Informatica I

6 – Strutture dati

2,7,14 Maggio 2013

Corso di Laurea in Matematica e applicazioni

Università di Camerino

A.A. 2012/2013

Agenda

- Array e liste
- Pile e code
- Grafi e alberi

Ricapitolando...

Dato

E' un valore che può assumere un'espressione.
Es. `3, x+y, y!=(x % 5)`

Tipo di dato

Determina l'insieme dei valori che un'espressione può assumere e le operazioni sugli stessi dati.
Es. `String` (`substring()`, `toUpperCase()`, ...)

Struttura dati

E' un'entità usata per organizzare un insieme di dati, insieme alle operazioni per manipolare gli elementi della struttura.
Es. `Array`(`sort()`, `reverse()`, ...)

Alcune caratteristiche

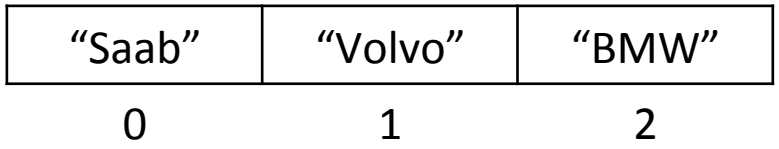
Le strutture dati possono essere classificate in base alla disposizione dei dati, al loro numero, ed al loro tipo, in:

- **lineari**: i dati sono disposti in sequenza e possono essere nominati come primo, secondo, terzo, (es. `a[0]`, `a[1]`, …)
- **non lineari**: i dati non sono disposti in sequenza
- **a dimensione fissa**: il numero degli elementi è costante
- **a dimensione variabile**: il numero degli elementi può variare durante l'esecuzione
- **omogenee**: i dati sono tutti dello stesso tipo
- **non omogenee**: i dati possono avere tipi diversi

Array e liste

Array

L'array è una struttura dati costituita da una sequenza indicizzata di elementi. -> **struttura lineare**



- Nella maggior parte dei linguaggi di programmazione, l'array è anche a **dimensione fissa** e **omogeneo**.
- L'Array JS invece è **a dimensione variabile**, e **non omogeneo**

Liste concatenate

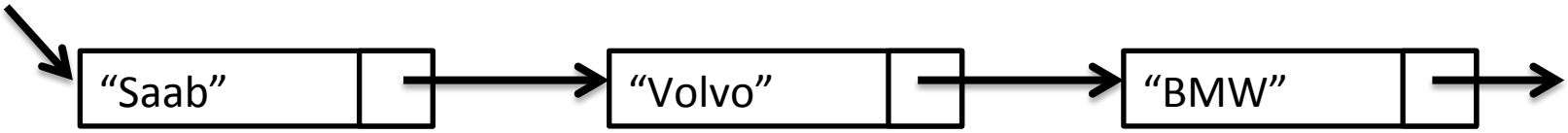
Una **lista concatenata** è una struttura dati in cui ogni elemento possiede un **link** (un collegamento) a un altro elemento

•Mentre nell'array si accede in **modo indicizzato**, nelle liste concatenate gli elementi vengono acceduti in **modo sequenziale** (uno dopo l'altro).

• Le liste permettono **l'aggiunta/rimozione dinamica** di elementi in modo efficiente

• Ogni elemento della lista contiene le informazioni necessarie per accedere all'elemento successivo.

Liste concatenate



Gli elementi (in questo caso stringhe: "Saab", "Volvo", "BMW") sono inseriti in un **nodo** (oggetto `Node`), un oggetto che memorizza il collegamento con il **nodo successivo** della lista (proprietà `next`)

Oggetto Node (funzione costruttore)

```
function Node(value, next){
    this.value=value;
    this.next=next;
}
```

```
var n1=new Node("Saab");
var n2=new Node("Volvo");
var n3=new Node("BMW");
n1.next=n2;
n2.next=n3;
```

oppure

```
var n3=new Node("BMW");
var n2=new Node("Volvo", n3);
var n1=new Node("Saab", n2);
```

Che valore `n3.next`?

`undefined`

Liste concatenate

• La lista concatenata (oggetto `LinkedList`) contiene il riferimento a un solo nodo, la **testa della lista** (il primo elemento, proprietà `head`).

• Gli altri elementi possono essere acceduti in modo sequenziale partendo dal nodo head e passando di next in next

Oggetto LinkedList (funzione costruttore)

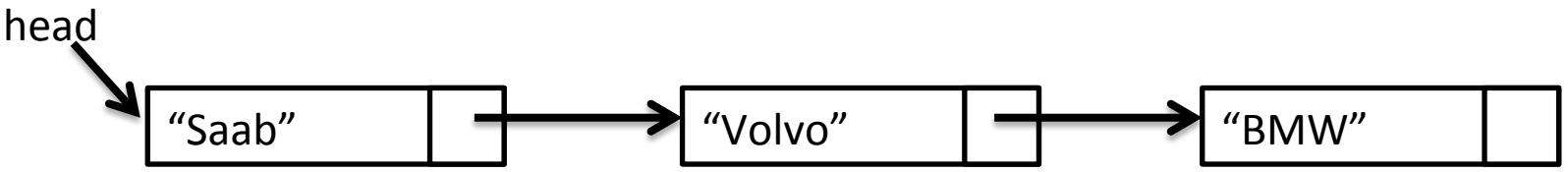
```
function LinkedList(head){
    this.head = head;
}
```

```
var n3=new Node("BMW");
var n2=new Node("Volvo", n3);
var n1=new Node("Saab", n2);
//n1 è la testa della lista
var list=new LinkedList(n1)
```

Chi è la **coda** (l'ultimo elemento) della lista?

E' `n3`, ovvero il primo nodo `n`, tale che `n.next` è `undefined`

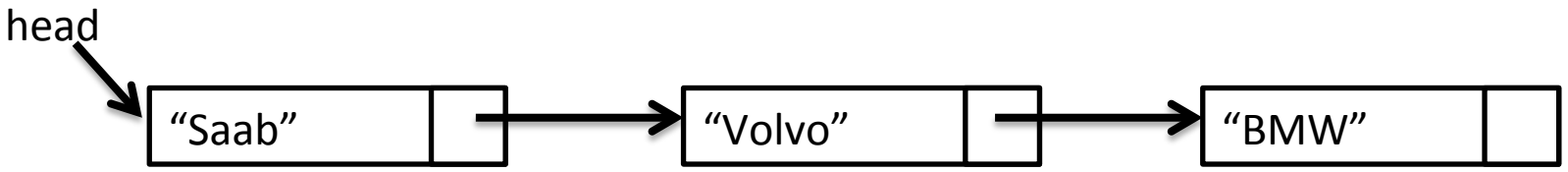
Liste concatenate



Come visito tutti gli elementi della lista?

```
for(var n = list.head; n!=undefined; n=n.next){
    //faccio qualcosa con n, ad esempio
    print(n.value);
}
```

Liste concatenate



Se voglio accedere al k-esimo elemento?

Accesso indicizzato (Array)

```
//direttamente
a[k];
```

Veloce: tempo di accesso costante

Accesso sequenziale (LinkedList)

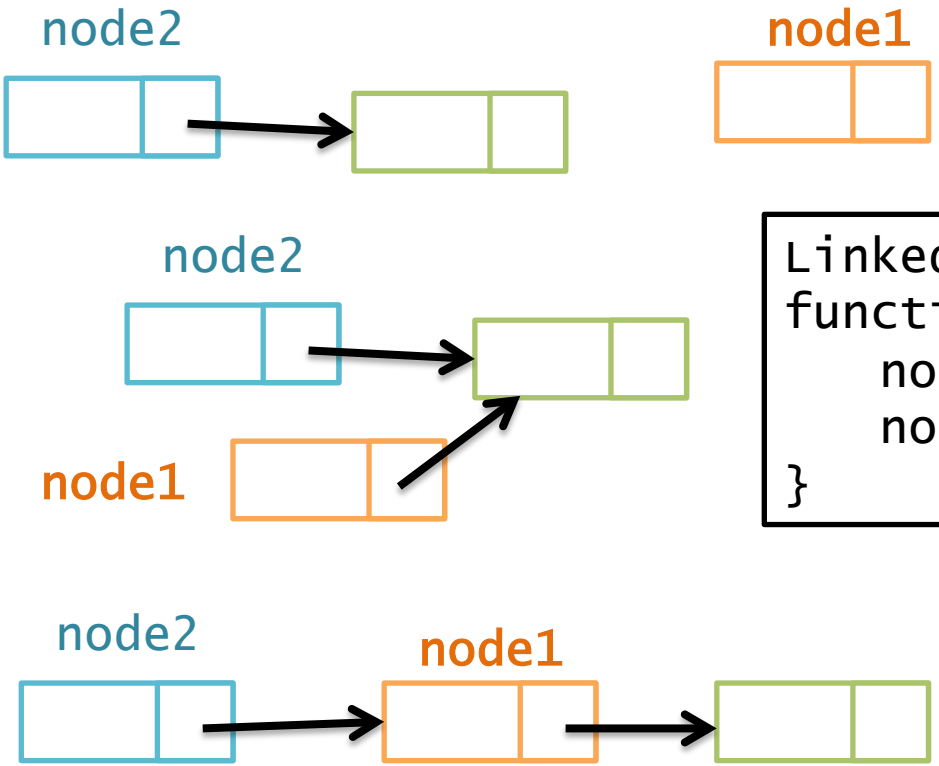
```
//devo accedere a tutti i precedenti
var n=list.head;
var i=0;
while(n!=undefined && i<k){
    n=n.next;
    i++;
}
n;
```

