la testa della lista di adiacenza del nodo con identificativo in ingresso

```
adj_list get_adjlist(graph g, int u){
    return g.nodes[u-1];
}
```

- restituisce il prossimo elemento della lista di adiacenza

```
adj_list get_nextadj(adj_list l){
    return l->next;
}
```



```

void add(graph& g, int u, int v, float w, bool d) {
    if (d)
        add_arc(g,u,v,w);
    else
        add_edge(g,u,v,w);
}

```

```

graph g_build(ifstream &g, bool d, bool w) {
    int n;
    g >> n;
    graph G = new_graph(n);
    int v1,v2;
    if (w) {
        float w;
        while (g >> v1 >> v2 >> w) {
            add(G,v1,v2,w,d);
        }
    } else {
        while (g >> v1 >> v2) {
            add(G,v1,v2,1.0,d);
        }
    }
    return G;
}

```

```

//Stampa dell'array di liste
adj_list tmp;
for (int i=1; i<=get_dim(G); i++) {
    tmp=get_adjlist(G,i);
    cout << i;
    while (tmp) {
        cout << " --> " << get_adjnode(tmp);
        tmp=get_nextadj(tmp);
    }
    cout << endl;
}

```

## - INTRODUZIONE

Le visite dei grafi sono realizzate partendo dalla realizzazione degli algoritmi di visita degli alberi.

Dato un grafo  $G = (V, E)$ , gli algoritmi per la visita di un grafo partono da un vertice  $v \in V$  e visitano, a differenza degli alberi, **SOLO i VERTICI RAGGIUNGIBILI** da  $v$  attraverso gli archi  $E$ .

Visto che un grafo può contenere dei CICLI, sia la visita in ampiezza (BFS) che la visita in profondità (DFS) usano un ARRAY BOOLEANO di appoggio  $raggiunto[V]$  tale per cui, dato  $v \in V$ ,  $raggiunto[v] = \text{TRUE}$  se il nodo è già stato scoperto.

In questo modo EVITO di generare una VISITA INFINITA.

### Algoritmo visitaBFSIterativa(grafo $G=(V, E)$ , vertice $v$ )

- 1)  $raggiunto[V] = \text{FALSE}$
- 2) coda  $C$
- 3)  $\text{enqueue}(C, v)$
- 4)  $raggiunto[v] = \text{TRUE}$
- 5) **while** ( $C \neq \text{NULL}$ )    /\* similmente agli alberi, continuo la visita fino a quando  
                                      scopro nodi non ancora visitati \*/
  - a)  $u = \text{dequeue}(C)$
  - b) visita il vertice  $u$
  - c) **for each** vertex  $w$  of  $\text{Adj}[u]$   
      **if** !  $raggiunto[w]$ 
    - a)  $raggiunto[w] = \text{TRUE}$
    - b)  $\text{enqueue}(C, w)$

## - COMPONENTI CONNESSE

Sia  $G = (V, E)$  un grafo **NON ORIENTATO**, una **COMPONENTE CONNESSA** è un **SOTTOGRAFO**  $G'$  di  $G$  **CONNESSO** (ovvero ogni coppia di nodi  $G'$  è connessa) e **MASSIMALE** di  $G$ .

Per individuare le componenti connesse di un grafo non orientato, quindi...

1. eseguo la VISITA DEL GRAFO;
2. l'elenco dei NODI  $v$  che sono stati RAGGIUNTI dalla visita costituiscono una componente connessa;
3. se non ho raggiunto tutti i nodi torno al passo 1 partendo da uno dei nodi che non sono stati raggiunti dalla visita.

## - PROGETTAZIONE DEL CODICE

Rivedere la funzione `connected(graph g, node v)` affinché restituisca il VETTORE DEI NODI RAGGIUNGIBILI a partire da  $v$ .

Aggiungere al file `graph_connected_sol.cc` la procedura **`connected_component`** che...

- mantenga traccia dei nodi raggiunti (esplorati) attraverso il vettore  $raggiunto\text{-}globale[V]$  inizializzato a **FALSE**;
- richiami `connected`, per ogni  $v \in V$ , tale per cui  $raggiunto\text{-}globale[v] = \text{FALSE}$ ;
- usi il risultato di `connected` per...
  - stampare l'elenco dei nodi che fanno parte della stessa componente connessa;
  - aggiornare il vettore  $raggiunto\text{-}globale$  con il risultato di `connected`.

```

bool connected(graph g, int v){
    bool* raggiunto = new bool[get_dim(g)];
    int i;

    for(int i=0; i<get_dim(g); i++){
        raggiunto[i]= false;
    }
    codaBFS C=newQueue();
    raggiunto[v-1]=true;
    C=enqueue(C,v);
    while(!isEmpty(C)){
        int w=dequeue(C);
        for(adj_node* n=get_adjlist(g,w); n!=NULL; n=get_nextadj(n)){
            i = get_adjnode(n);
            if(!raggiunto[i-1]){
                raggiunto[i-1]=true;
                C=enqueue(C,i);
            }
        }
    }
    for(int i=0; i<get_dim(g); i++){
        if(!raggiunto[i])
            return false;
    }
    return true;
}

