Dall'Algoritmo del Torneo agli Heap

corso di laurea in **Matematica**

Informatica Generale, Lezione **19(a)**

Ivano Salvo

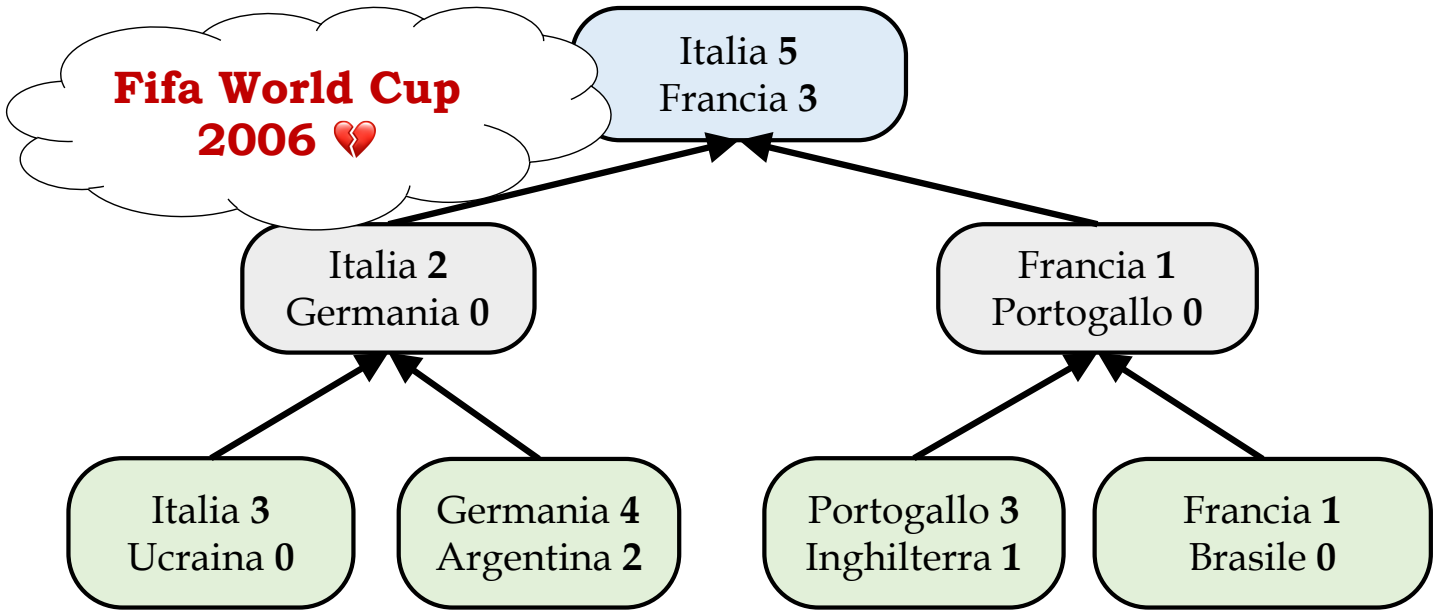
SAPIENZA
UNIVERSITÀ DI ROMA

L'algoritmo del Torneo

Abbiamo visto che è possibile usare un algoritmo **divide et impera** per calcolare il **massimo/minimo** di un **vettore**, basato sull'idea dei consueti **tornei a eliminazione diretta**.

Abbiamo visto **la sua complessità** è comunque $\Theta(n)$, esattamentente come nell'algoritmo naif, ma abbiamo arguito su **alcune virtù**:

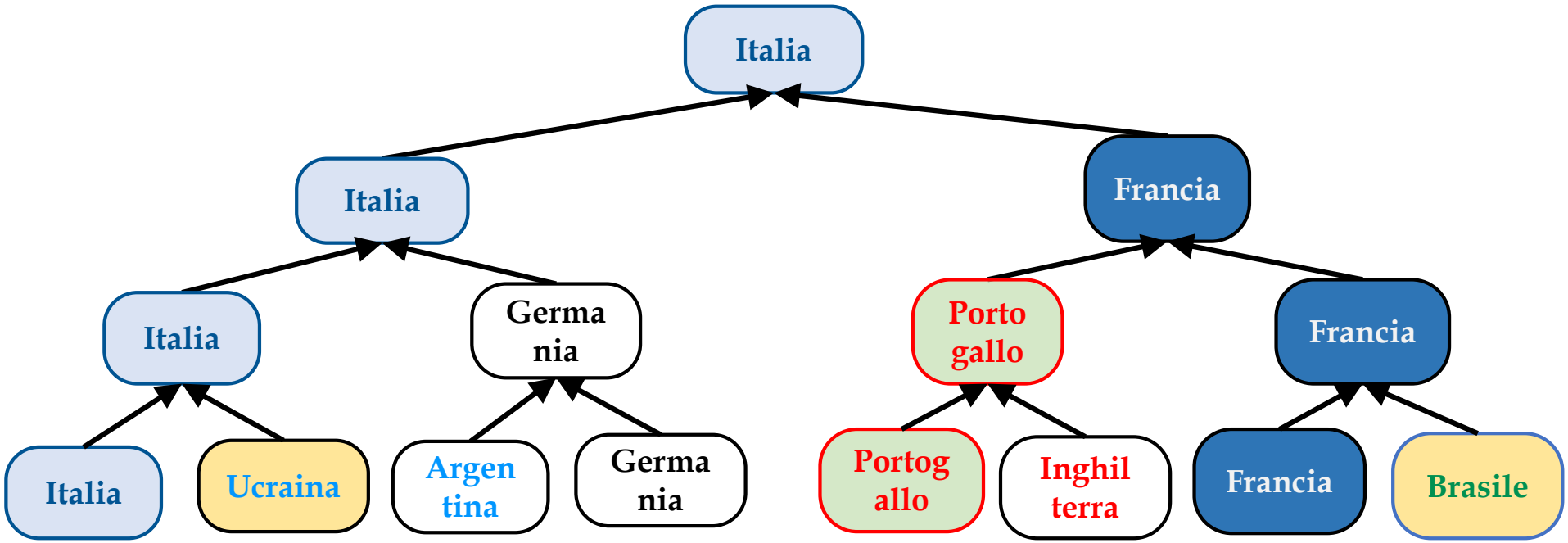
- **tutti gli elementi** vengono **confrontati** $\mathcal{O}(\log n)$ volte, e
- sarebbe **facile computazionalmente** (avendo memorizzato la sequenza di "partite") trovare il **secondo elemento**, semplicemente ispezionando i **perdenti del vincitore**, ancora in **tempo** $\Theta(\log n)$.