Meno efficiente: nel caso peggiore, tempo di accesso proporzionale alla lunghezza della lista

Operazioni su liste concatenate

insertAfter(node1,node2)

Inserisce il nodo node1 dopo il nodo node2

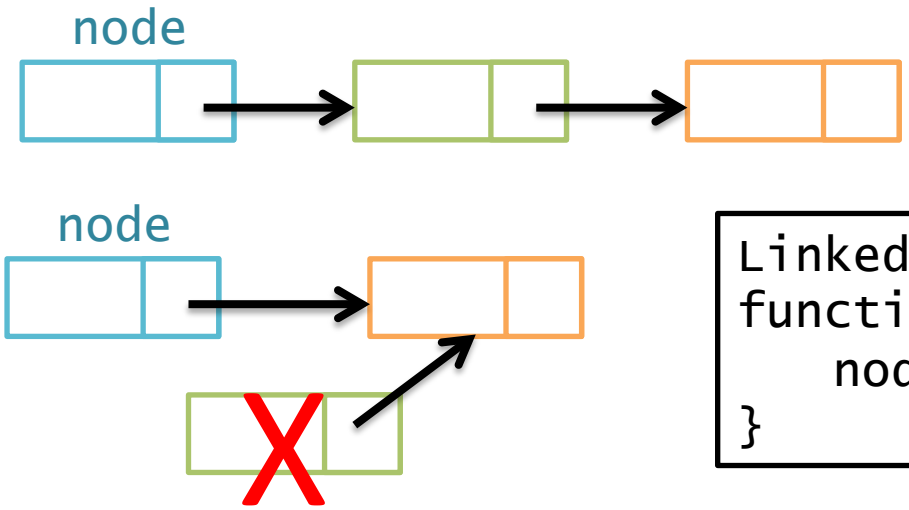



```
LinkedList.prototype.insertAfter =
function(node1,node2){
    node1.next=node2.next;
    node2.next=node1;
}
```

Operazioni su liste concatenate

removeAfter(node)

Rimuove il nodo dopo node




```
LinkedList.prototype.removeAfter =
function(node){
    node.next=node.next.next;
}
```

Operazioni su liste concatenate

Inserimento e rimozione nelle liste richiedono un solo comando
→**veloce: tempo costante**

Nell'array inserimento e rimozione hanno un tempo di esecuzione proporzionale al numero dei suoi elementi. In JS utilizziamo il metodo *splice*

```
function insertAfter(array, element, k){
    var a=new Array(array.length + 1);
    a[k+1]=element;
    for(var i=0; i<array.length; i++){
        //j=1 se i>k, 0 altrimenti
        var j = (i>k)? 1:0;
        a[i + j] = array[i];
    }
    return a;
}
```

Operazioni su liste concatenate

Inserimento e rimozione nelle liste richiedono un solo comando
→**veloce: tempo costante**

Nell'array inserimento e rimozione hanno un tempo di esecuzione proporzionale al numero dei suoi elementi. In JS utilizziamo il metodo *splice*

```
function removeAfter(array, k){
    var a=new Array(array.length - 1);
    for(var i=0; i<array.length; i++){
        var j = (i>k)? -1:0;
        if(i!=k+1)
            a[i + j] = array[i];
    }
    return a;
}
```

Operazioni su liste concatenate

removeAll()

Rimuove tutti gli elementi

```
LinkedList.prototype.removeAll = function(){
    this.head=undefined;
}
```

isEmpty()

Restituisce true se la lista è vuota, false altrimenti.

```
LinkedList.prototype.isEmpty = function(){
    return this.head==undefined;
}
```

Operazioni su liste concatenate

getAt(k)

Restituisce il k-esimo elemento della lista

```
LinkedList.prototype.getAt = function(k){
    var n=this.head;
    var i=0;
    while(n!=undefined && i<k){
        n=n.next;
        i++;
    }
    return n;
}
```

Operazioni su liste concatenate

removeAt(k)

Rimuove il k-esimo elemento della lista

```
LinkedList.prototype.removeAt = function(k){
    if(k==0)
        this.head=this.head.next;
    else{
        var n=this.head;
        var i=0;
        while(n.next!=undefined && i<k-1){
            n=n.next;
            i++;
        }
        n.next=n.next.next;
    }
}
```

Esercizio 1

getLast()

Restituisce l'ultimo elemento della lista

```
LinkedList.prototype.getLast = function(){
    var n=this.head;
    while(n.next!=undefined)
        n=n.next;
    return n;
}
```

Esercizio 2

size()

Restituisce la dimensione della lista, ovvero il numero dei suoi elementi

```
LinkedList.prototype.size = function(){
    var count=0;
    for(var n=this.head; n!=undefined; n=n.next)
        count++;
    return count;
}
```

Esercizio 3

contains(element)

Restituisce `true` se la lista contiene l'elemento specificato, `false` altrimenti.

```
LinkedList.prototype.contains = function(element){
    for(var n=this.head; n!=undefined; n=n.next)
        if(n.value==element)
            return true;
    return false;
}
```

```
LinkedList.prototype.contains = function(element){
    var trovato=false;
    for(var n=this.head; n!=undefined&&!trovato; n=n.next)
        if(n.value==element)
            trovato=true;
    return trovato;
}
```

Esercizio 4

insertAt(element,k)

Inserisce `element` in posizione `k` nella lista

```
LinkedList.prototype.insertAt = function(element, k){
    var node = new Node(element);
    if(k==0){
        node.next=this.head;
        this.head=node;
    } else{
        var n = this.head;
        for(var i=1; n.next!=undefined && i<k-1; i++)
            n=n.next;
        node.next=n.next;
        n.next=node;
    }
}
```

Esercizio 5

getAll(element)

Restituisce un Array con tutti i nodi della lista che hanno come valore `element`

```
LinkedList.prototype.getAll = function(element){
    var a = new Array();
    for(var n=this.head; n!=undefined; n=n.next)
        if(n.value==element)
            a[a.length]=element;
    return a;
}
```

Pile e code

Struttura dati Pila (Stack)

Lo **stack** (o pila) è una struttura dati in cui gli elementi sono gestiti tramite una politica **LIFO (Last In First Out)**, ovvero l'ultimo ad essere stato inserito è il primo ad essere estratto

Supporta due operazioni principali:

- `push(element)`: aggiunge `element` in cima allo stack
- `pop()`: estrae (ovvero rimuove e restituisce) l'elemento in cima allo stack

Struttura dati Pila (Stack)

```
Stack s
```


```
s.push("Fiat")
```


```
var car = s.pop()
```


```
car: "Fiat"
```

Implementazione con Array

- L'Array JS fornisce già le operazioni di push e pop
- Nella maggior parte dei linguaggi, questo non avviene

Vediamo l'implementazione con Array

Costruiamo una struttura dati costituita da un array (proprietà a) e da una variabile che memorizza l'indice dell'ultimo elemento (proprietà n)

Oggetto Stack (costruttore)

```
function Stack(){
    this.a=new Array();
    this.n=-1;
}
```

Implementazione con Array

push

```
Stack.prototype.push = function(element){
    this.a[n++]=element;
}
```

pop

```
Stack.prototype.pop = function(){
    if(this.n>=0)
        return this.a[--n];
}
```

Quale differenza tra n++ (o n--) e ++n (o --n)?

L'espressione n++ valuta n dopo essere stato incrementato
L'espressione ++n valuta n prima che venga incrementato

Es. se n=1, n++ è valutato a 2, mentre ++n è valutato a 1. In ogni caso però la variabile n viene incrementata.

Implementazione con Array

top()

Restituisce l'ultimo elemento dello stack (senza rimuoverlo)

```
Stack.prototype.top = function(){
    return this.a[n];
}
```

isEmpty()

Restituisce true se lo stack è vuoto, false altrimenti.

```
Stack.prototype.isEmpty = function(){
    return this.n==-1;
}
```

Implementazione con Array

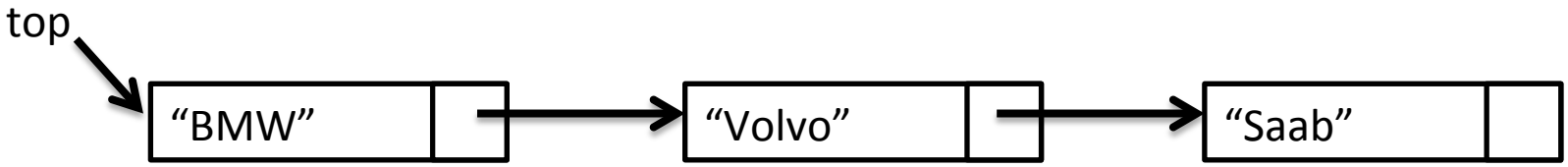
size()

Restituisce il numero di elementi dello stack

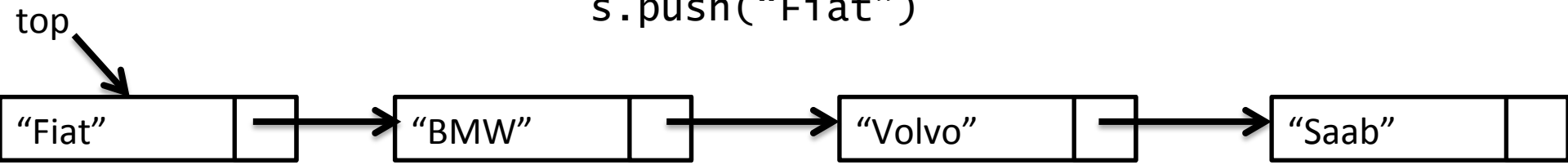
```
Stack.prototype.size = function(){
    return this.n+1;
}
```

Implementazione con LinkedList

IDEA: anzichè aggiungere alla fine della lista, gli elementi vengono aggiunti all'inizio. Quindi list.head è il top dello stack.

