



# Strutture dati

Size

Big topic

Funzioni, Array, Oggetti, Insiemi, Alberi

## ▼ Array

Gli array sono una struttura dati formata da più celle (o elementi)

In generale:

- ogni cella si comporta come una variabile tradizionale
- tutte le celle sono variabili di uno stesso tipo preesistente (detto tipo base dell'array)
- il numero di celle è prefissato e denota la dimensione dell'array
- ogni cella è identificata da un valore di indice gli indici vanno da 0 a N-1 (con N dimensione dell'array)



La possibilità di accedere ad un elemento grazie al proprio indice è la principale caratteristica dell'array

Esiste una serie di funzioni speciali che semplificano le operazioni su array

- `A.push()` – Aggiunge un elemento in coda a un array, restituisce la nuova lunghezza
- `A.pop()` – Estrae l'ultimo elemento dall'array e lo restituisce
- `A.shift()` – Estrae il primo elemento dall'array e lo restituisce
- `A.unshift()` - Aggiunge un elemento in testa all'array, restituisce la nuova lunghezza
- `A.concat(B)` – Concatena l'array B in coda all'array A, restituisce il risultato
  - $A = [3,4,5], B = [7,9] \Rightarrow A.concat(B) = [3,4,5,7,9]$

Esempi di funzioni speciali su un array  $A = [1, 2, 3, 4]$

<code>A.push(10)</code> → 5		$A \rightarrow [1, 2, 3, 4, 10]$
<code>A.pop()</code> → 4		$A \rightarrow [1, 2, 3]$
<code>A.shift()</code> → 1		$A \rightarrow [2, 3, 4]$
<code>A.unshift(10)</code> → 5		$A \rightarrow [10, 1, 2, 3, 4]$

Tutti gli array hanno una proprietà implicita `length` che restituisce la loro lunghezza corrente (elementi fra 0 e il massimo indice definito)

- `delete` cancella un elemento (diventa `undefined`) ma non cambia la lunghezza
- assegnare un valore a `length` tronca o estende un array

Gli array in Javascript sono molto più flessibili rispetto al semplice accesso con indice. Un array si dice omogeneo se tutti i suoi elementi hanno lo stesso tipo

#### Array disomogenei

```
[ 1, 2, "pippo", 3.1415, {x: 10, y:12}, 3 ] [ 1, 2, <empty>×4, 7 ]
```

#### Array sparsi

#### Array con chiavi non numeriche

```
var a=[1, 2]; a.pippo=3;  
a → [1, 2, pippo: 3] a.length
```

## Array come tuple

Una tuple è una collezione di **elementi ordinati** e **immutabili**. Le tuple sono simili agli array in quanto archiviano una collezione di elementi, ma a differenza degli array, gli elementi non possono essere modificati una volta creati.

Gli array possono essere usati come **tuple**, fissando la loro lunghezza.

Ad esempio, la tuple  $\langle 4, 1 \rangle$  può essere rappresentata dall'array  $[4, 1]$ .



Se ho una tuple  $t = [4, 1]$ , posso fare:

- Accedere agli elementi:  $t[0] \rightarrow 4$        $t[1] \rightarrow 1$
- Assegnare l'intera tuple:  $q = t$ ;       $q \rightarrow [4, 1]$
- Assegnare separatamente gli elementi:  $[a, b] = t$ ;       $a \rightarrow 4$        $b \rightarrow 1$
- Chiamare una funzione passando la tuple:  $f(t)$
- Restituire più risultati da una funzione: `return t`;

**Assegnare separatamente gli elementi = Assegnamento destrutturante**

## Array come liste

Gli array possono essere usati anche come **liste**, ovvero come **sequenze** di elementi di lunghezza variabile.

In questo stile, raramente si accede agli elementi tramite la loro posizione (*per indice*). Invece, si lavora in maniera **strutturale**:

- Una **lista vuota** è una lista: `[]`
- Un **elemento seguito da una lista** è una lista: `[testa resto]`, **idem se una lista è seguita da un elemento** `[resto coda]`

**testa** → elemento  
**coda** → elemento  
**resto** → lista

Si può fare un assegnamento destrutturante con *spread* anche sulle liste:

- $[\text{testa}, \dots \text{resto}] = [1, 2, 3, 4]$  **testa** → 1    **resto** →  $[2, 3, 4]$
- $[\text{testa}, \dots \text{resto}] = [1]$                       **testa** → 1    **resto** →  $[]$
- $[\text{testa}, \dots \text{resto}] = []$                       **testa** → *undefined*    **resto** →  $[]$

Ciò consente di scrivere facilmente algoritmi ricorsivi sulle liste, es. *numeriche*

```
function len(a) {
  let [t, ...r] = a
  return t?1+len(r):0
}
```

```
function sum(a) {
  let [t, ...r] = a
  return t?t+sum(r):0
}
```

```
function max(a) {
  let [t, ...r] = a
  if (t)
    return Math.max(t,max(r))
  else
    return -Infinity
}
```

## Metodi predefiniti per scorrere array come liste

Siano  $A$  un array di elementi di tipo  $E$ ,  $p : E \rightarrow \text{Bool}$  un predicato,  $f : E \rightarrow X$  una funzione