L'albero del Torneo

Non è molto difficile costruire **l'albero del torneo** in $\Theta(n)$.

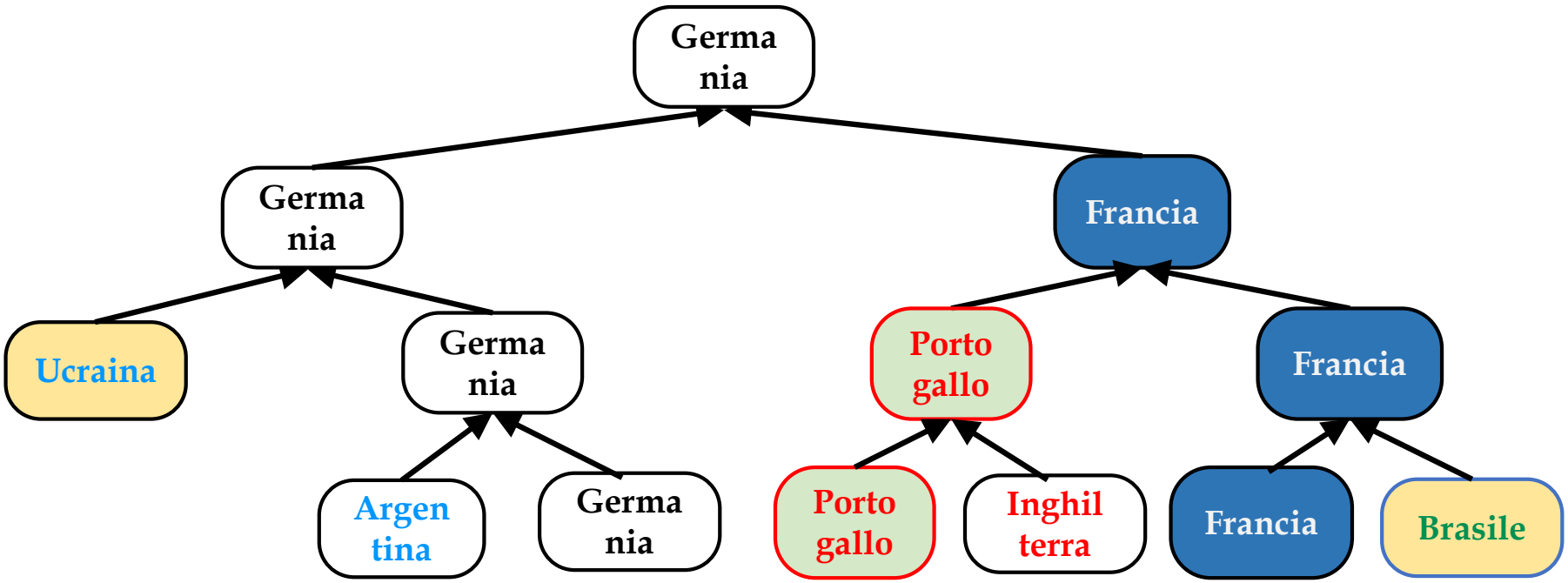
```
def torneo(u, inf, sup):
    if inf + 1 == sup:   # caso base foglia
            return v[inf], mkLf(v[inf])
    if inf == sup: return -∞, EMPTY # caso base vuoto
    m = puntoMedio(inf, sup)
    v1, t1 = torneo(u, inf, m)
    v2, t2 = torneo(u, m, sup)
    return max(v1, v2), makeBT(max(v1, v2), t1, t2)
```



Selection Sort "ottimo"

Lasciamo per [⏵ **Esercizio**] scrivere un **ordinamento** che riscrive il vettore u ordinato **estraendo i massimi successivi** dall'albero in $\mathcal{O}(\log n)$ allo scopo di ri-ordinare il vettore (le complessità scendono via via che si fanno le "estrazioni" e l'albero si abbassa).

Qui vediamo la Germania venire "a galla", una volta rimosso il massimo, cioè l'Italia, che quindi è la seconda (anche se non è arrivata in finale).

