Per fare un Albero ...

corso di laurea in **Matematica**

Informatica Generale, Lezione **17(a)**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Varietà di Alberi e definizioni

Alberi: generalità

Un **albero** è una **struttura dati non lineare**.

Abbiamo già definito gli **alberi binari** come **tipo di dato induttivo**, e li abbiamo usati più volte soprattutto come **strumento concettuale** per analizzare/pensare algoritmi e problemi, ad esempio l'albero di decisione di un algoritmo di ordinamento.

Sono alberi, ad esempio:

- **tutte le chiamate ricorsive** generate da un algoritmo **divide et impera** (è **binario** nel caso tipico in cui un problema viene risolto partendo dalle soluzioni di **due sotto-problemi**)
- la **struttura sintattica** di **espressioni** (qui sono alberi binari perché usualmente consideriamo operazioni binarie), **programmi**, e altri linguaggi parentesizzati;

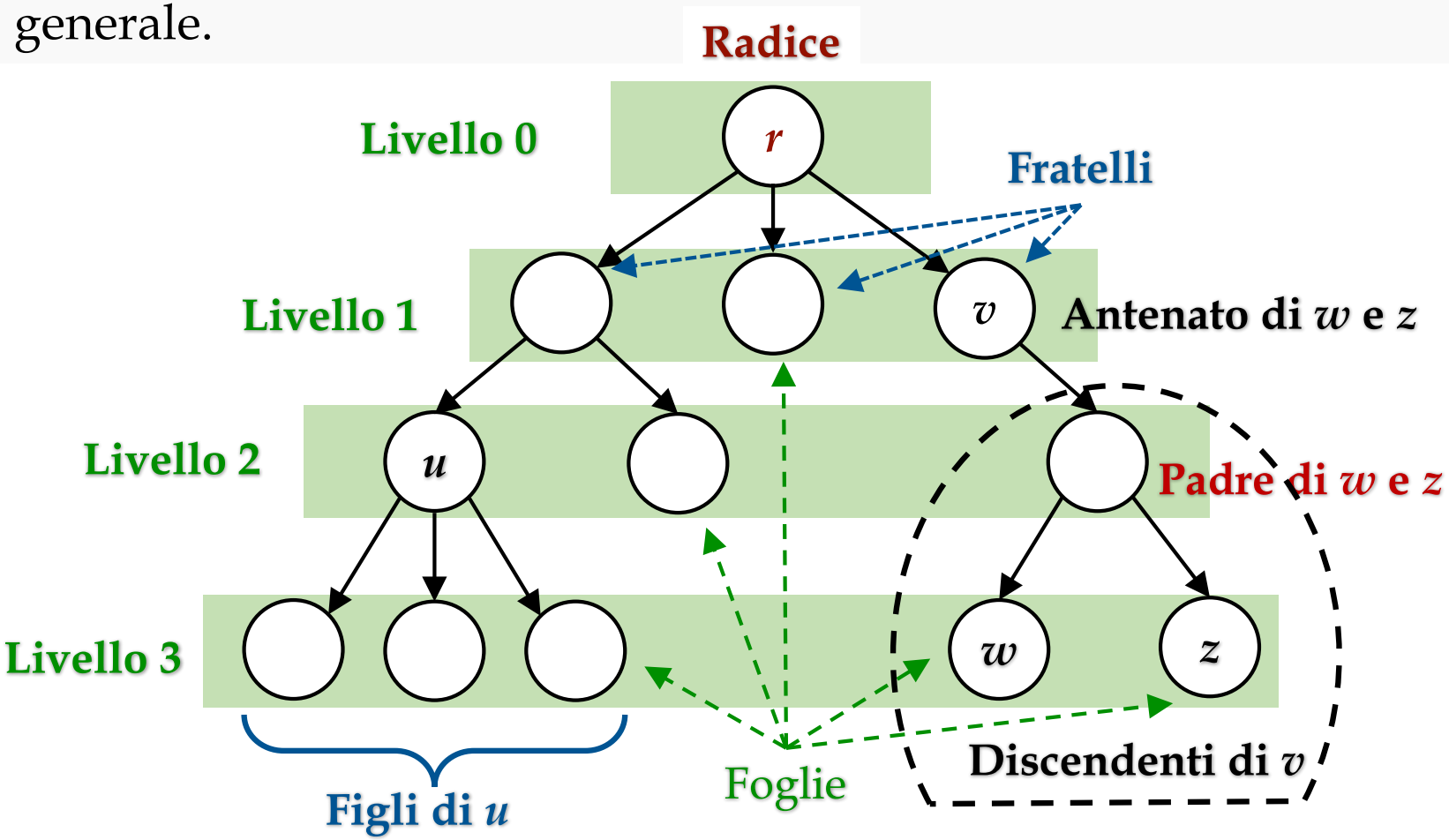
Oltre l'informatica, sono alberi ad esempio:

- gli **alberi genealogici**
- le **classificazioni gerarchiche** (ad esempio la classificazione degli organismi viventi in classi, ordini, famiglie, specie…)
- i **possibili futuri** in un **gioco di strategia** come scacchi o Tris.

Alberi Radicati: Nomenclatura

Sugli alberi c'è una ricca nomenclatura, perlopiù ispirata agli alberi genealogici (in versione, **patriarcale**).

Noi ci occuperemo soprattutto di "alberi radicati", alberi in cui c'è un nodo speciale, detto **radice**. Vedremo nel seguito una forma più generale.



Definizioni

In un albero radicato ci sono **nodi** (pallocchi) e **archi** (le frecce che gli uniscono), che possono essere entranti o uscenti (hanno una **direzione**). La **radice** è l'unico nodo **senza archi entranti**.

Tutti **gli altri nodi** hanno **esattamente un arco entrante**.

I **figli** di un nodo u sono tutti i nodi v per cui c'è un arco $u \to v$.
Se ogni nodo **ha al più due** (o k) **figli**, si dice che l'albero è **binario** (**k-ario**).

Un albero k-ario **è pieno** se tutti i nodi **o** sono **foglie** o hanno **esattamente k figli**. È **completo** se **tutti i livelli hanno k^n nodi**.

Un albero è **ordinato**, se viene stabilito **un ordine tra i sottoalberi**. Ad esempio, negli alberi binari, posso immaginare che il **figlio sinistro preceda** sempre il **figlio destro**.

Un albero è **quasi-completo** se è completo fino al livello k-1, e tutti i nodi dell'ultimo livello sono a "sinistra".

L'**altezza** è la **distanza** (in **numero di archi**) dalla radice ai nodi dell'ultimo livello: se l'altezza è h, ho h+1 livelli, numerati da 0 a h.

Altezza, nodi e bilanciamento

Chiamiamo n il numero dei nodi e h l'altezza di un albero

Abbiamo già visto (Lezione **4**) una semplice dimostrazione che in un albero binario completo, ci sono $n = 2^{h+1}-1$ nodi e 2^h foglie
(⏵ **Esercizio**: generalizzare queste formule agli alberi k-ari).

Questo implica che in un **albero completo** l'altezza $h = \log(n+1) - 1$
o se preferite, per le proprietà dei logaritmi, $h = \log((n+1)/2)$.

L'altezza è logaritmica rispetto al numero dei nodi anche sotto **ipotesi più deboli**. Usualmente, si danno le seguenti nozioni di **albero bilanciato** (notare che sono definizioni **induttive**):

Dato un albero binario B con sottoalberi L (sinistro) e R (destro).

B è **bilanciato in altezza** (***h*-bilanciato**) se $|h(L) - h(R)| \leq 1$ e a loro volta L ed R sono bilanciati in altezza.

B è **bilanciato nel numero dei nodi** (***n*-bilanciato**) se $|\#(L) - \#(R)| \leq 1$ e a loro volta L ed R sono bilanciati nel numero dei nodi.