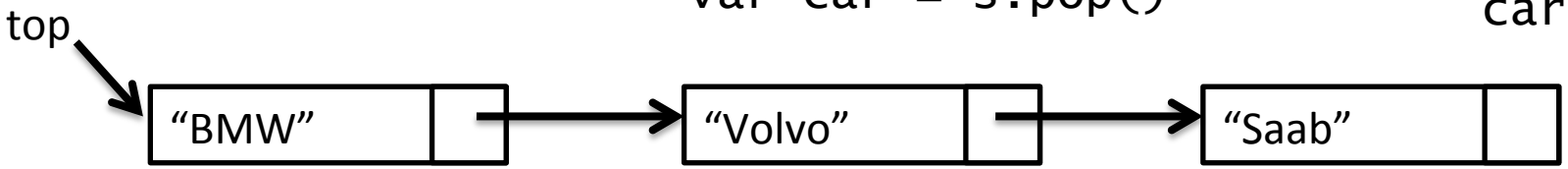



```
s.push("Fiat")
```




```
var car = s.pop()
```

car: "Fiat"



Implementazione con LinkedList

LinkedList (costruttore)

```
function LinkedList(head){
    this.head = head;
}
```

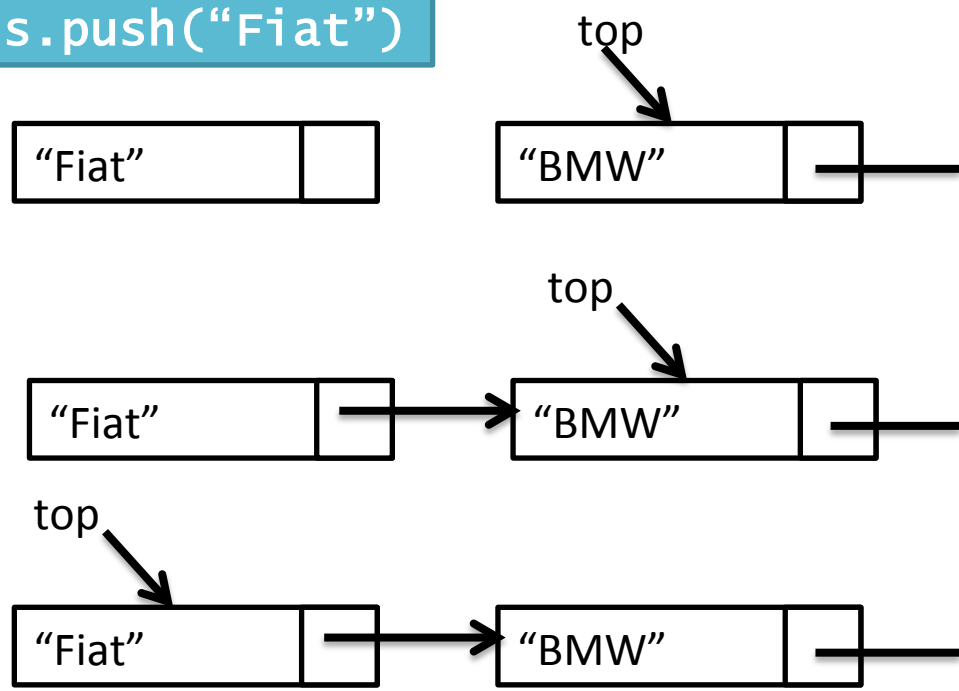
Stack (costruttore)

```
function Stack(){
    //this.top=undefined;
}
```

push

```
Stack.prototype.push =
function(element){
    var node=new Node(element);
    node.next=this.top;
    this.top=node;
}
```

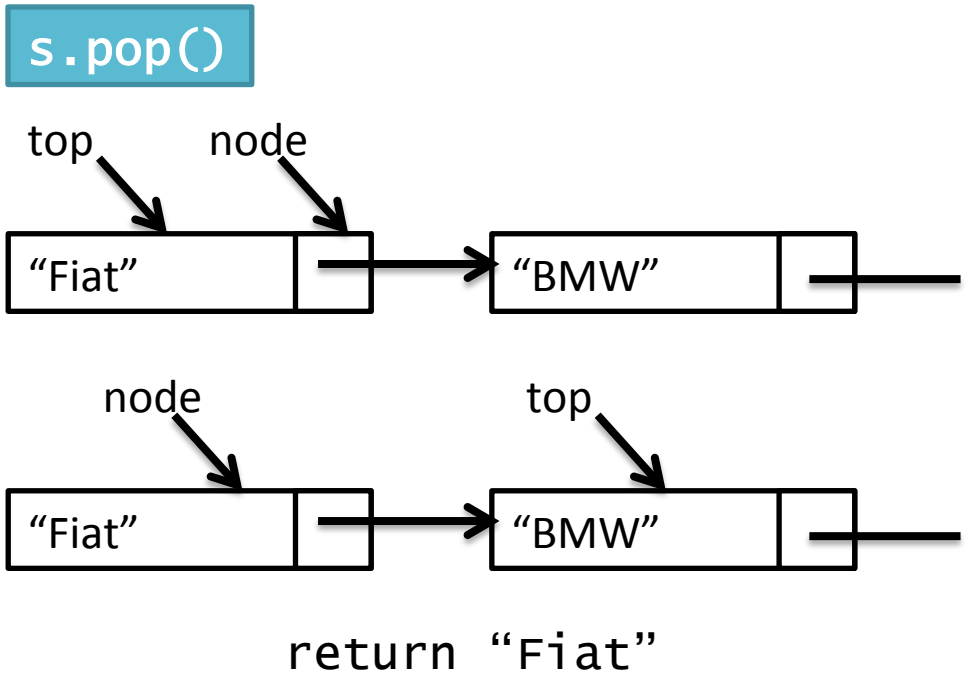
s.push("Fiat")



Implementazione con LinkedList

pop

```
Stack.prototype.pop =
function(){
    var node = this.top;
    this.top = this.top.next;
    return node.value;
}
```



Implementazione con LinkedList

top()

```
Stack.prototype.top = function(){
    return this.top.value;
}
```

isEmpty()

```
Stack.prototype.isEmpty = function(){
    return this.top==undefined;
}
```

Struttura dati Coda (Queue)

La **queue** (o **coda**) è una struttura dati in cui gli elementi sono gestiti tramite una politica **FIFO (First In First Out)**, ovvero il primo ad essere stato inserito è il primo ad essere estratto (viene estratto quello che rimane nella coda più a lungo)

Supporta due operazioni principali:

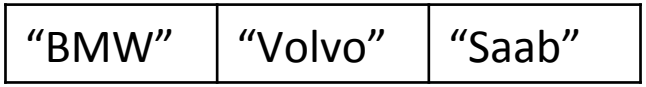
- `put(element)`: aggiunge `element` nella coda
- `get()`: estrae (ovvero rimuove e restituisce) l'elemento della coda meno recemente inserito

L'Array JS supporta due operazioni equivalenti

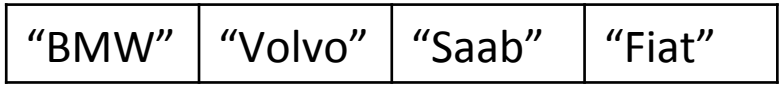
shift()	Rimuove il primo elemento di un array, e lo restituisce
unshift()	Aggiunge nuovi elementi all'inizio dell'array e restituisce la nuova lunghezza

Struttura dati Coda (Queue)

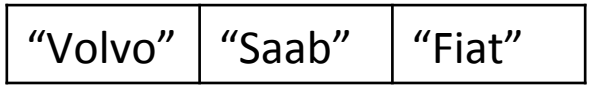
```
Queue q
```




```
q.put(“Fiat”)
```