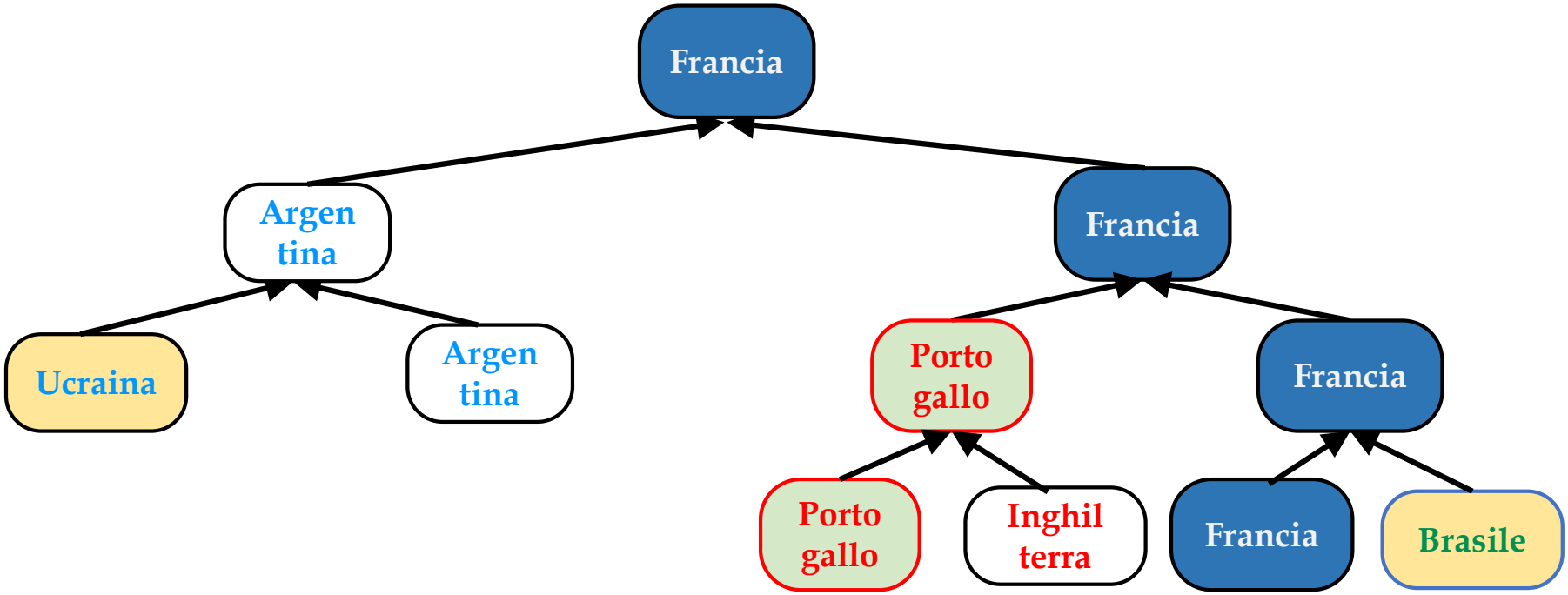


Selection Sort "ottimo": difetti

Benché si ottenga un "**Selection Sort**" divide et impera **ottimo** che ordina in $\Theta(n \log n)$, vediamo alcuni difetti:

- dobbiamo **allocare un albero** con $2n$ **nodi**
- molti **valori sono ripetuti** (fino a log n volte per il vincitore)

[Nel frattempo continuiamo l'immaginario "torneo degli sconfitti"]