```

```

void connected_component(graph g) {
    bool* raggiunto_globale = new bool[get_dim(g)];
    for(int i=0; i<get_dim(g); i++){
        raggiunto_globale[i]= false;
    }

    // Prendo il primo nodo non raggiunto a false
    bool nodivisitati = false;
    while (!nodivisitati) {
        int j = -1;
        for(int i=0; i<get_dim(g) && j == -1 ; i++){
            if (raggiunto_globale[i] == false) {
                j = i;
            }
        }

        nodivisitati = true;
        if (j != -1) {
            bool* raggiunto = connected(g,j+1);
            cout << "Connected component: ";
            for(int i=0; i<get_dim(g); i++) {
                if (raggiunto[i]) {
                    cout << i+1 << " ";
                }
                raggiunto_globale[i] |= raggiunto[i];
                nodivisitati &= raggiunto_globale[i];
            }
            cout << endl;
        }
    }
}

```

## - ALBERO BFS E ALBERO DI COPERTURA

In una visita BFS gli archi che conducono a vertici ancora non visitati formano un albero detto ALBERO BFS la cui struttura dipende dall'ordine di visita.

Per costruire l'albero registriamo il padre di ogni nodo nel VETTORE DEI PADRI.

l'uso del vettore dei padri NON rappresenta un limite in questo caso perché la DIMENSIONE del GRAFO è NOTA e quindi è possibile allocare dinamicamente il vettore dei padri all'inizio della visita.

Se l'albero BFS include TUTTI I VERTICI G allora l'albero BFS ottenuto è un ALBERO DI COPERTURA o SPANNING TREE.

### Algoritmo visitaBFSIterativa(grafo $G=(V, E)$ , vertice $v$ )

- 1) raggiunto[V]=FALSE
  - 2) padre[V]=-1
  - 3) coda C
  - 4) raggiunto[v]=TRUE
  - 5) enqueue(C,v)
  - 6) while (C!=NULL)
    - a) u=dequeue(C)
    - b) for each vertex w of Adj[u]
      - If !raggiunto[w]
        - a) raggiunto[w]=TRUE // serve per evitare cicli
        - b) padre[w]=u
        - c) enqueue(C,w)
- |  
il vertice da cui si parte determina il modo in  
cui viene eseguita la visita

```
bool connected(graph g, int v){
    bool* raggiunto = new bool[get_dim(g)];
    int* padre = new int[get_dim(g)];
    int i;

    for(int i=0; i<get_dim(g); i++)
        raggiunto[i]= false;
    for(int i=0; i<get_dim(g); i++)
        padre[i]= -1;
    codaBFS C=newQueue();
    raggiunto[v-1]=true;
    C=enqueue(C,v);
    while(!isEmpty(C)){
        int w=dequeue(C);
        for(adj_node* n=get_adjlist(g,w); n!=NULL; n=get_nextadj(n)){
            i = get_adjnode(n);
            if(!raggiunto[i-1]){
                raggiunto[i-1]=true;
                padre[i-1]=w-1;
                C=enqueue(C,i);
            }
        }
    }
    for(int i=0; i<get_dim(g); i++)
        if(!raggiunto[i])
            return false;
    cout<<"Spanning tree del nodo "<<v<<endl;
    for(int i=0; i<get_dim(g); i++)
        if (padre[i]!=-1)
            cout<<"il padre del nodo "<<i+1<<" e' il nodo "<<padre[i]+1<<endl;
    return true;
}
```

**DIJKSTRA(grafo  $G=(V,E)$ , pesi  $w$ , sorgente  $s$ )**

1. INIZIALIZE ( $G,s$ )
2.  $S = \emptyset$  ←
3.  $Q = V$
4. while  $Q \neq \emptyset$
5.      $u = \text{extract-min}(Q)$  /\* estrarre il minimo ritorna il vertice di peso minimo trovato fino a quel momento \*/
6.      $S = S \cup \{u\}$  // aggiungo  $u$  ad  $S$
7.     for each vertex  $v$  of  $\text{Adj}[u]$
8.          $\text{RELAX}(u,v)$  /\* verifico se è possibile migliorare il cammino che collega i nodi adiacenti ad  $u$  \*/

Insieme  $S$  di vertici i cui pesi finali dei cammini minimi dalla sorgente  $s$  sono già stati determinati

**INIZIALIZE( $G,s$ )**

1. for each  $v \in V$
2.      $\text{dest}[v] = \infty$
3.      $\text{parent}[v] = \text{NULL}$
4.      $\text{dest}[s] = 0$  // inizialmente raggiungo i nodi pagando 0 [???