- **`A.every(p)`** Restituisce *true* se **tutti** gli elementi di  $A$  soddisfano il predicato **p**, **false** altrimenti
- **`A.some(p)`** Restituisce *true* se **almeno uno** degli elementi di  $A$  soddisfa il predicato  $p$ , **false** altrimenti
- **`A.find(p)`** Restituisce un elemento **e** di  $A$  tale che  $p(e)$  è *true*, o *undefined* se nessun **e** soddisfa  $p$
- **`A.findIndex(p)`** Restituisce l'indice di un elemento **e** di  $A$  tale che  $p(e)$  è *true*, o  $-1$  se nessun **e** soddisfa  $p$
- **`A.includes(e)`** Restituisce *true* se  $A$  contiene un elemento uguale a **e**, **false** altrimenti

- `A.forEach(f)` Invoca  $f(e)$  per ogni elemento  $e$  dell'array  $A$
- `A.map(f)` Restituisce un nuovo array  $[f(e_1), f(e_2), \dots, f(e_n)]$
- `A.filter(p)` Restituisce un nuovo array  $A'$  contenente i soli elementi  $e$  di  $A$  che soddisfano  $p$ , in ordine

A volte sono utili delle funzioni di riduzione, ovvero:

Partire da un valore dato che costituisce il risultato iniziale, poi si scorre la lista, un elemento alla volta, e si applica una funzione data al risultato precedente e al nuovo elemento. Alla fine della lista, si restituisce il risultato.

- `A.reduce(f, z)` Fa una riduzione da sinistra a destra, partendo da  $z$ , e applicando  $f$
- `A.reduceRight(f, z)` Fa una riduzione da destra a sinistra, partendo da  $z$ , e applicando  $f$

## Assegnamenti destrutturanti

Il concetto di *destrutturazione* è semplice: si prende un **tipo strutturato** (array, oggetto) e lo si “spacchetta” andando a guardare al suo interno.

In molti casi in cui normalmente è previsto un **nome di variabile** (teoricamente: una *left-handside* o *lhs*, ciò che può stare a sinistra di un  $=$ ), JavaScript consente di usare la notazione di un **letterale** array o oggetto, in cui però al posto dei valori si indicano dei **nomi di variabili** (dei *lhs*).

*Notate che anche questa definizione è ricorsiva: si possono destrutturare array o oggetti che contengono altri array o oggetti come elementi*

Alcuni esempi di array destrutturati

Se  $A = [4, 7, 1]$

- $[a,b] = A$                        $a \rightarrow 4$      $b \rightarrow 7$
- $[a,b,c,d] = A$                        $a \rightarrow 4$      $b \rightarrow 7$      $c \rightarrow 1$      $d \rightarrow \text{undefined}$
- $[a,b,c=3,d=8] = A$                  $a \rightarrow 4$      $b \rightarrow 7$      $c \rightarrow 1$      $d \rightarrow 8$
- $[,c] = A$                              $c \rightarrow 1$

- $[a, \dots, r] = A$        $a \rightarrow 4 \quad r \rightarrow [7, 1]$
- $[y, x] = [x, y]$       scambia i valori di  $x, y$  // senza variabili intermedie
- $[x, y] = f("pippo", 3)$       ottiene due valori di ritorno distinti da una funzione

## Assegnamenti destrutturanti

La destrutturazione sugli oggetti consente anche di avere funzioni con molti parametri, non tutti obbligatori;  
il chiamante indica **per nome** anziché **per posizione** quelli che vuole indicare.

```
function disegna(x, y, { raggio=0, colore="nero", bordo=1, etichetta="" } ) {
  /* codice di disegno; usa x,y, raggio, colore, bordo, etichetta */
}
```

E il chiamante può di volta in volta decidere quali caratteristiche vuole specificare

```
disegna(5,8, {colore="blu"})    disegna(3, 2)    disegna(5,8,{colore="rosso", bordo=2})disegna(0,1,sti
```

## Operatore di spread

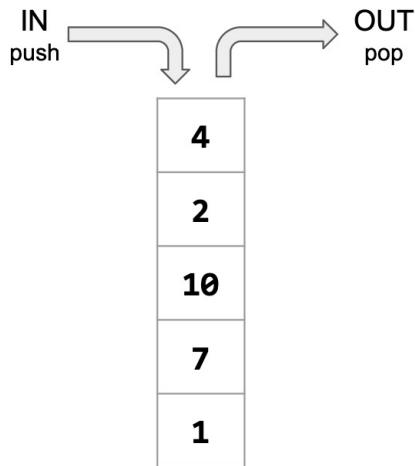
Consente di espandere una serie di elementi in un array o in un oggetto. Ciò significa che può essere utilizzato per unire due array, copiare un array o un oggetto, o per passare più argomenti a una funzione invece di passare un array.

- $f(\dots, A)$       chiama la funzione  $f$ , passando gli elementi di  $A$  come argomenti singoli
- $p = \dots, q, c: 3$        $p$  è una copia dell'oggetto  $q$ , con la chiave  $c$  che vale 3
- $p = \dots, c: 3, \dots, q$        $p$  è una copia di  $q$ , con aggiunto  $c = 3$  se manca in  $q$ , altrimenti il valore di  $c$  in  $q$
- $B = [1, 2, \dots, A, 6]$        $B$  è un array contenente gli elementi di  $A$ , ma con 1 e 2 davanti e 6 in coda
- $C = [\dots, A, \dots, B]$        $C$  è la concatenazione degli array  $A$  e  $B$
- $R = \dots, O1, \dots, O2$        $R$  è l'unione dei due oggetti  $O1$  e  $O2$ ; se hanno delle chiavi in comune, “vince” il valore di  $O2$
- $[testa, \dots, resto] = A$        $testa$  prende il primo elemento di  $A$ ,  $resto$  tutti gli altri