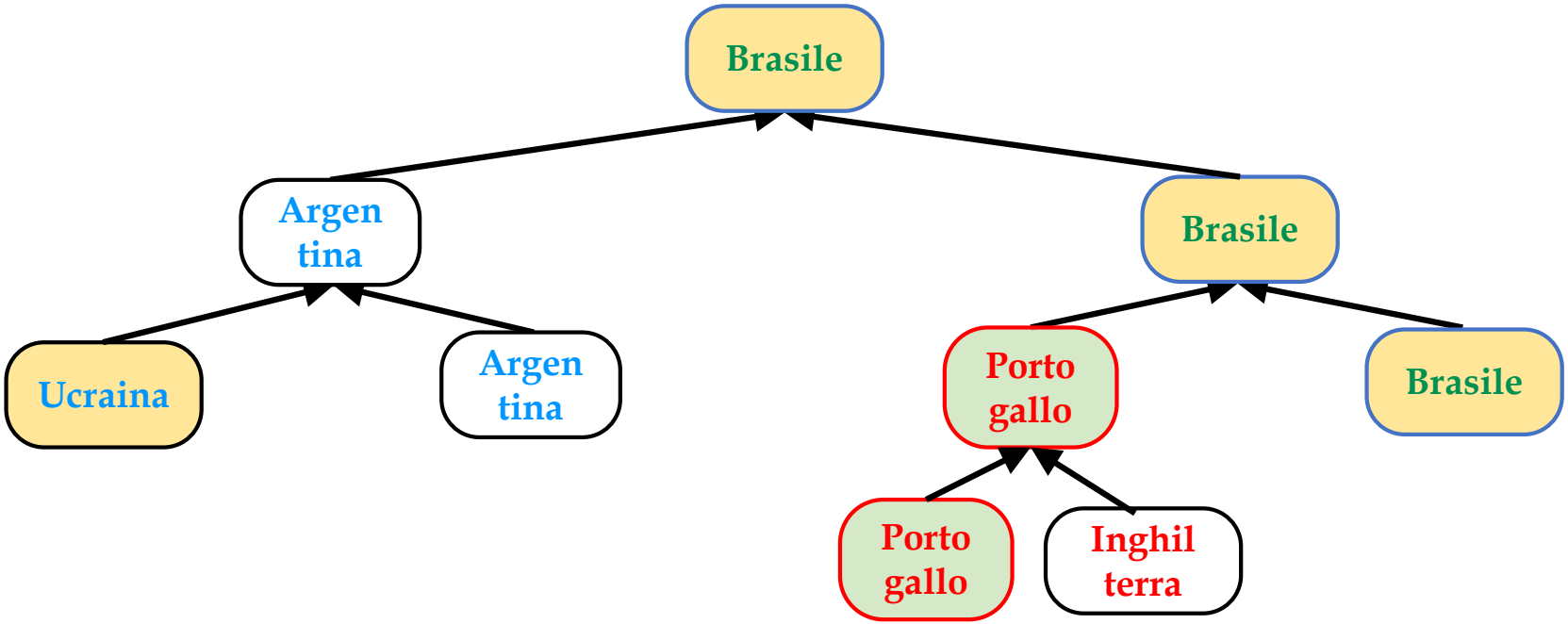


Dall'albero del Torneo agli Heap

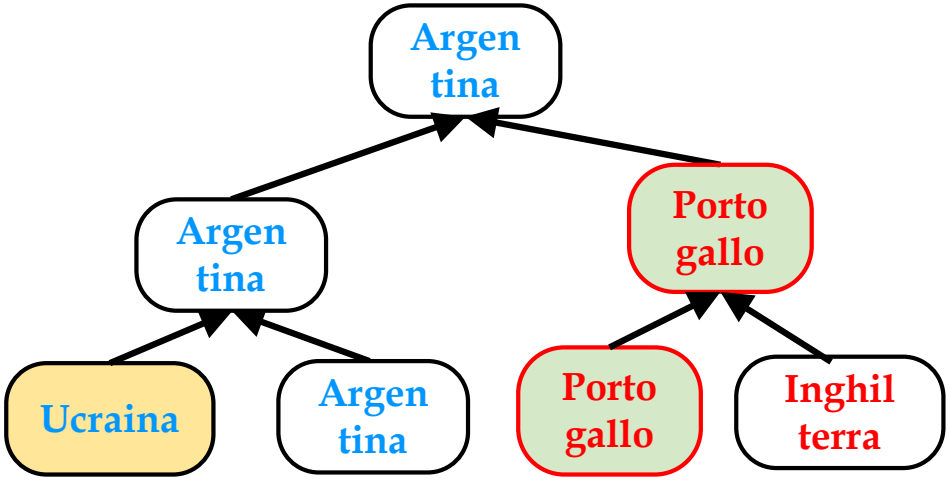
L'idea della struttura dati **heap** è ispirata ai tabelloni dei tornei, però cercando di migliorare i problemi visti sopra:

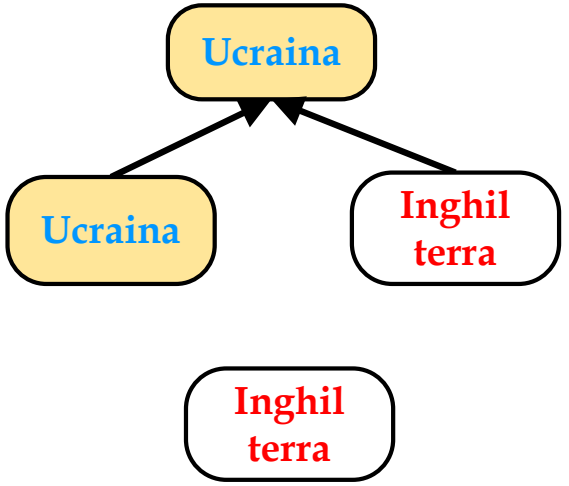
- ogni elemento **appare una sola volta**
- si usa la **rappresentazione posizionale** in un vettore.

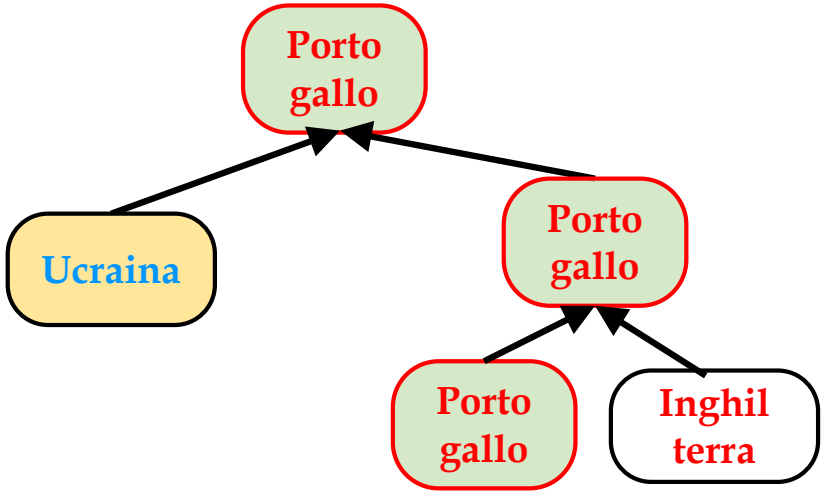
L'aspetto sorprendente è che si può **costruire uno heap in-place** in un vettore in **tempo $\Theta(n)$**.



Finiamo il "torneo degli sconfitti"







Morale: con $\Theta(n \log n)$ partite abbiamo stabilito la classifica **totale** (**ognuna delle n squadre gioca al più $\Theta(\log n)$** partite).

Nel seguito vediamo come quest'idea, grazie agli **heap** si riesce a usare per un algoritmo ottimo in place di ordinamento.