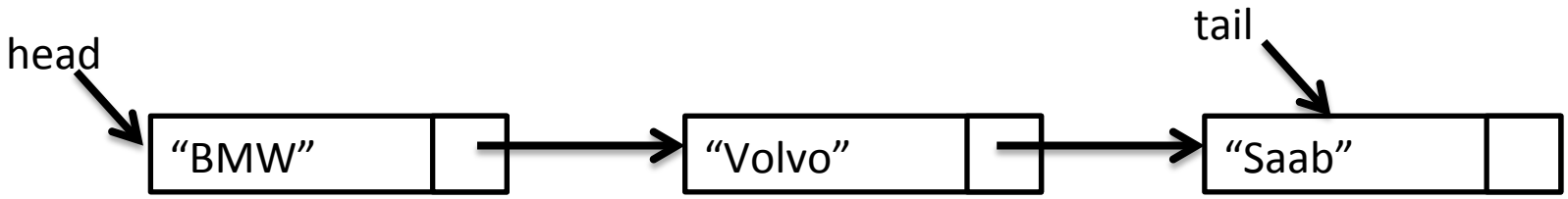

```
var car = q.get()
```

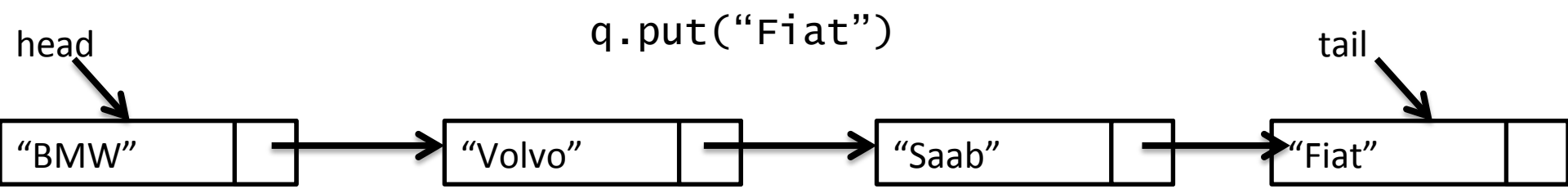


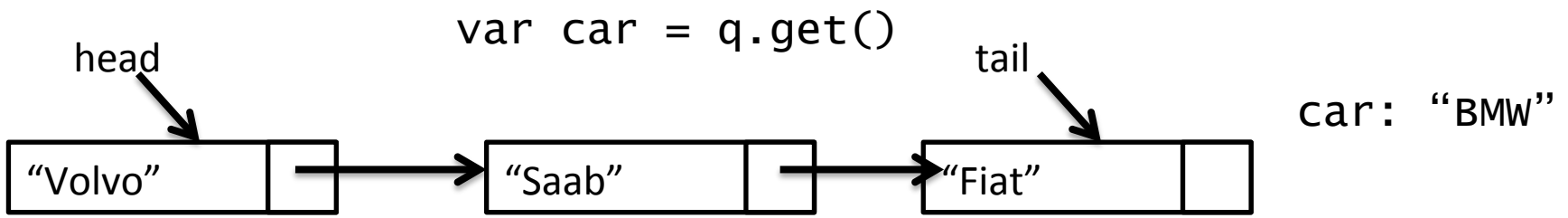

```
car: “BMW”
```

Implementazione con LinkedList

IDEA: utilizzare due puntatori, uno alla testa (**head**) uno alla fine (**tail**) della coda.







Implementazione con LinkedList

Queue (costruttore)

```
function Queue(){
}
```

isEmpty()

```
Queue.prototype.isEmpty = function(){
    return this.head==undefined;
}
```

first()

```
Queue.prototype.first = function(){
    return this.head.value;
}
```

Implementazione con LinkedList

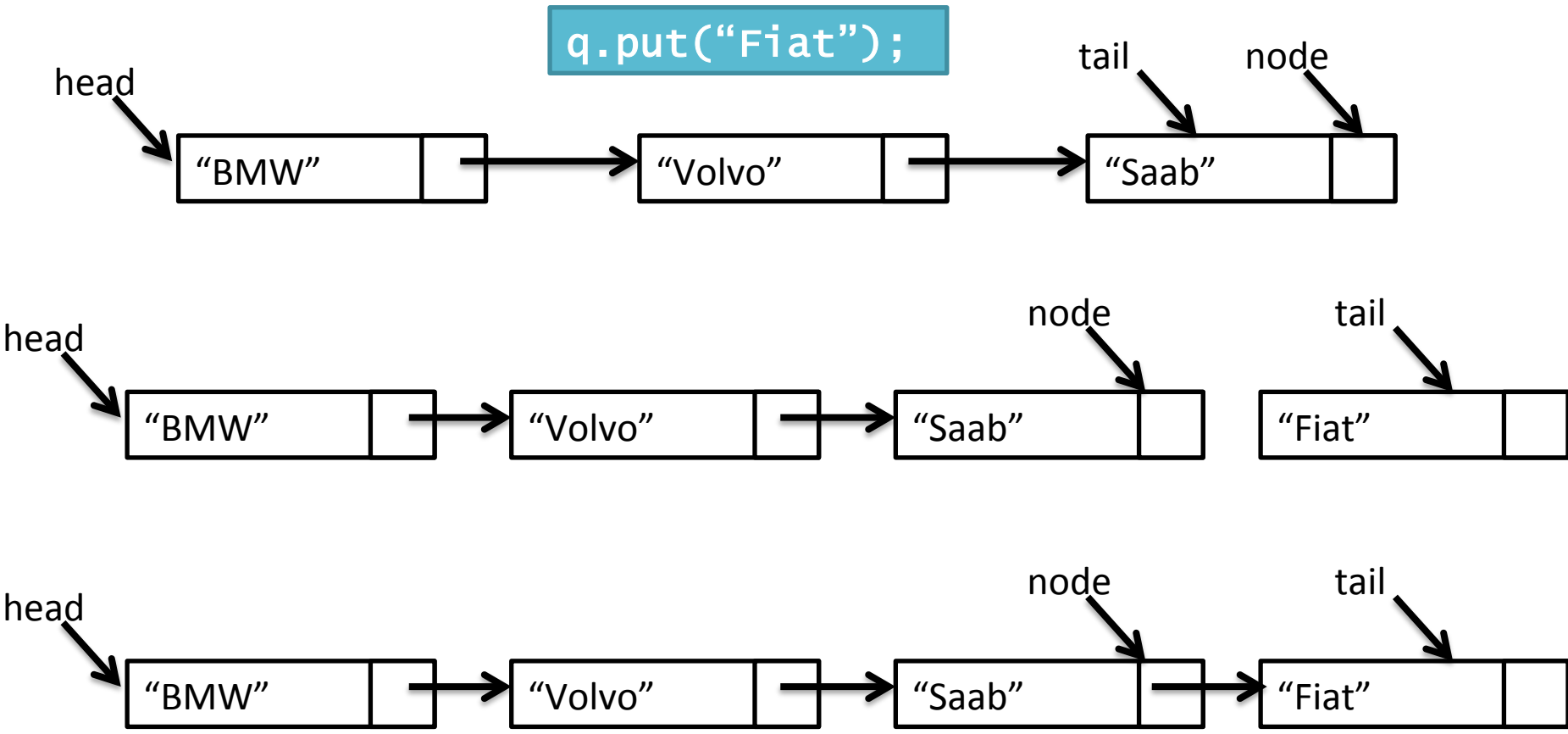
put

```
Queue.prototype.put = function(element){
    var node=this.tail;
    this.tail=new Node(element);
    if(this.isEmpty())
        this.head=this.tail;
    else
        node.next=this.tail;
}
```

get

```
Queue.prototype.get = function(){
    var val=this.head.value;
    this.head=this.head.next;
    return val;
}
```

Implementazione con LinkedList



Grafi

Grafo

Un grafo è una struttura matematica composta da elementi (**nodi**, **vertici**) connessi tra loro attraverso collegamenti (**lati**, **archi**). Formalmente...

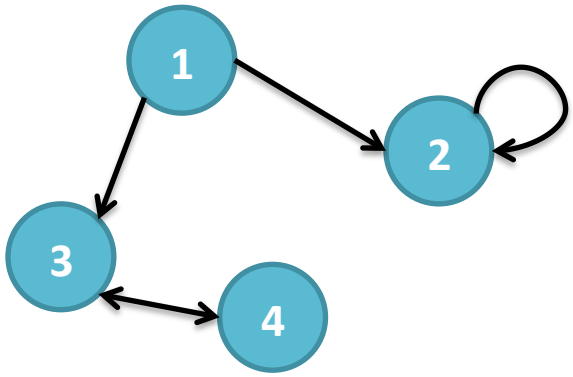
Grafo (diretto)

Un grafo diretto è una coppia $G=(V, E)$ in cui

- V è l'insieme dei vertici
- $E \subseteq V \times V$ è l'insieme degli archi, definito quindi come una relazione tra coppie di vertici t.c. $(v_1, v_2) \in E$ sse v_1 e v_2 sono collegati.

Nel grafo **diretto**, gli archi hanno una direzione specifica, infatti ***E* è una relazione** (il che implica $(v_1, v_2) \neq (v_2, v_1)$)

$V = \{1, 2, 3, 4\}$
$E = \{(1,2), (2,2), (1,3), (3,4), (4,3)\}$



Grafo diretto

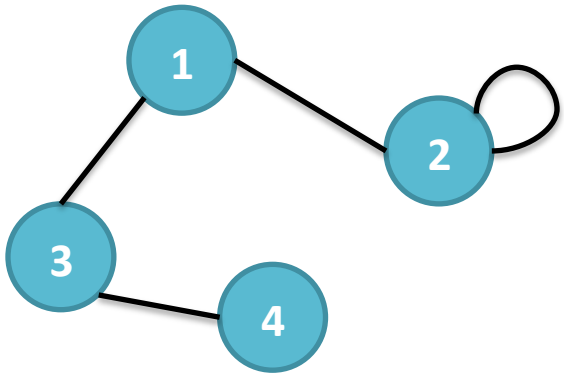
Grafo

Grafo (semplice)

Un grafo semplice è una coppia *G=(V, E)* in cui

- *V* è l'insieme dei vertici
- *E* è l'insieme degli archi, in cui ogni arco è un sottoinsieme a 2 elementi di *V* (coppia non ordinata). Quindi (i,j) in E sse (j,i) in E.