## Pila



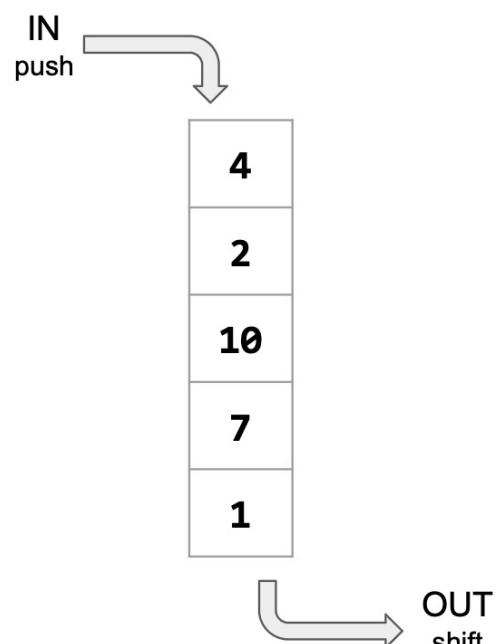
**LIFO (Last In, First Out)**



## Coda



**FIFO (First In, First Out)**



Per realizzare una pila con un array A:

Crescita in Coda:

- `A.push()` per inserire sulla pila
- `A.pop()` per estrarre dalla pila

Crescita in Testa:

- `A.unshift()` per inserire sulla pila
- `A.shift()` per estrarre dalla pila

Per realizzare una coda con un array A:

- `A.push()` per aggiungere in coda
- `A.shift()` per estrarre dalla testa
- `A.unshift()` per aggiungere in testa
- `A.pop()` per estrarre dalla coda



Per convenzione, si può usare `push/pop` per le pile, `push/shift` per le code

## Metodi predefiniti per scorrere array come liste

<code>A.every(p)</code>	Restituisce true se <b>tutti</b> gli elementi di <b>A</b> soddisfano il predicato <b>p</b> , <b>false</b> altrimenti
<code>A.some(p)</code>	Restituisce true se <b>almeno uno</b> degli elementi di <b>A</b> soddisfa il predicato <b>p</b> , <b>false</b> altrimenti
<code>A.find(p)</code>	Restituisce un elemento <b>e</b> di <b>A</b> tale che <b>p(e)</b> è <b>true</b> , o <b>undefined</b> se nessun <b>e</b> soddisfa <b>p</b>
<code>A.findIndex(p)</code>	Restituisce l'indice di un elemento <b>e</b> di <b>A</b> tale che <b>p(e)</b> è <b>true</b> , o <b>-1</b> se nessun <b>e</b> soddisfa <b>p</b>
<code>A.includes(e)</code>	Restituisce <b>true</b> se <b>A</b> contiene un elemento uguale a <b>e</b> , <b>false</b> altrimenti
<code>A.forEach(f)</code>	Invoca <b>f(e)</b> per ogni elemento <b>e</b> dell'array <b>A</b>
<code>A.map(f)</code>	Restituisce un nuovo array <b>[f(e1), f(e2), ...f(en)]</b>
<code>A.filter(p)</code>	Restituisce un nuovo array <b>A'</b> contenente i soli elementi <b>e</b> di <b>A</b> che soddisfano <b>p</b> , in ordine

Siano **A** un array di elementi di tipo **E**, **p : E → Boolean** un predicato **f : E → X** una funzione

## Funzioni di riduzione

Partendo da un valore dato che costituisce il risultato iniziale, poi si scorre la lista, un elemento alla volta, e si applica una funzione data al risultato precedente e al nuovo elemento. Alla fine della lista, si restituisce il risultato.

<code>A.reduce(f, z)</code>	Fa una riduzione da sinistra a destra, partendo da <code>z</code> , e applicando <code>f</code>
<code>A.reduceRight(f, z)</code>	Fa una riduzione da destra a sinistra, partendo da <code>z</code> , e applicando <code>f</code>

## Assegnamenti destrutturati

Prendendo un **tipo strutturato** (array, oggetto) e lo si “spacchetta” andando a guardare al suo interno.

In molti casi in cui normalmente è previsto un **nome di variabile**, JavaScript consente di usare la notazione di un **letterale** array o oggetto, in cui però al posto dei valori si indicano dei **nomi di variabili**.

## Assegnamenti destrutturanti su array

Se `A = [4,7,1]`

- `[a, b] = A`       $a \rightarrow 4 \quad b \rightarrow 7$
- `[a, b, c, d] = A`       $a \rightarrow 4 \quad b \rightarrow 7 \quad c \rightarrow 1 \quad d \rightarrow \text{undefined}$
- `[a, b, c=3, d=8] = A`       $a \rightarrow 4 \quad b \rightarrow 7 \quad c \rightarrow 1 \quad d \rightarrow 8$
- `[, , c] = A`       $c \rightarrow 1$
- `[a, ...r] = A`       $a \rightarrow 4 \quad r \rightarrow [7, 1]$
- `[y, x] = [x, y]`      *scambia i valori di x, y senza variabili intermedie*
- `[x, y] = f("pippo", 3)`      *ottiene due valori di ritorno distinti da una funzione*

## Algoritmi di ordinamento per confronti

Sono algoritmi che effettuano l'ordinamento utilizzando solo confronti tra gli elementi di input, senza fare uso di altre primitive (operazioni aritmetiche, logiche, o altro)

L'operazione dominante è il confronto tra elementi

- il costo in tempo è proporzionale al numero di confronti effettuati
- si prende come unità di misura della complessità il numero di confronti