L'algoritmo di Ordinamento HeapSort

corso di laurea in **Matematica**

Informatica Generale, Lezione **19(a)²**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Definizione di Heap

Uno **heap** è:

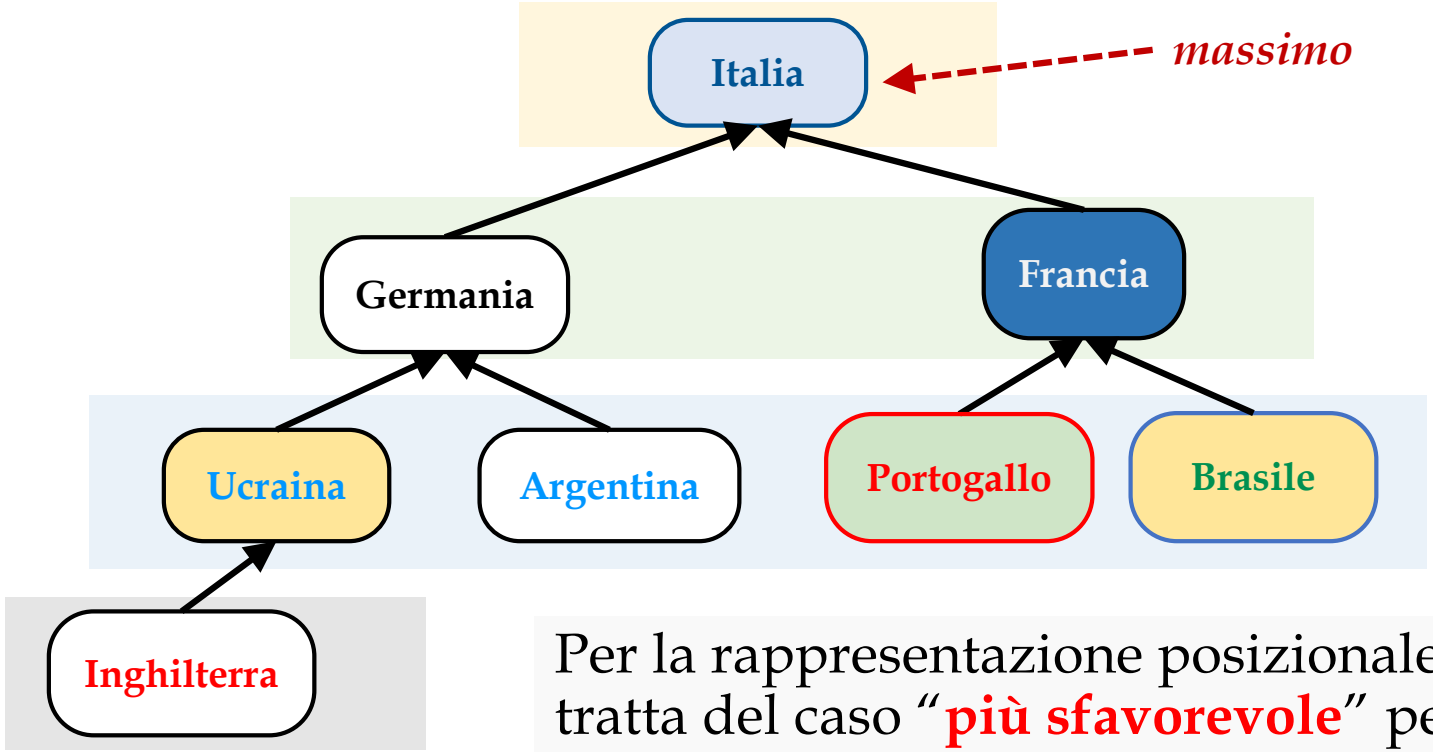
- un **albero binario** [usualmente in **rappresentazione posizionale**]
- **completo** o **quasi completo**, cioè con tutti i **livelli pieni**, tranne l'ultimo, i cui nodi sono **addensati a sinistra**;
- in cui la **chiave** di ogni nodo è **maggiore o uguale** alla chiave dei **suoi figli** (proprietà di **ordinamento verticale**).

Osservazione: una delle attrattive della rappresentazione posizionale è che **le operazioni** *left*(*B*), *right*(*B*) e *parent*(*B*) si implementano in modo **estremamente efficiente** perché:

- *B* è semplicemente **l'indice di un vettore** e
- le operazioni sono **divisioni e moltiplicazioni per 2** che in binario corrispondono a **shift destri/sinistri** della rappresentazione binaria di un intero.

Esempio di Heap

Vediamo l'esempio "calcistico" nella rappresentazione ad Heap.