V = {1, 2, 3, 4}
E = {{1,2}, {2,2}, {1,3}, {3,4}}



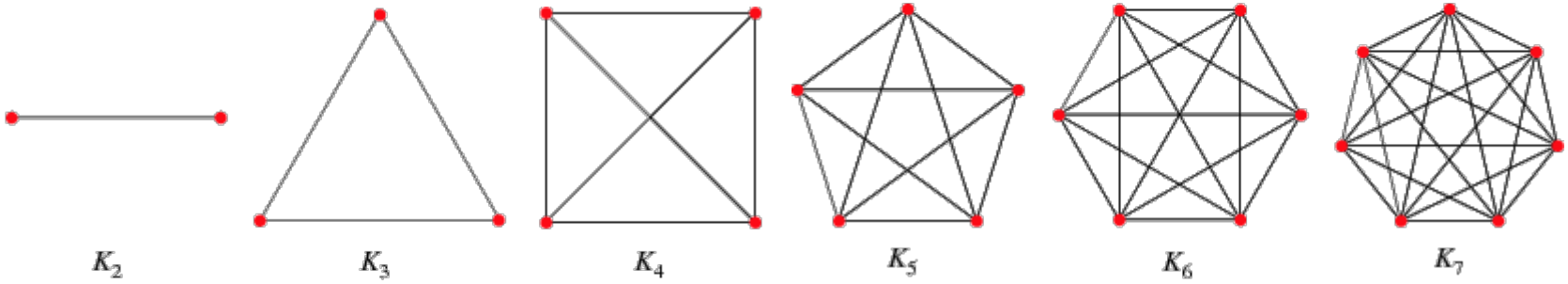
Grafo semplice

Glossario (1/2)

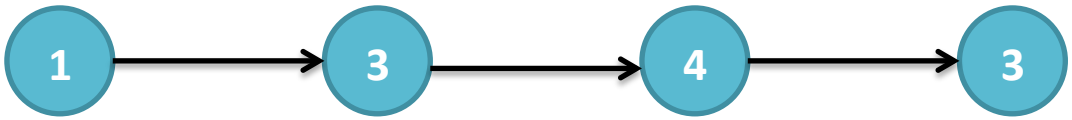
- **Grado di un vertice**: numero di archi incidenti il vertice dato

  $grado(v) = |\{(v1,v2) \in E: v1=v \text{ oppure } v2=v\}|$

- **Vertici adiacenti:** due vertici si dicono adiacenti se esiste un arco che li collega
- **Grafo completo (clique):** grafo in cui ogni coppia di vertici è collegata



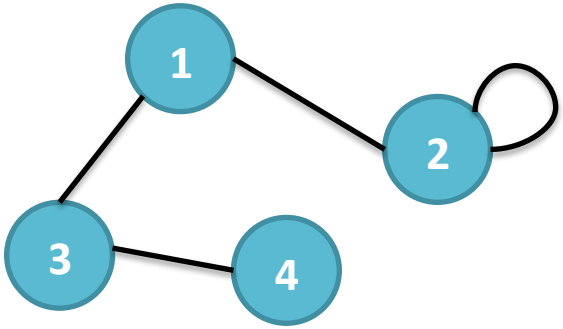
- **Percorso:** Un percorso di lunghezza *n* è dato da una sequenza di vertici $v_0, v_1, \ldots, v_n$ e da una sequenza di archi che li collegano $(v_0,v_1),(v_1,v_2),\ldots,(v_{n-1},v_n)$. I vertici v_0 e v_n si dicono *estremi* del percorso.
- **Cammino:** è un percorso in cui i lati sono a due a due distinti



Glossario (2/2)

- **Ciclo:** un cammino in cui gli estremi coincidono ($v_0 = v_n$)
- **Vertici connessi:** due vertici *u,v* si dicono connessi se esiste un cammino con estremi *u* e *v.* Altrimenti si dicono sconnessi.
- **Vertice isolato:** un vertice si dice isolato se non è connesso ad alcun altro vertice
- **Grafo connesso:** un grafo è connesso se ogni suo vertice è connesso con ogni altro suo vertice
- **Grafo denso:** è un grafo *(V, E)* in cui $|E| \cong |V|^2$
- **Grafo sparso:** è un grafo *(V, E)* in cui $|E| << |V|^2$

Implementazione grafo



Matrice di adiacenza

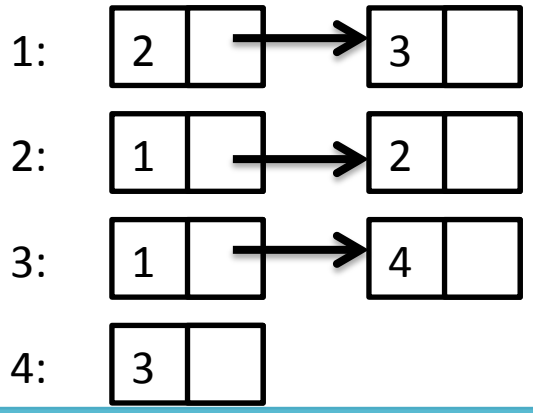
Una matrice quadrata di dimensione $|V|$, tale che $M_{ij} = 1$ se esiste l'arco (i, j); 0 altrimenti.

M =

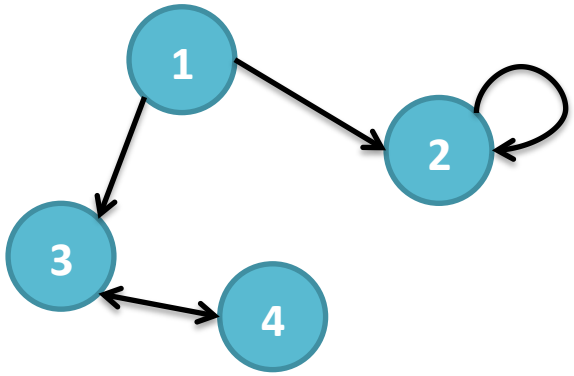
	1	2	3	4
1	0	**1**	**1**	0
2	**1**	**1**	0	0
3	**1**	0	0	**1**
4	0	0	**1**	0

Liste di adiacenza

Per ogni vertice è definita una LinkedList dei nodi adiacenti



Implementazione grafo



Matrice di adiacenza

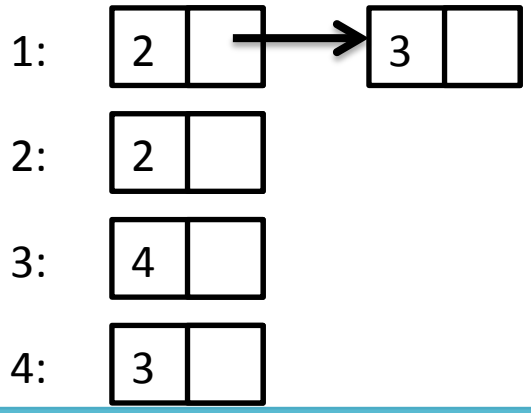
?

M =

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

Liste di adiacenza

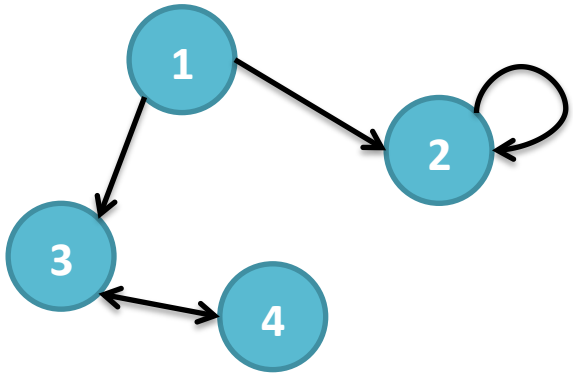
?



Implementazione grafo

Graph con matrice di adiacenza

```
function Graph(nNodes, isDir){
    this.isDir = isDir;
    this.m = new Array(nNodes);
    //inizialmente, grafo sconnesso
    for(var i=0; i<nNodes; i++){
        this.m[i]=new Array(nNodes);
        for(var j=0; j<nNodes; j++)
            this.m[i][j] = 0;
    }
}
Graph.prototype.setEdge =
function(i,j){
    this.m[i][j] = 1;
    if(!this.isDir)
        this.m[j][i] = 1;
}
```




```
var g = new Graph(4,true);
g.setEdge(1,3);
g.setEdge(3,4);
g.setEdge(4,3);
g.setEdge(1,2);
g.setEdge(2,2);
```

Implementazione grafo

Graph con liste di adiacenza

```
function Graph(nNodes, isDir){
    this.isDir = isDir;
    this.a = new Array(nNodes);
    for(var i=0; i<nNodes; i++)
        this.a[i]=new LinkedList();

}
Graph.prototype.setEdge = function(i,j){
//inserisco all'inizio della lista
    if(!this.a[i].contains(j))
        this.a[i].insertAt(j,0);
    if(!this.isDir && !this.a[j].contains(i))
        this.a[j].insertAt(i,0);
}
```

Esercizio 1

degree(v)

Restituisce il grado (numero di vertici adiacenti) del vertice v

Matrice di adiacenza

```
Graph.prototype.degree = function(v){
    var n=0;
    for(var i=0; i<this.m[v].length; i++)
        n+=this.m[v][i];
    return n;
}
```

Liste di adiacenza

```
Graph.prototype.degree = function(v){
    return this.a[v].size();
}
```

Esercizio 2

adjacents(v)

Restituisce i vertici adiacenti al vertice v (considerare solo il caso di implementazione con lista).