## Alcuni algoritmi per l'ordinamento su array:

### Selection Sort

```
function selectionSort(a) {
    for (let i=0; i < a.length-1; i++) {
        let min = a[i]
        let minIndex = i

        for (let j=i+1; j < a.length; j++) {
            if (a[j] < min) {
                min = a[j]
                minIndex = j
            }
        }

        let tmp = a[i]

        a[i] = a[minIndex]
        a[minIndex] = tmp
    }
}
```



#### COMPLESSITÀ

Caso ottimo:  $O(n^2)$

Caso medio:  $O(n^2)$

Caso peggiore:  $O(n^2)$

Complessità spaziale:  $O(1)$

## Insertion Sort

```
function insertionSort(a) {  
    for (j = 1; j < a.length; j++) {  
        key = a[j];  
        i = j - 1;  
        while (i>-1 && a[i] > key) {  
            a[i+1] = a[i];  
            i = i - 1;  
        }  
        a[i+1] = key;  
    }  
}
```



### COMPLESSITÀ

Caso ottimo:  $O(n)$

Caso medio:  $O(n^2)$

Caso peggiore:  $O(n^2)$

Complessità spaziale:  $O(1)$

## Quick Sort

```
function quickSortCormen0(a,p,r) {  
    if (p < r) {  
        q = partitionCormen(a,p,r);  
        quickSortCormen0(a,p,q-1);  
        quickSortCormen0(a,q+1,r);  
    }  
}  
  
function partitionCormen(a,p,r) {  
    x = a[r];  
    i = p - 1;  
    for (j = p; j < r; j++) {  
        if (a[j] <= x) {  
            i = i + 1;  
        }  
    }  
    a[i] = a[j];  
    a[j] = x;  
    return i;  
}
```

```

        temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
temp = a[i+1];
a[i+1] = a[r];
a[r] = temp;
return i+1;
}

```



## COMPLESSITÀ

Caso ottimo:  $O(n \log n)$

Caso medio:  $O(n \log n)$

Caso peggiore:  $O(n^2)$

Complessità spaziale:  $O(n)$

## Merge Sort

```

function merge(a1, a2){
    let result = [];
    let i = 0;
    let j = 0;

    while(i < a1.length && j < a2.length){
        if(a1[i] > a2[j]) {
            result.push(a2[j]);
            j++;
        } else {
            result.push(a1[i]);
            i++;
        }
    }

    while(i < a1.length){
        result.push(a1[i]);
        i++;
    }
}

```

```

    }

    while(j < a2.length){
        result.push(a2[j]);
        j++;
    }

    return result;
}

```



## COMPLESSITÀ

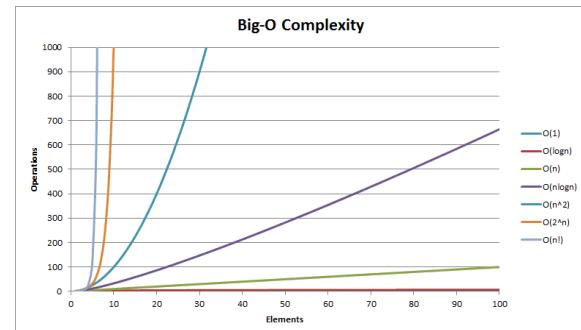
Caso ottimo:  $O(n \log n)$

Caso medio:  $O(n \log n)$

Caso peggiore:  $O(n \log n)$

Complessità spaziale:  $O(n)$

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$
Quick Sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(n)$
Heap Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n + k)$
Bucket Sort	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n)$



## ▼ Oggetti

Un oggetto (o dizionario) è una mappa da chiavi a valori. Potremmo anche dire che un oggetto è un insieme di coppie (chiave, valore).

*Gli oggetti consentono quindi di raccogliere in un solo valore (di tipo oggetto) molti valori diversi (di qualunque tipo), assegnando a ciascuno di essi un nome (la chiave). Ad esempio:*

```
pippo = {  
    passione: "Fondamenti"  
    età: 35  
    capelli: true  
}
```

Alcuni esempi di letterali di tipo oggetto

{ } - oggetto vuoto

{ nome: "pippo" , età: "35" } - il più comune { Nome: Valore }

{ x } espansione di variabile, ad esempio, se x vale 5, è come { x: 5 }

{ "nome": "Pippo", "età" : 35 } – le chiavi possono essere stringhe

{ "!!": 5, "😎": true } – anche stringhe che non sono identificatori

{ 0: 6, 1: 4, 2: 12 } – le chiavi possono essere numeri

{ ["pip"+f]: 1 } – chiave calcolata: se f vale "po", è come { pippo: 1 }

```
let pippo = { passione: "Fondamenti" , età: 35 , capelli: true}  
  
console.log (pippo.passione)  
  
console.log (pippo ["passione"])  
  
let prop = "Passione"  
console.log (...)
```

L'operazione principale sugli oggetti è l'accesso a un membro (chiave)

*!!! da mettere a posto !!!*

- == verifica se gli indirizzi sono uguali
- con this cambi lo stato dell'oggetto

## Operazioni sugli oggetti:

L'operazione principale sugli oggetti è l'accesso a un membro (chiave)

Altre operazioni utili:

- Controllo se una chiave esiste: chiave in oggetto
- Cancellazione di una coppia chiave-valore: delete chiave
- Scorrere le chiavi: for (k in oggetto)

## Assegnamenti destrutturati

Prendendo un **tipo strutturato** (array, oggetto) e lo si "spacchetta" andando a guardare al suo interno.