**RELAX( $u,v,w$ )**

1. If  $\text{dest}[v] > \text{dest}[u] + w(u,v)$  /\* se la distanza di quello adiacente è maggiore della
2.      $\text{dest}[v] = \text{dest}[u] + w(u,v)$  distanza minima fino a quel momento + il peso tra  $u$  e  $v$
3.      $\text{Decrease\_Priority}(Q,v,\text{dest}[v])$  allora aggiorno il cammino che porta a  $v$  \*/
4.      $\text{parent}[v] = u$  /\* allo stesso tempo aggiorniamo anche il padre perché nel mio percorso a  $v$  ci arriverò a partire da  $u$  \*/

Il grafo può essere rappresentato con **LISTE DI ADIACENZE**:

- $\text{Adj}[v]$  è la LISTA DI ADIACENZA del vertice  $v$ ;
- $w(u,v)$  è il PESO ASSOCIATO a  $v \in \text{Adj}[u]$ .

Usiamo il **MODULO GRAFO**:

- **dest** è un VEETTORE DINAMICO della dimensione di  $V$  che contiene la STIMA DEL CAMMINO MINIMO;
- **parent** è un VEETTORE DINAMICO che rappresenta il VEETTORE DEI PADRI;
- **Q** è una CODA CON PRIORITA' dove ogni vertice  $v \in V$  ha associato la stima del cammino minimo  $\text{dest}(v)$ .

Implementa la coda con priorità come **LISTA ORDINATA** dove gli elementi vengono mantenuti in ORDINE CRESCENTE per PESO  $w$ .

```
struct elem {
    int inf;
    float w;
    elem* pun ; };

typedef elem* codap;
```

```

codap enqueue(codap c, int i, float w) {
    elem *e = new_elem(i, w);
    if (c == NULL || e->w < c->w) {
        e->pun = c;
        return e;
    } else {
        codap c1 = c;
        while (tail(c1) != NULL && tail(c1)->w < e->w)
            c1 = tail(c1);
        e->pun = c1->pun;
        c1->pun = e;
        return c;
    }
}

```

```

int dequeue(codap &c) {
    int ris; // Commento ris
    ris = c->inf;
    elem *app = c;
    c = c->pun;
    delete app;
    return ris;
}

```

### Prim(grafo $G=(V,E)$ , pesi $w$ , radice $r$ )

1. INITIALIZE ( $G,s$ )
2.  $Q = V$
3. while  $Q \neq \emptyset$
4.      $u = \text{extract-min}(Q)$
5.     for each vertex  $v$  of  $\text{Adj}[u]$
6.         RELAX( $u,v,w$ )

#### INITIALIZE( $G,r$ )

1. for each  $v \in V$
2.      $\text{key}(v) = \infty$
3.      $\text{parent}[v] = \text{NULL}$
4.      $\text{key}[r] = 0$

#### RELAX( $u,v,w$ )

1. If  $\text{key}[v] > w(u,v)$
2.      $\text{key}[v] = w(u,v)$
3.      $\text{parent}[v] = u$
4.     Decrease\_Priority( $Q,v,\text{key}[v]$ )



N.B.

Il nome del file, la tipologia di grafo, orientato/non orientato, pesato/non pesato, vengono passati come argomenti alla chiamata del programma.

esempio:

`graph graph1 1 0`

- richiama l'eseguibile `graph...`
- ...passandogli come parametri il file «`graph1`» e...
- ...specificando che...
  - il grafo è orientato (1) e...
  - ...non pesato (0).

**RICORDA !!!**

eseguire un programma equivale a chiamare la funzione `main`

Per questo motivo, per leggere tali argomenti da linea di comando riscriviamo la funzione `main` nel seguente modo:

**`int main(int argc, char *argv[])`**

riceve in ingresso due argomenti:

- un intero `argc` che corrisponde al NUMERO di argomenti;
- un array di stringhe `argv` che contiene gli ARGOMENTI, uno per stringa.

per convenzione `argv[0]` contiene il nome con il quale il programma è stato invocato.

Quindi...

- `argc` vale sempre almeno 1;
- gli argomenti passati al programma sono memorizzati nelle stringhe `argv[1]... argv[argc-1]`.

esempio:

```
int main(int argc, char *argv[])
{ /* main che stampa gli argomenti */
  for (int i = 0 ; i < argc ; i++)
    cout<<argv[i]<<endl ;
  return 0 ;
}
```

```
int main(int argc, char *argv[]) {
/* Se il numero di parametri con cui e' stato chiamato il client e'
inferiore a tre - si ricordi che il primo parametro c'e' sempre, ed e' il
nome del file eseguibile - si ricorda all'utente che deve inserire
anche il nome del file che descrive il grafo e il flag weighted*/

if (argc<3) {
cout << "Usage: " << argv[0] << " filename directed weighted\n";
exit(0);
};

ifstream g;
g.open(argv[1]);
cout << argv[1] << " " << argv[2] << " " << argv[3]<<endl;
int directed = atoi(argv[2]);
int weighted = atoi(argv[3]);

/*Chiamata a g_build che costruisce un grafo*/
graph G=g_build(g, directed, weighted);

cout<<get_dim(G)<<endl;
```