Liste di adiacenza

```
Graph.prototype.adjacents = function(v){
    return this.a[v];
}
```

Visita di grafi

Gli algoritmi di visita di grafi hanno l'obiettivo di accedere a tutti i vertici del grafo a partire da un dato vertice (detto **sorgente**), in modo

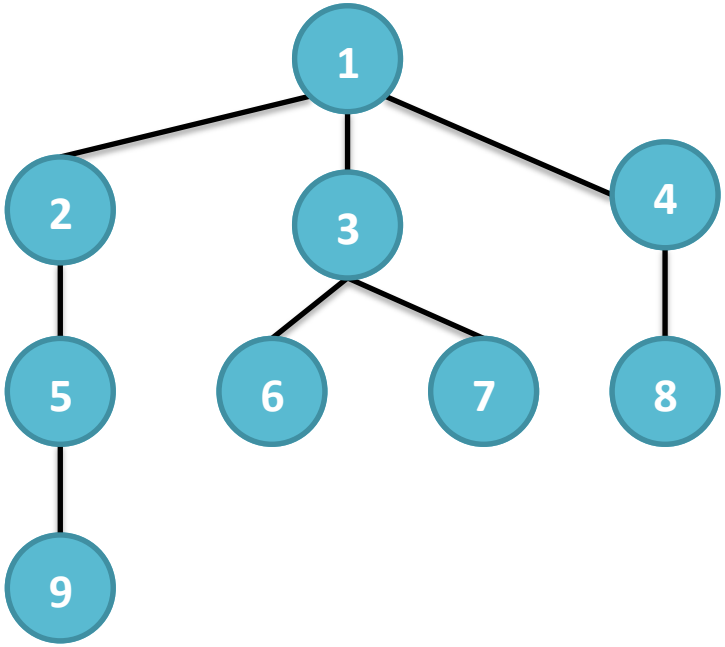
- **corretto** (tutti i vertici vanno visitati)
- **efficiente** (performante)

Due algoritmi principali:

- **BFS (Breadth-First Search):** è un algoritmo di visita in **ampiezza**
- **DFS (Depth-First Search):** è un algoritmo di visita in **profondità**

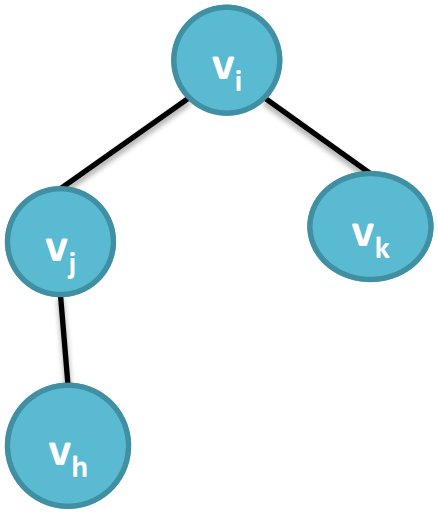
Breadth-First Search

Ordine di visita dei vertici



Breadth-First Search

La visita in ampiezza si serve di una **coda** per memorizzare ad ogni passo i vertici adiacenti (non ancora visitati) al vertice appena visitato. Ogni volta che il vertice viene visitato è estratto dalla coda.



1 – La coda contiene la sorgente v_i

2 – v_i viene visitato e vengono accodati i suoi adiacenti

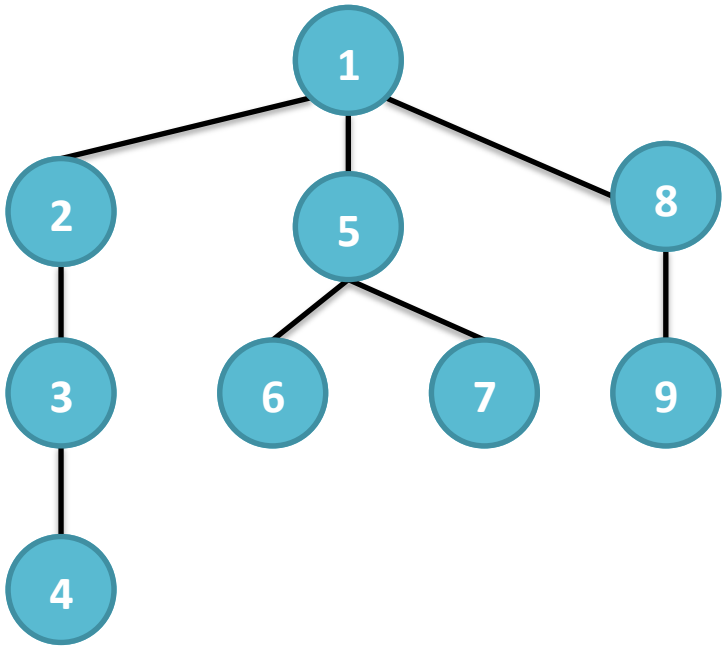
3 – v_j viene visitato e viene accodato il suo adiacente v_h

4 – v_k viene visitato (non ha adiacenti non visitati)

5 – v_h viene visitato (non ha adiacenti non visitati)

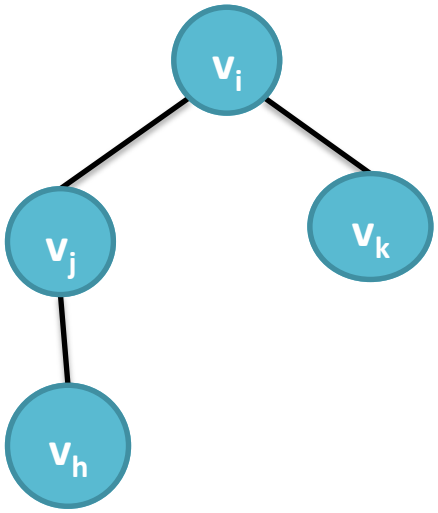
Depth-First Search

Ordine di visita dei vertici



Depth-First Search

La visita in profondità si serve di uno **stack** per memorizzare ad ogni passo i vertici adiacenti (non ancora visitati) al vertice appena visitato.



A differenza di BFS (che utilizza la coda), l'algoritmo esamina l'ultimo nodo inserito, dirigendo così la visita in profondità

1 – Lo stack contiene la sorgente v_i

2 – v_i viene visitato e vengono impilati i suoi adiacenti

3 – v_j viene visitato e viene impilato il suo adiacente v_h

4 – v_h viene visitato (non ha adiacenti non visitati)

5 – v_k viene visitato (non ha adiacenti non visitati)

Implementazione BFS

- Bisogna mantenere una struttura in cui memorizziamo i vertici visitati
- Supponiamo l'implementazione con liste di adiacenza

```
Graph.prototype.BFS = function (v){
    var q = new Queue();
    q.put(v);
    var visited = new Array();
    while(!q.isEmpty()){
        //prendo il primo elemento della coda
        var u = q.get();
        //lo visito
        visited[u]=true;
        //scorro tutti gli adiacenti di u
        for(var adj=u.adjacents().head; adj!
        =undefined; adj=adj.next)
            if(!visited[adj])
                q.put(adj);
    }
}
```

Implementazione DFS

```
Graph.prototype.DFS = function (v){
    var s = new Stack();
    s.push(v);
    var visited = new Array();
    while(!s.isEmpty()){
        var u = s.pop();
        visited[u]=true;
        for(var adj=u.adjacents().head; adj!
        =undefined; adj=adj.next)
            if(!visited[adj])
                s.push(adj);
    }
}
```

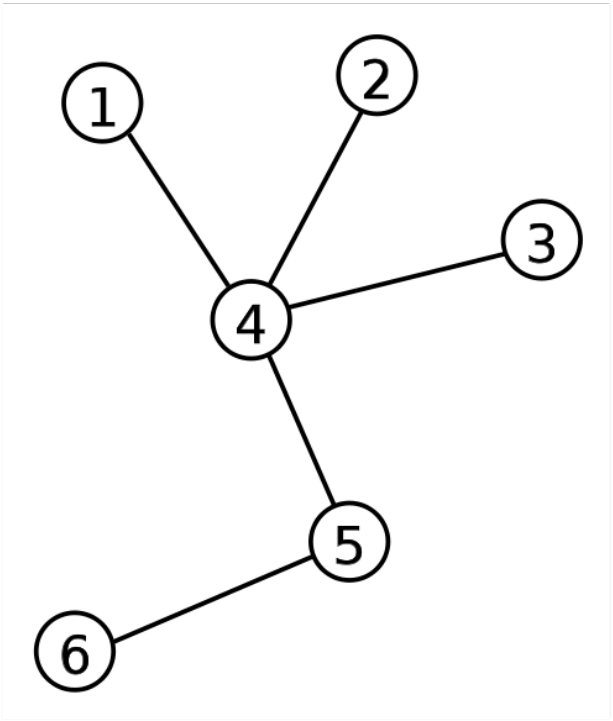
Alberi

Albero (teoria dei grafi)

Nella teoria dei grafi, un **albero** è un grafo G non diretto con le seguenti caratteristiche