In molti casi in cui normalmente è previsto un **nome di variabile**, JavaScript consente di usare la notazione di un **letterale** array o oggetto, in cui però al posto dei valori si indicano dei **nomi di variabili**.

## Proprietà di questo tipo di funzione

In genere tutte le funzioni che usiamo, tipo `console.log()` , `Math.max()` non sono altro che proprietà di questi oggetti, con valori di tipo funzione

*Naturalmente, anche gli oggetti definiti da noi possono avere proprietà di tipo funzione.*

JavaScript prevede che nel corpo di queste funzioni si possa usare una variabile speciale che contiene un riferimento all'oggetto di cui la funzione è proprietà: `this`



*Quando una funzione è fornita come proprietà di un oggetto, e al suo interno riferisce l'oggetto, si usa chiamarla **metodo** dell'oggetto*

In questo modo, possiamo scrivere:

```
persona.compleanno = ()=>{this.età++}
```

e poi per invocare `compleanno()`

```
pippo.compleanno();
```

## Metodi e costrutti

È utile assicurarsi che *tutte* le persone abbiano un metodo compleanno, senza doverlo assegnare “a mano” tutte le volte che ci serve.

Fortunatamente, possiamo usare le funzioni costruttori per assicurarci che tutti gli oggetti di un certo *tipo* (in senso lasco) abbiano i metodi che ci servono.

Più proprietà e metodi vogliamo aggiungere a un oggetto, più è utile definire una funzione per assicurarsi che tutti gli oggetti dello stesso *tipo* siano costruiti esattamente allo stesso modo!

```
function persona(n, e) {  
    return {  
        nome: n,  
        età: e,  
        compleanno() {  
            this.età++  
        }  
    }  
  
var pippo=persona("Pippo", 35)
```

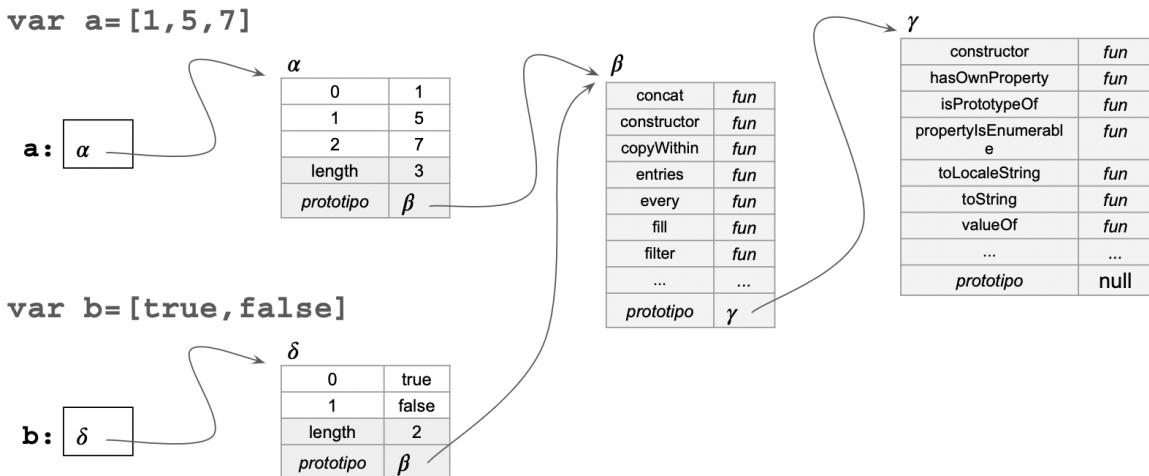
## Prototipo

Ogni oggetto ha un **prototipo**

- Quando si vuole leggere il valore di una proprietà di un oggetto, si guarda se l'oggetto ha la chiave cercata.
  1. Se la chiave è presente, il valore è quello della chiave nell'oggetto
  2. Se la chiave non è presente, e l'oggetto ha un prototipo, si cerca la proprietà nel prototipo
  3. Se la chiave non è presente, e l'oggetto non ha un prototipo, il risultato è `undefined`
- Quando si vuole scrivere il valore di una proprietà di un oggetto, la chiave e il valore vengono inseriti nell'oggetto (eventualmente sovrascrivendo il valore precedente)



Possiamo scoprire chi è il prototipo di un oggetto o accedendo alla sua proprietà “speciale” `o.__proto__` (scritta con due \_\_ prefissi e due suffissi), oppure invocando `Object.getPrototypeOf(o)`



📌 Se aggiungiamo un **metodo** a un particolare **oggetto** (diciamo, *pippo*), il metodo sarà disponibile **solo per quell'oggetto**.

Se aggiungiamo un **metodo** al **prototipo** di un **oggetto**, (diciamo, *Persona*), il metodo sarà disponibile per tutti gli oggetti che hanno lo **stesso prototipo**.

**Tutte le funzioni** (e in particolare: le **funzioni costruttori**) hanno una proprietà che si chiama `prototype`, inizializzata automaticamente dal linguaggio quando si dichiara una funzione, che contiene un oggetto pronto per fare da prototipo per gli oggetti inizializzati dalla funzione.