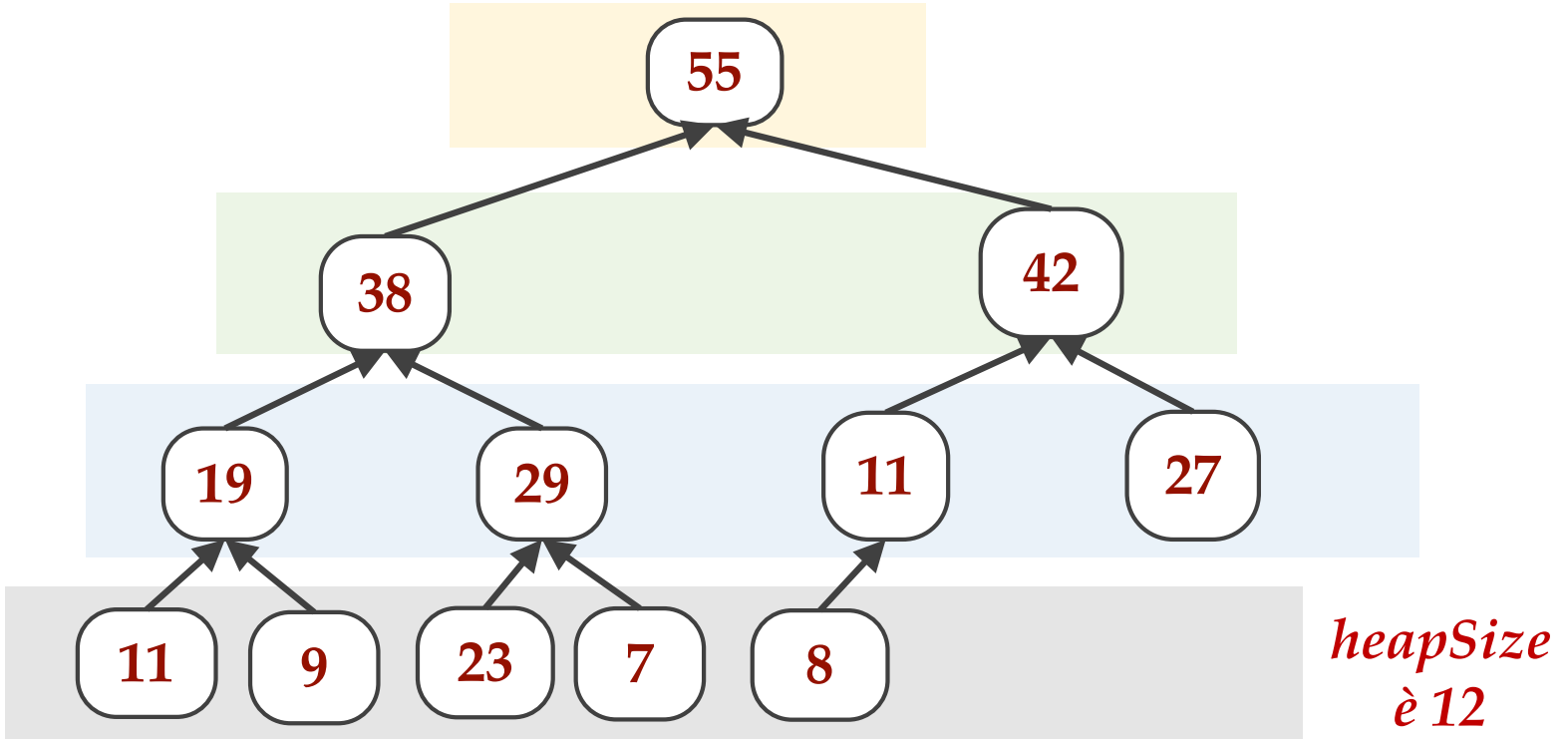
Per la rappresentazione posizionale, si tratta del caso "**più sfavorevole**" perché essendo gli elementi 8 (quindi una potenza di 2) serve un vettore lungo 15.

ITA	GER	FRA	UKR	ARG	POR	BRA	ING	•	•	•	•	•	•	•
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Esempio di Heap

Ecco un esempio numerico in cui è più **facile** (e **oggettivo**) valutare il ≤). Verificate che tutte le proprietà degli heap sono soddisfatte.

Sotto la corr. rappresentazione posizionale: il vettore contiene spazio per tutto l'ultimo livello (si tiene una variabile **`heapSize`**).



55	38	42	19	29	11	27	11	9	23	7	8	•	•	•
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Costruzione di un Heap: heapify

Il cuore della costruzione di uno heap è la funzione heapify.

La precondizione di **`heapify`** è di avere in input un albero binario B che **vìola la condizione di heap solo nel nodo radice**.

È una funzione ricorsiva: prima verifica se è violato l'ordinamento verticale e in caso affermativo sceglie il **sottoalbero B' con la chiave maggiore** (funzione **`violaOrdVert`**, prossima slide 

).

Se l'ordinamento è violato, si scambia la chiave alla radice con la chiave alla radice di B' e **`heapify`** viene chiamata **ricorsivamente** su B'.

```
def heapify(B):
# REQ: B è un heap, a parte la radice
    vov, B' = violaOrdVert(B)
    if vov:
        key[B], key[B'] = key[B'], key[B]
        heapify(B')
```