I. **aciclico** (ovvero non ci sono cicli)

II. **connesso** (ovvero da ogni nodo è possibile raggiungere qualsiasi altro nodo)



I. e II. sono equivalenti alle seguenti:

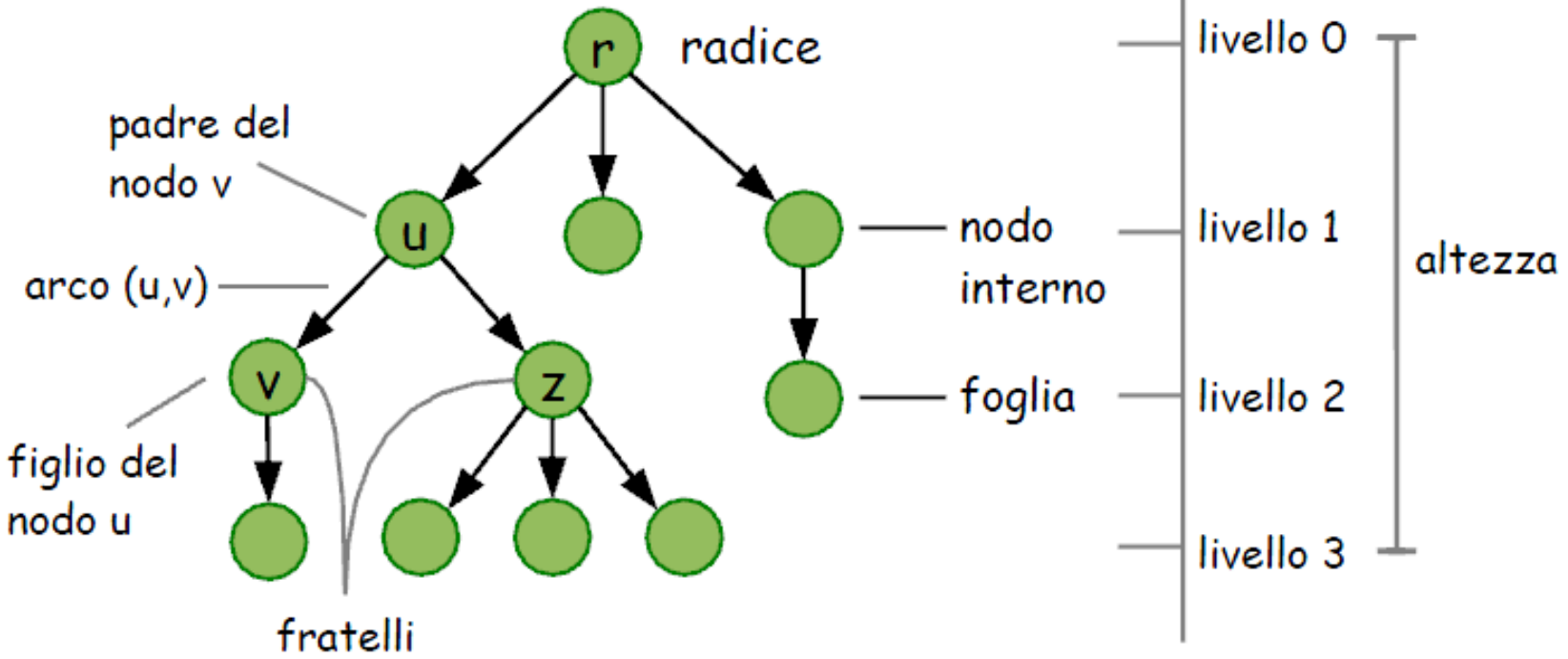
- G è aciclico, e l'aggiunta di qualsiasi arco genera un ciclo
- G è connesso, ma la rimozione di un qualsiasi arco non lo rende più connesso
- Due vertici qualsiasi sono connessi da un unico cammino
- G è connesso e $|E| = |V| - 1$
- G è aciclico e $|E| = |V| - 1$

Albero (struttura dati)

In informatica, il concetto di albero viene utilizzato per **l'organizzazione gerarchica** dei dati.

• Viene considerato diretto, e gli archi stabiliscono una relazione di padre-figlio tra nodi, quindi $(u,v) \in E$ significa che *v* è **figlio** di *u,* o equivalentemente che u è **padre** di v. Nodi con lo stesso padre sono detti **fratelli.**
• quindi il grado di un nodo è dato dal numero di suoi figli
• Esiste un unico nodo senza padre, detto **radice.** Ogni altro nodo possiede un solo padre.
• un nodo senza figli è detto **foglia**, mentre i nodi che non sono nè foglie nè la radice sono detti **nodi interni**
• la **profondità** (o **livello**) di un nodo è dato dal numero di archi che bisogna attraversare dalla radice per raggiungerlo
• **altezza** di un albero: massima profondità a cui si trova una foglia

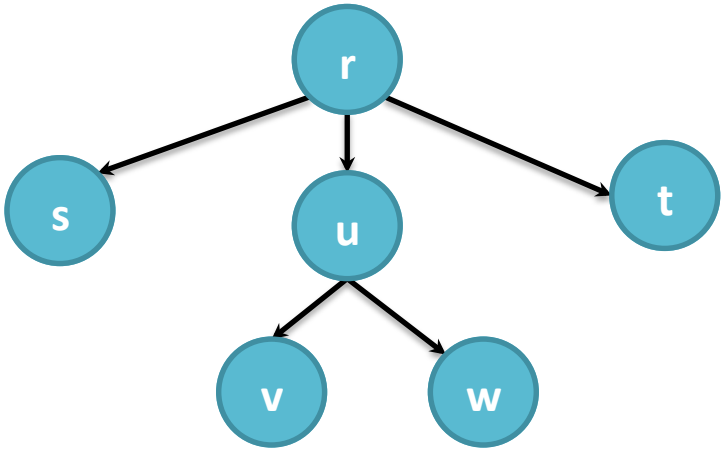
Albero (struttura dati)

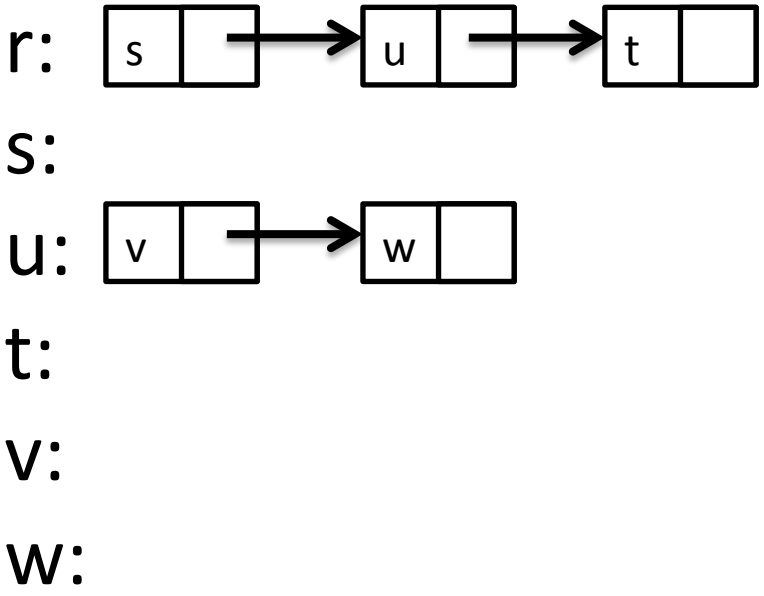


- **Antenato** di un nodo *u*: il padre di *u* o un antenato del padre di *u*
- **Discendente** di un nodo u: un figlio di *u* o un discendente di uno dei figli di *u*

Realizzazione con Liste

- Esattamente come per i grafi, è possibile realizzare un albero servendosi delle liste concatenate.
- In grafi generici, ad ogni nodo era associata la lista dei nodi adiacenti
- Nell'albero, gli adiacenti ad un nodo sono proprio i figli del nodo stesso (gli archi sono diretti da padre in figlio)

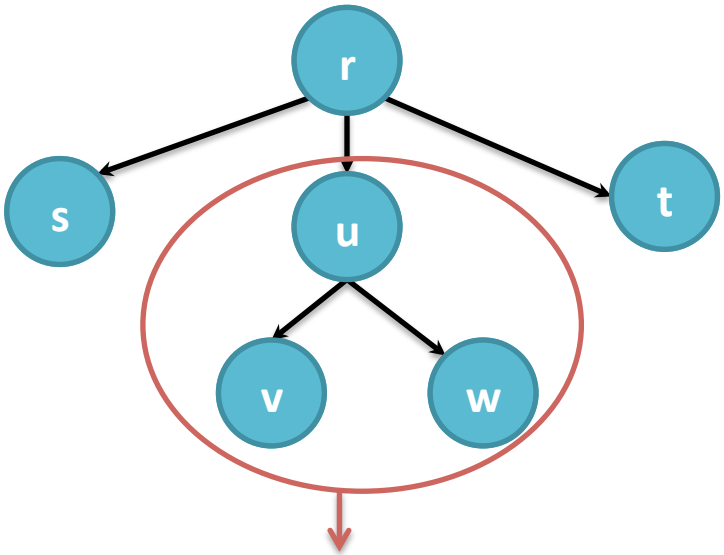




Visite di alberi

• Gli algoritmi di visita in profondità e in ampiezza funzionano in modo equivalente

• Per visitare tutti i nodi di un albero si parte dalla radice. Partendo da un nodo generico *u*, l'algoritmo visita il sottoalbero con radice *u*