L'operatore `new` assegna automaticamente come prototipo dell'oggetto creato, il **prototype** della sua funzione costruttrice. Ciò crea un vero **legame permanente** fra gli oggetti e i loro costruttori

```
> function Persona(n,e) { this.nome=n; this.età=e}
< undefined
> Persona.prototype
< ▷ {constructor: f}
> var pippo=new Persona("Pippo",35)
< undefined
> pippo.__proto__
< ▷ {constructor: f}
> pippo.__proto__ == Persona.prototype
< true _
```

Questa caratteristica suggerisce un modo diverso per definire costruttori in JavaScript, che lo avvicina ad altri linguaggi (quelli basati su classi):

1. Aggiungiamo le proprietà che descrivono lo stato di un oggetto **all'oggetto** creato
2. Aggiungiamo i metodi che descrivono il suo comportamento **al prototipo della funzione costruttore**

In questo modo, i *campi* dell'oggetto sono enumerabili, i suoi metodi no.



Tutta la componente **orientata agli oggetti** di JavaScript è basata sul concetto di prototipo

1. Gli oggetti sono creati da `new` tramite funzioni-costruttori
2. Le funzioni-costruttori definiscono implicitamente il prototipo di ogni oggetto creato
3. Tutte le parti “**a comune**” di oggetti della stessa famiglia vengono implementate dai prototipi; le parti “**proprie**” sono implementate da ogni oggetto individualmente

Esempio di costruttore con prototipo

```
function Persona(n, e) {
  this.nome=n
  this.età=e
}

Persona.prototype.compleanno = function() {this.età++}

var pippo=new Persona("Pippo", 35)
pippo.compleanno()

OUTPUT
pippo → {nome: "Pippo", età: 36}
```

## Assegnamenti destrutturanti su Oggetti

Se `O = {n:"Pippo",a:35,c:true}`:

- `{a, c} = O`      **a** → 35      **c** → true
- `{a, b=2} = O`      **a** → 35      **b** → 2

- `{a,b} = 0`       $a \rightarrow 35$        $b \rightarrow \text{undefined}$
- `{a,...r} = 0`       $a \rightarrow 35$        $r \rightarrow \{n:"Pippo",c:true\}$
- `{a:età,n:nome} = 0`       $\text{età} \rightarrow 35$        $\text{nome} \rightarrow "Pippo"$
- `{a:età,b:bimbi=0} = 0`       $\text{età} \rightarrow 35$        $\text{bimbi} \rightarrow 0$
- `{x,y} = f("pluto", 3)`      **estrae 2 fra molti valori di ritorno di una funzione, indicandoli per nome (x e y)**

La destrutturazione sugli oggetti consente anche di avere funzioni con molti parametri, non tutti obbligatori; il chiamante indica **per nome** anziché **per posizione** quelli che vuole indicare.

```
function disegna(x, y, { raggio=0, colore="nero", bordo=1, etichetta="" } )
{
  /* codice di disegno; usa x,y, raggio, colore, bordo, etichetta */
}
```

E chiamandola più di una volta si può decidere quali caratteristiche vuole specificare

```
disegna(5,8, {colore="blu"})
disegna(3, 2)
disegna(5,8,{colore="rosso", bordo=2})
disegna(0,1,stile)
```

## ▼ Insiemi

Un insieme è un raggruppamento di elementi di qualsiasi tipo che può essere individuato mediante una caratteristica comune agli elementi che vi appartengono oppure per semplice elencazione degli elementi dell'insieme

Un insieme contiene tutti elementi differenti

## Cardinalità di operazioni su insiemi



### Corollario:

- $|A| = |A \setminus B| + |A \cap B|$
- $|A \cup B| = |A| + |B| - |A \cap B|$
- $|A \setminus B| = |A| - |A \cap B|$
- $|A \cup B| \leq |A| + |B|$  e sono uguali se e solo se  $A$  e  $B$  sono disgiunti
- Se  $B \subseteq A$  allora  $|B| \leq |A|$
- $|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$
- $|A \times B| = |A| \cdot |B|$
- $|A_1 \times A_2 \times \dots \times A_n| = |A_1| \times |A_2| \times \dots \times |A_n|$

Un generico insieme S di elementi può essere realizzato sfruttando le chiavi degli oggetti JS, associando loro un valore predefinito

```
{ Pipp, Pluto, Topolino }
```

```
{ Pippo: 1, Pluto: 1, Topolino: 1 }
```

## Funzioni per operazioni su Insiemi

### Funzione che verifica se val è contenuto in S

```
function contains(S, val) {  
    return (val in S)  
}
```

### Funzione che inserisce val nell'insieme S

```
function insert(S,val){  
    S[val] = 1  
}
```

### Funzione che rimuove val dall'insieme S

```
function remove(S,val){  
    delete S[val]  
}
```

### Funzione che verifica se l'insieme A è sottoinsieme dell'insieme B

```
function subset(A,B){  
    for (let x in A) {  
        if (!contains(B,x))  
            return false  
    }  
    return true  
}
```

### Funzione che verifica se l'insieme A è uguale all'insieme B

```
function equal(A,B){  
    return subset(A,B) && subset(B,A)  
}
```

### Funzione che restituisce l'intersezione di 2 insiemi

```
function intersection(A,B){  
    let I = {}  
    for (let x in B) {  
        if (contains(A,x))  
            insert(I,x)  
    }  
    return I  
}
```

### Funzione che restituisce l'unione di 2 insiemi

```
function union(A,B){  
    let U = {}  
    for (let x in A)  
        insert(U,x)
```

```

for (let x in B)
    insert(U,x)
return U
}

```

### Funzione che restituisce la sottrazione di 2 insiemi (A - B)

```

function subtract(A,B) {
    let S = {}
    for (let x in A) {
        if(!contains(B,x))
            insert(S,x)
    }
    return S
}

```

### Funzione che calcola la cardinalità di un insieme

```

function cardinality(A) {
    let c = 0
    for (let x in A)
        c++
    return c
}

```

### Funzione che calcola la similarità di Jaccard tra 2 insiemi A e B

```

function jaccard(A,B){
    return cardinality(intersection(A,B))/cardinality(union(A,B))
}

```

## ▼ Alberi

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

I nodi di **grado 1** sono detti **foglie**, i nodi di **grado > 1** sono detti **nodi interni**.