Costruzione di un Heap: heapify

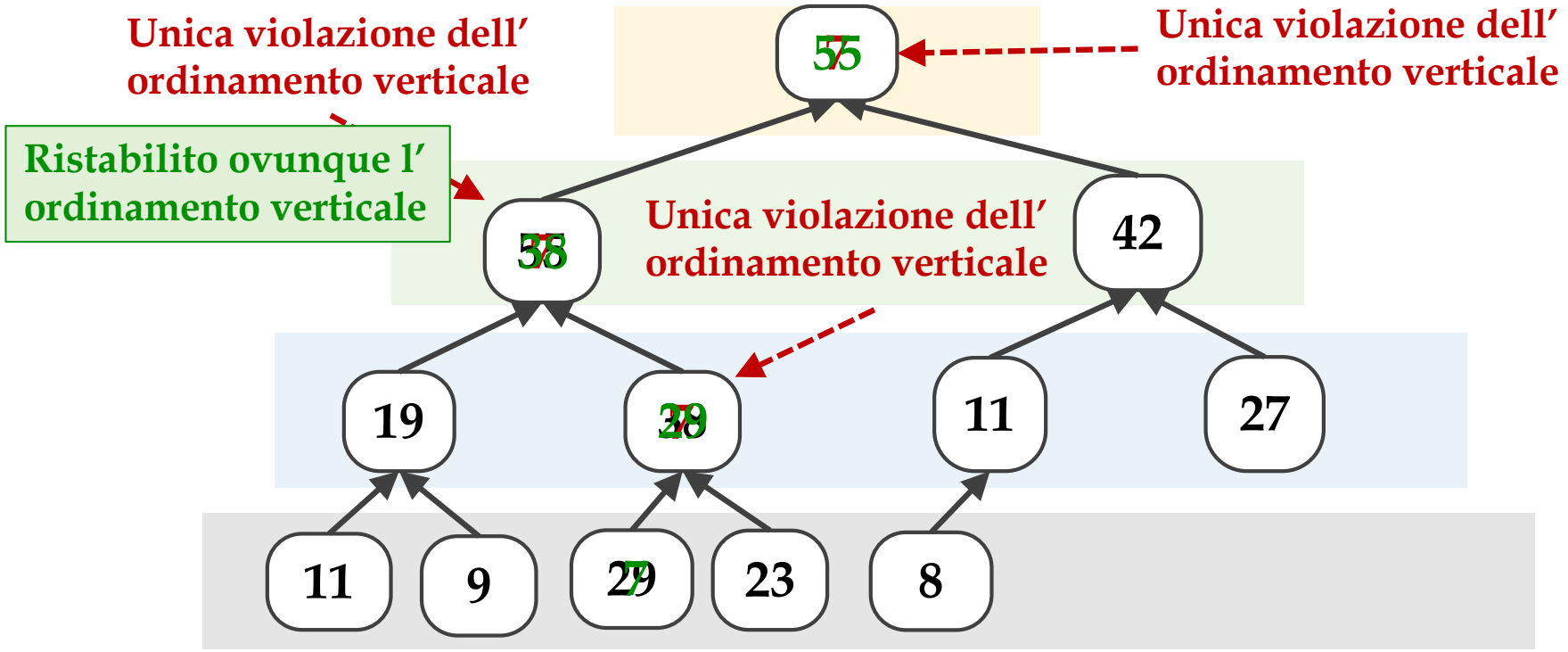
La funzione **`noOrdVert`** è un po' noiosa da scrivere, ma è **istruttiva** perché fa vedere che **è sempre necessario verificare che un sottoalbero non sia vuoto prima di leggere gli attributi** `key` o `left`.

```
def noOrdVert(B):
    if B==EMPTY: return False, B
    maxSon = key[B]
    if left(B) ≠ EMPTY and key[left(B)]>maxSon:
        maxSon, B′ = key[left(B)], left(B)
    if right(B) ≠ EMPTY and key[right(B)]>maxSon:
        maxSon, B′ = key[right(B)], right(B)
    if maxSon > key[B]: return True, B′
    return False, B
```

Esempio di Heapify

Vediamo sul nostro albero un esempio di esecuzione di heapify.

Verifichiamo le precondizioni.



55	38	42	19	39	11	27	11	9	29	23	8	•	•	•
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Costruzione di un Heap: buildHeap

Come usare **`heapify`** per costruire un heap partendo da un vettore qualsiasi?

Il segmento di vettore v[$n/2, n$) può essere visto come un insieme **foglie** che **sono banalmente heap**.

Gli indici nell'intervallo [$n/4, n/2$) sono le **radici di heap di altezza 1** (3 nodi) che possono **violare** l'**ordinamento verticale** al più **alla radice**. **Applico `heapify`** a tutti questi nodi.

A questo punto, sul segmento [$n/8, n/4$) ho di nuovo degli alberi di altezza 2 che vìolano la condizione di ordinamento verticale al più alla radice. Applico **`heapify`**, sistemando così heap di altezza 2.

Morale: posso costruire un heap, semplicemente **applicando `heapify`** andando **da destra a sinistra** a partire **dall'indice $n/2$**.

```
def buildHeap(v):
    n = len(v)
    for j=n/2 to 0 step -1:
        heapify(j)
```

Complessità di buildHeap

Siccome **heapify** è $\Theta(h)$ e si costruisce un heap (=quasi completo), $h = \log n$ **buildHeap** è alla peggio $\mathcal{O}(n \log n)$, visto che chiama $n/2$ volte **heapify**.

C'è il **legittimo dubbio** che questa sia **solo una limitazione superiore** perché le altezze di molti heap è **molto minore di** $\log n$.

Dubbio fondato: con conti più accurati si può far vedere che la complessità di **buildHeap** è $\Theta(n)$.