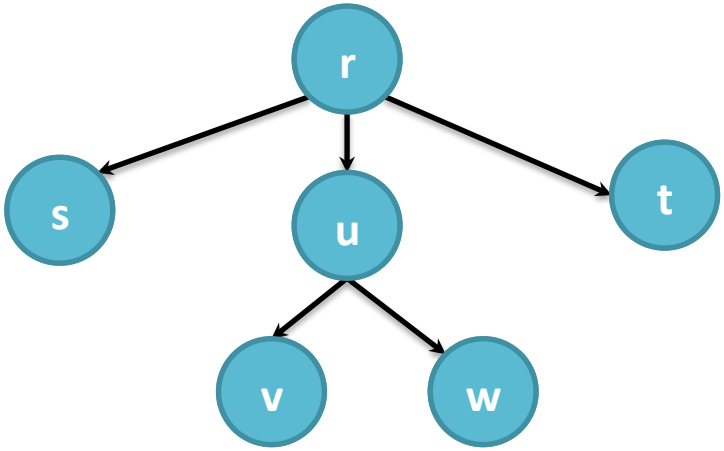


Sottoalbero con radice *u*

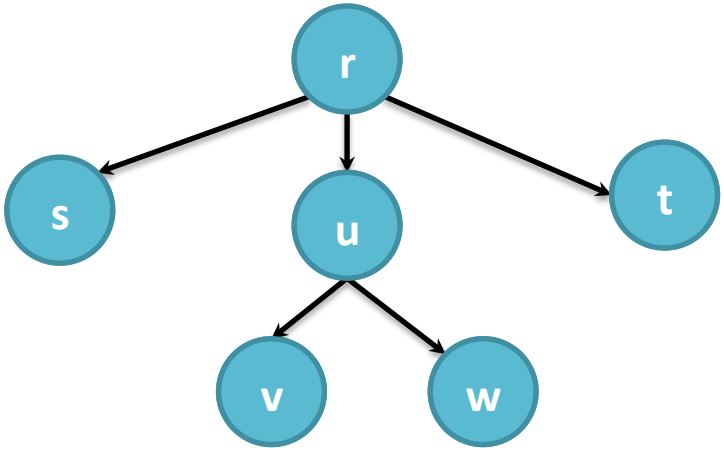
• BFS visita in ampiezza, quindi nodi al livello successivo vengono visitati quando tutti quelli al livello corrente sono stati visitati

• DFS visita in profondità, quindi procede visitando nodi di padre in figlio, fino a raggiungere una foglia, e poi retrocede al primo antenato che ha figli non ancora visitati

Visite di alberi

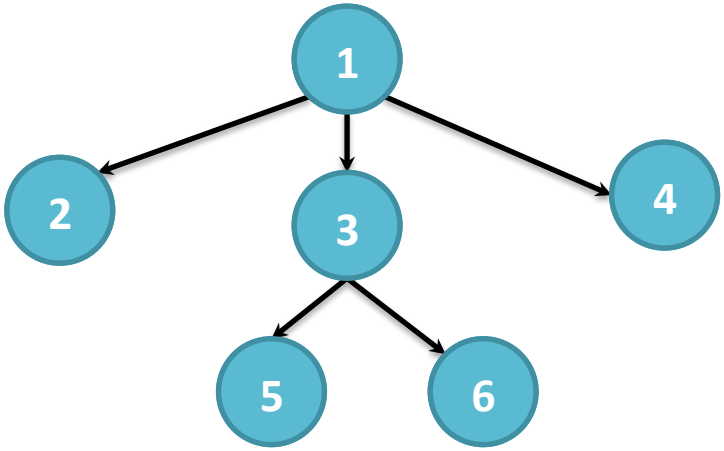


BFS?

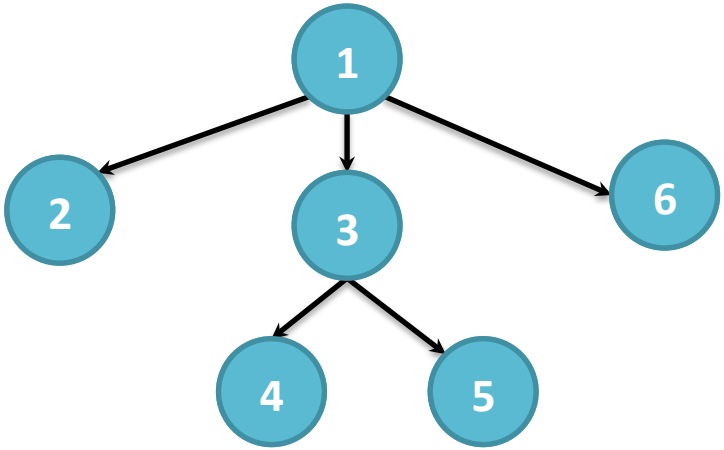


DFS?

Visite di alberi



BFS?

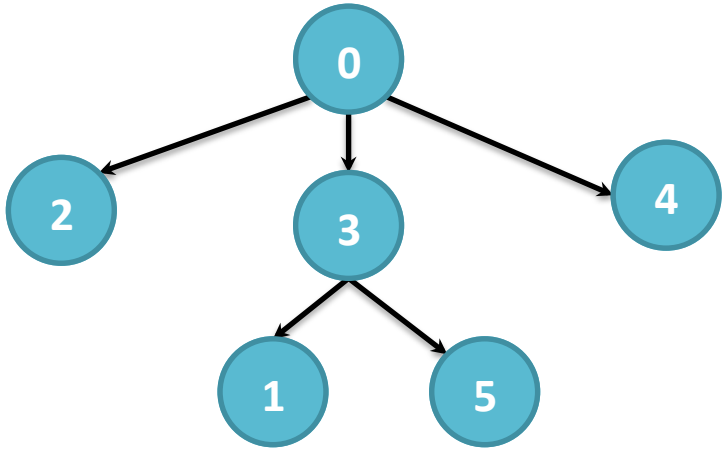


DFS?

Realizzazione con Array dei padri

- Supponendo che gli *n* nodi dell'albero siano etichettati da *0* a *n-1*, si realizza un array T di dimensione *n* (che contiene quindi una posizione per ogni nodo), in cui ad ogni posizione *i* è memorizzato il padre del nodo *i*.
- Ricordando che la radice è l'unico nodo senza padre...

`T[i] = -1`, se `i` è la radice
`T[i] = j`, se `j` è padre di `i`



T

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

Realizzazione con array dei padri

Tale implementazione assicura che ogni nodo ha al più un padre

Tree (array dei padri)

```
function Tree(numNodes, root){
    this.fathers=new Array(numNodes);
    this.root=root;
    this.fathers[root]=-1;
}
// metodo per inserire l'arco i->j
Tree.prototype.setEdge = function(i,j){
    if(j!=this.root)
        this.fathers[j] = i;
}
```

Esercizio

getChildren(u)

Restituisce un Array contenente tutti i nodi figlio del nodo u

```
Tree.prototype.getChildren = function(u){
    var sons = new Array();
    for(var i=0; i<this.fathers.length; i++)
        if(this.fathers[i]==u)
            sons[sons.length]=i;
    return sons;
}
```

getBrothers(u)

Restituisce un Array contenente tutti i nodi fratelli del nodo u (incluso)

```
Tree.prototype.getBrothers = function(u){
    return this.getChildren(this.fathers[u]);
}
```

Come verifico che un oggetto* `Tree` *è connesso?

Si effettua una visita a partire dalla radice, e i nodi visitati saranno tutti i nodi dell'albero

```
Tree.prototype.isConnected = function(){
    //inizio BFS
    var q = new Queue();
    q.put(root);
    var visited = new Array(this.fathers.length);
    while(!q.isEmpty()){
        var u = q.get();
        visited[u]=true;
        var sonsU = this.getChildren(u);
        for(var i=0; i<sonsU.length; i++)
                if(!visited[i])
                    q.put(i);
    }
    //fine BFS, verifico se tutti i nodi sono visitati
    var result=true;
    for(var i=0; i<visited.length; i++)
        result=result&&visited[i];
    return result;
}
```

Come verifico che un oggetto* `Tree` *è aciclico?

Si effettua una visita a partire dalla radice, e i nodi figli del nodo appena visitato non devo essere già stati visitati

```
Tree.prototype.isAcyclic = function(){
    //inizio BFS
    var q = new Queue();
    q.put(root);
    var visited = new Array(this.fathers.length);
    while(!q.isEmpty()){
        var u = q.get();
        visited[u]=true;
        var sonsU = this.getChildren(u);
        for(var i=0; i<sonsU.length; i++)
                //se un figlio è già visitato, ciclico
                if(visited[i])
                    return false;
                else
                    q.put(i);
    }
    return true;
}
```

Come verifico che un oggetto* Tree *è effettivamente un albero (connesso e aciclico)?

```
Tree.prototype.isTree = function(){
    var q = new Queue();
    q.put(root);
    var visited = new Array(this.fathers.length);
    while(!q.isEmpty()){
        var u = q.get();
        visited[u]=true;
        var sonsU = this.getChildren(u);
        for(var i=0; i<sonsU.length; i++)
                if(visited[i])
                    return false;
                else
                    q.put(i);
    }
    var result=true;
    for(var i=0; i<visited.length; i++)
        result=result&&visited[i];
    return result;
}
```

?

Esempi di stack e code:
Ovvero quando utilizzo l'uno o l'altro