Un **albero radicato** ha un nodo identificato come **radice**. Ogni nodo interno ha **1 o più figli**, che consideriamo ordinati (e numerati  $1 \dots k$ ). Nel caso particolare di  $k = 2$ , ovvero che ogni nodo da solo 2 nodi figli si parla di albero binario; in questo caso, chiameremo i figli sinistro (*sx*) e destro (*dx*).

I nodi senza figli vengono detti *foglie* o nodi terminali; un nodo non foglia è un nodo interno.



Si definisce inoltre **foresta** un grafo non orientato nel quale due vertici qualsiasi sono connessi al più da un cammino (grafo non orientato e privo di cicli). Una foresta risulta costituita da una unione disgiunta di alberi (e questa proprietà giustifica il suo nome); questi alberi costituiscono le sue componenti connesse massimali.

- **Foresta:** grafo non orientato, aciclico e non vuoto (*ogni componente连通的 è un albero*)
- **Foglia:** nodo con grafo 1 senza archi uscenti
- **Radice:** nodo privo di archi entranti
- **Nodo interno:** nodo con grado < 1
- **Nodo unario:** nodo con un solo figlio
- **Nodo non unario:** nodo con più di un figlio



#### *NODI NON UNARI IN UN ALBERO $T$ CON $f$ FOGLIE*

$$NU(f) = \begin{cases} NU(f), & \text{se } x \text{ ha 1 figlio} \\ 1 + NU(f - 1), & \text{se } x \text{ ha 2 figli} \\ 0 & \text{se } f = 1 \end{cases}$$

Si dice **albero** un grafo  $G$  connesso, non orientato e senza cicli. Per essere tale, il grafo deve rispettare almeno una delle seguenti richieste:

- Possedere un solo cammino per ogni coppia di vertici.
- Essere *aciclico massimale* ossia, se gli si aggiunge un altro arco per unire due suoi nodi si forma un ciclo.

- Costituire un *grafo connesso minimale* ossia, se si rimuove un suo arco si perde la connessione.

### *Livello*

Il livello di un nodo in un albero è pari a 1 più il livello del nodo padre, intendendo che la radice ha livello 0. Per esempio un nodo figlio della radice sarà di livello 1, suo figlio sarà di livello 2.

### *Altezza*

L'altezza di un albero è la massima distanza di una foglia dalla radice (*contando gli archi*). Ogni albero binario con  $k$  foglie ha un'altezza  $\geq \log_2 k$ .

### *Lunghezza del cammino*

La lunghezza del cammino interno di un albero binario è data dalla somma dei livelli dei nodi interni, mentre la lunghezza del cammino esterno di un albero binario è data dalla somma dei livelli dei nodi esterni

## **Tipi di albero:**

### **Albero con radice**

Un albero con radice è una **coppia**  $(T, r)$  dove  $T$  è un **albero** e  $r$  un suo **vertice** che viene detto radice. Un albero con radice è quindi un albero in cui viene evidenziato un vertice (la radice); esso viene anche detto **albero radicato**.

### **Albero ordinato**

Un albero ordinato (detto anche piano) è un albero radicato in cui i figli di ogni vertice sono totalmente ordinati.

### **Alberi Binari**

Un **albero binario** è un albero i cui nodi hanno **grado compreso tra 0 e 2**. Per *albero* si intende un grafo *non diretto, connesso e aciclico* mentre per *grado* di un nodo si intende il numero di sotto alberi del nodo, che è uguale al numero di figli del nodo.

Anche l'albero costituito da *un solo nodo e nessun arco* si considera un albero binario valido, sebbene il grado del nodo in questo caso sia nullo.

### Albero completo

Se tutti i nodi interni hanno figli

### Albero pieno

Se tutte le foglie sono alla stessa distanza dalla radice



#### Caratteristiche di un albero binario completo

- Foglie:  $2^h$
- Nodi:  $2^{h+1} - 1$



#### Caratteristiche di un albero pieno

- Foglie:  $2^h$
- Nodi:  $2^{h+1} - 1$
- Nodi interni:  $2^h - 1$

## Funzioni strutturali su alberi binari:

### Dimensioni di un albero binario

1.  $\text{size}(\lambda) = 0$
2.  $\text{size}(N(t_1, t_2)) = \text{size}(t_1) + \text{size}(t_2) + 1$

### Altezza di un albero binario

1.  $height(\lambda) = -1$
2.  $height(N(t_1, t_2)) = \max(height(t_1), height(t_2)) + 1$

**Visita di nodi che vengono inseriti in un array con la funzione `app()` definita induttivamente**

1.  $visit(\lambda) = []$
2.  $visit(N(t_1, a, T_2)) = app(visit(t_1), a : visit(t_2))$

### Lunghezza

1.  $lengthBT(\lambda) = \lambda$
2.  $lengthBT(N(t_1, w, t_2)) = N(lengthBT(t_1), |w|, lengthBT(t_2))$

## Alberi binari su javascript

Si può rappresentare un nodo con un oggetto JavaScript con tre chiavi:

- `val:` il valore associato al nodo
- `sx:` il figlio sinistro del nodo (chiave assente o null se il figlio sx manca)
- `dx:` il figlio destro del nodo (chiave assente o null se il figlio dx manca)

```
{ val: 5, sx: ..., dx: ... }
```

### Come si possono indicare i figli?

I figli di un nodo sono anch'essi nodi... quindi si può usare la stessa codifica

```

let pino={val: "Giuseppe"}, gigi={val: "Luigi"}
let ciccio={val: "Francesco", sx: pino, dx: gigi}

oppure direttamente:

let ciccio={val: "Francesco", sx: {val: "Giuseppe"}, dx: {val: "Luigi"}}

```

Il valore (val) associato a un nodo potrebbe dunque essere qualunque cosa:

- **Una stringa**, come "Francesco" o "Luigi"
- **Un numero**, come 17 o 3.1415926
- **Un booleano**
- **Una funzione**
- **Un array**
- **Un oggetto**

## Codifica degli alberi

Gli alberi sono naturalmente una struttura **ricorsiva**.

Abbiamo nodi, i cui figli sono nodi, i cui figli sono nodi ... fino ad arrivare a nodi senza figli. Se sappiamo come manipolare un nodo e i suoi figli per fare qualcosa... allora sappiamo come manipolare un albero intero, di qualunque dimensione!

## Funzioni su alberi binari

*Funzione max(tree) che - dato un albero tree - restituisce il valore massimo tra quelli contenuti nei suoi nodi*

```

function max(tree) {
  if (!tree)
    return undefined
  // Assumo che il massimo sia nel nodo corrente
  let m = tree.val
  // Se c'è un sottoalbero sx/dx, calcolo il massimo del sottoalbero e aggiorno il massimo corrente
  if (tree.sx) {
    let m1 = max(tree.sx)
    if (m1 > m) m = m1
  }
  if (tree.dx) {
    let m2 = max(tree.dx)
    if (m2 > m) m = m2
  }
  return m
}

```

```

        let m1 = max(tree.dx)
        if (m1 > m) m = m1
    }
    // Ritorno massimo calcolato (tree.val, se foglia)
    return m
}

```

*Funzione sum(tree) che - dato un albero tree - restituisce la somma dei valori contenuti nei suoi nodi*

```

function sum(tree) {
    if (!tree)
        return 0
    let somma = tree.val
    if (tree.sx) somma += sum(tree.sx)
    if (tree.dx) somma += sum(tree.dx)
    return somma
}

```

*Funzione even\_odd\_visit(tree) che - dato un albero tree - stampa i valori dei nodi attraversati implementando la seguente visita: se il valore di un nodo è pari, visita il figlio sinistro; altrimenti, visita il figlio destro*

```

function even_odd_visit(tree) {
    if (!tree) return
    console.log(tree.val)
    if (tree.val%2==0)
        even_odd_visit(tree.sx)
    else
        even_odd_visit(tree.dx)
}

```

*Funzione even\_odd\_sum(tree) che - dato un albero tree - restituisce la somma dei valori dei nodi attraversati implementando una visita analoga a quella di even\_odd\_visit*

```

function even_odd_sum(tree) {
    return _even_odd_sum(tree,0)
}
function _even_odd_sum(tree,acc) {
    if (!tree) return acc
    if (tree.val%2==0)
        return _even_odd_sum(tree.sx, tree.val+acc)
}

```

```
    else
        return _even_odd_sum(tree.dx, tree.val+acc)
}
```

## Alberi k-ari su javascript

Come abbiamo visto già negli alberi binari dove abbiamo  $k = 2$ , con gli alberi k-ari possiamo spingerci oltre e considerare  $k > 2$ .

Non possiamo più dare un *nome* ai figli ( $dx$  e  $sx$ )... ora potremmo avere un numero qualunque di figli.

```
{ val: 5, figli: [ ... ] }
```

Alberi k-ari a valori arbitrari:

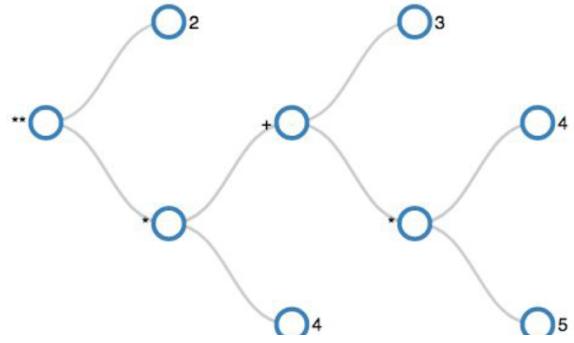
```
{ val: " ... ", figli: [ ... ] }
{ val: { ... }, figli: [ ... ] }
{ val: [ ... ], figli: [ ... ] }
```

Possiamo sfruttare il fatto che il valore associato a un nodo può essere di qualunque tipo (anche diverso tra nodi) per creare strutture interessanti

**Esempio:** alberi sintattici

Immaginiamo un albero in cui i nodi intermedi sono operatori (aritmetici) e le foglie sono operandi (numerici)

```
{
  val: "***",
  sx: { val: "**",
        sx: { val: "4",
              dx: { val: "+" },
              sx: { val: "*",
                    sx: { val: "5" },
                    dx: { val: "4" }
                  },
              dx: { val: "3" }
            },
        dx: { val: "2" }
      }
}
```



$$(4 * ((5 * 4) + 3)) * *2$$

## Rappresentazione alternativa *k*-ari

Possiamo rappresentare un albero *k* – *ario come se fosse binario*:

```
{ val: 5, child: ... , next: ... }
```