Siccome **buildHeap** chiama **heapify** su esattamente $n/2^h$ heap di altezza h (per $h = 1, \ldots, \log n$), ho:

$$\sum_{h=1}^{\log_2 n} h\frac{n}{2^h} = n\sum_{h=1}^{\log_2 n} \frac{h}{2^h} \leq n\sum_{h=1}^{\infty} \frac{h}{2^h} = 2n$$

(**serie geometrica** di ragione ½, già incontrata un paio di volte…)

```
def buildHeap(v):
    n = len(v)
    for j=n/2 to 0 step -1:
        heapify(j)
```

Algoritmo heapSort

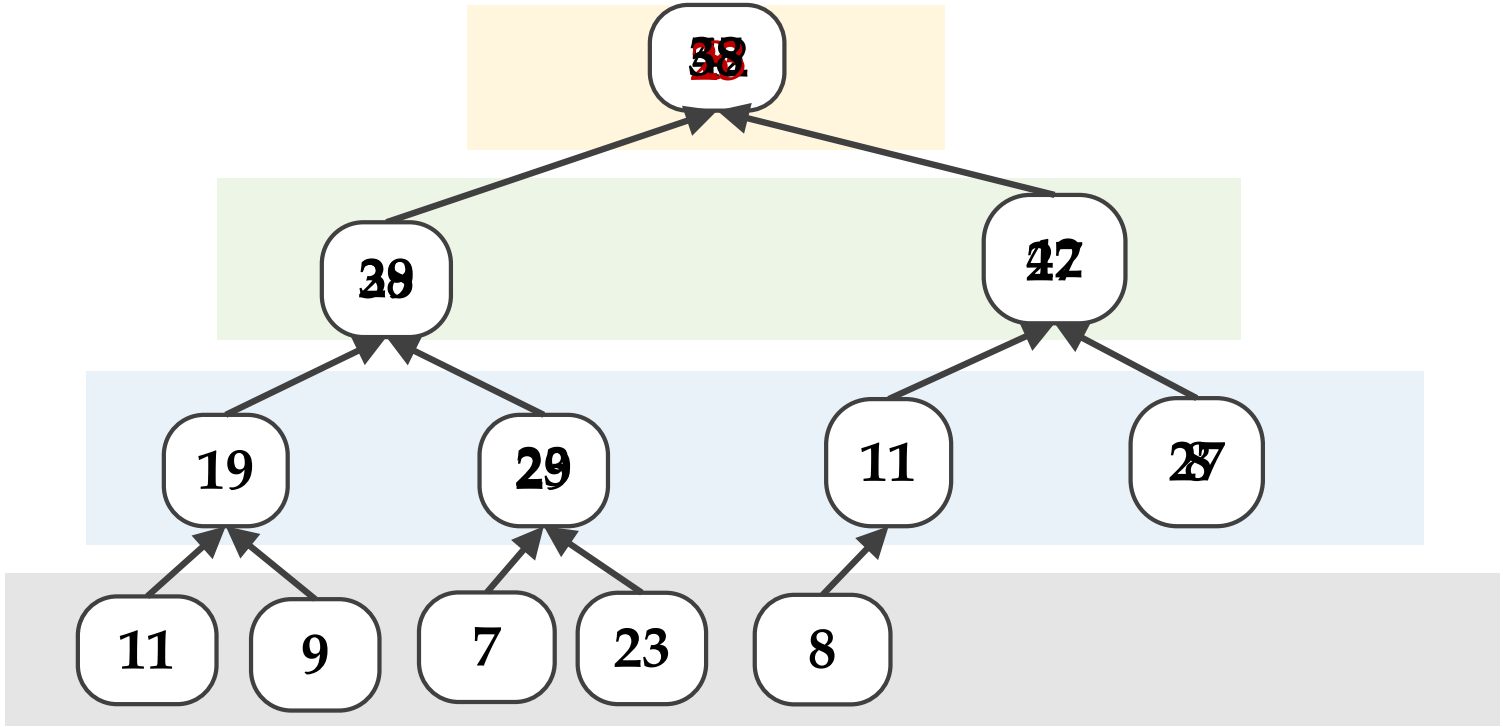
Idea:

- **scambio il massimo** (in prima posizione per def. di heap) con l'ultimo elemento del vettore mettendolo al **posto giusto dopo l'ordinamento** (esattamente come in Selection Sort).
- Si diminuisce di 1 la dimensione dell'heap. Ho un heap che **viola l'ordinamento verticale** al più alla **radice**. Chiamo **`heapify`**.
- Ripeto finché non ho sistemato tutti gli elementi chiamando n-1 volte **`heapify`**. La complessità è $\Theta(n \log n)$ e il limite è stretto, perché faccio almeno $n/2$ **volte `heapify`** su heap di **altezza** massima (**log** n).

```
def heapSort(v):
    n = len(v)
    buildHeap(v) #Θ(n)
    heapSize = n #delimita l'heap
    for j=n-1 to 2 step -1:
    #INV: Ord(v[0,j))
        v[0], v[j], heapSize = v[j], v[0], heapSize - 1
        heapify(0)
```

Esempio di HeapSort

Abbiamo costruito il nostro consueto heap. heapSize è 12.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
19	29	11	11	29	11	23	11	9	7	42	55	•	•	•

Code con priorità

In una coda con priorità, l'**elemento** che viene **estratto** è quello con **maggiore priorità**.

Gli heap sono una struttura dati particolarmente adatta a rappresentare questo tipo di dato, in quanto **estrazioni** e **inserimenti** si possono fare in tempo $\boldsymbol{\theta(\log n)}$.

Infatti l'**estrazione** del massimo costa $\theta(1)$, in quanto il massimo è sempre il primo elemento del vettore, ma poi occorre chiamare `heapify` per ripristinare la struttura dati heap.

Analogamente, l'**inserimento** costa $\boldsymbol{\theta(\log n)}$ ma richiede qualche tecnica leggermente più complicata.

◗ Segnalo un interessante problema sulle code con priorità, detta **starvation** (morte per fame): un elemento potrebbe **non uscire mai dalla coda** perché **entrano** sempre **elementi con maggiore priorità**.

Si usano tecniche di **aging** (gli elementi aumentano di priorità col tempo) per evitare queste situazioni usualmente indesiderate.