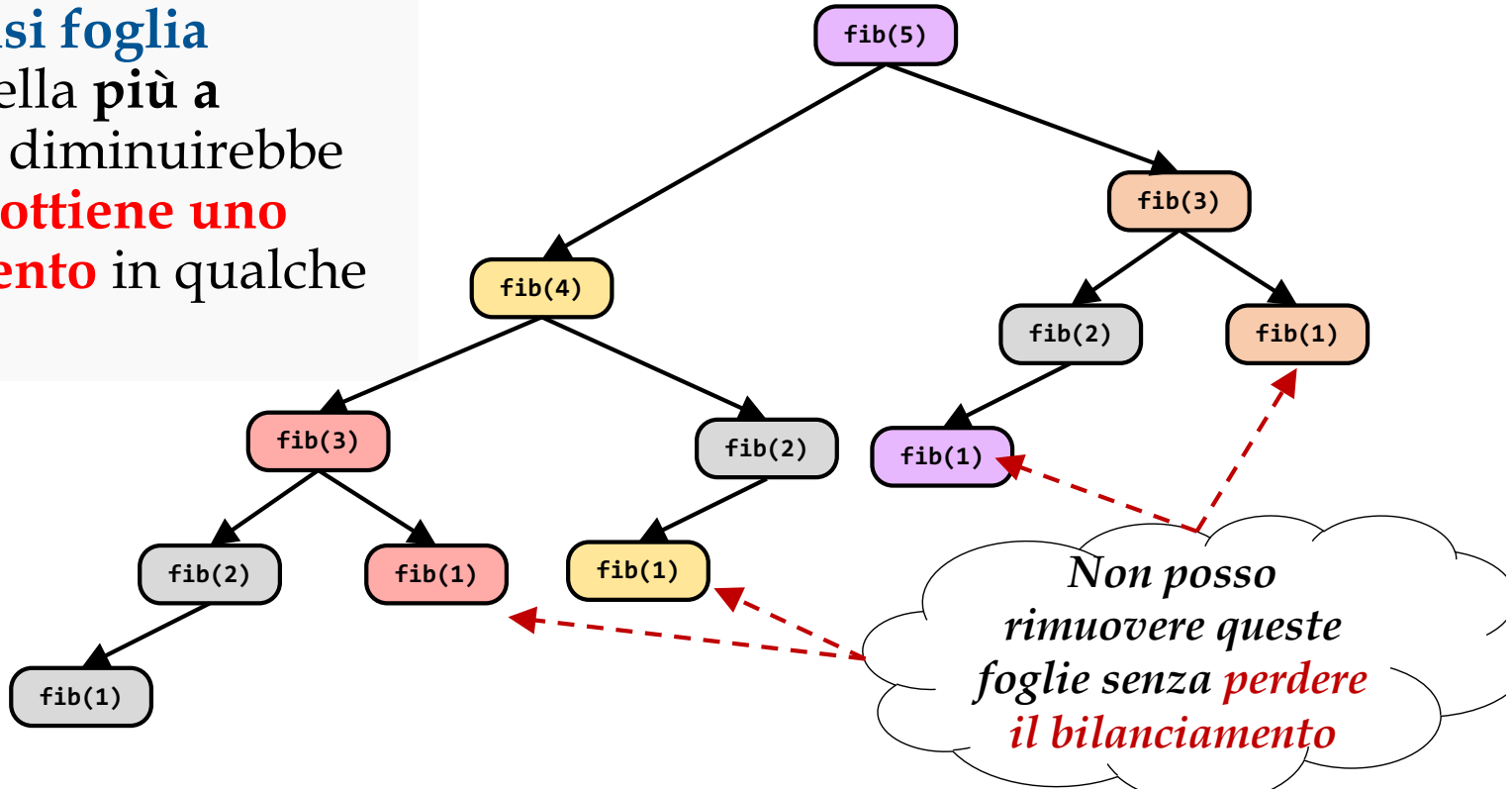
Si può dimostrare (laborioso) l'essere ***n*-bilanciato è una condizione più forte** e quindi se B è n-bilanciato allora è anche h-bilanciato.

Bilanciamento: altezza vs nodi

Teorema: Se B è un albero bilanciato in altezza con n nodi di altezza h, allora $h = \mathcal{O}(\log n)$.

Senza una dimostrazione formale, vediamo l'**albero più sbilanciato** di **altezza 4**: è l'albero delle chiamate ricorsive della funzione di **Fibonacci inefficiente** (senza le chiamate a *fib*(0)).

Osservate che **rimuovendo una qualsiasi foglia** (**escluso** quella **più a sinistra** che diminuirebbe l'altezza) si **ottiene uno sbilanciamento** in qualche sottoalbero.



Rappresentazioni di Alberi (Binari)

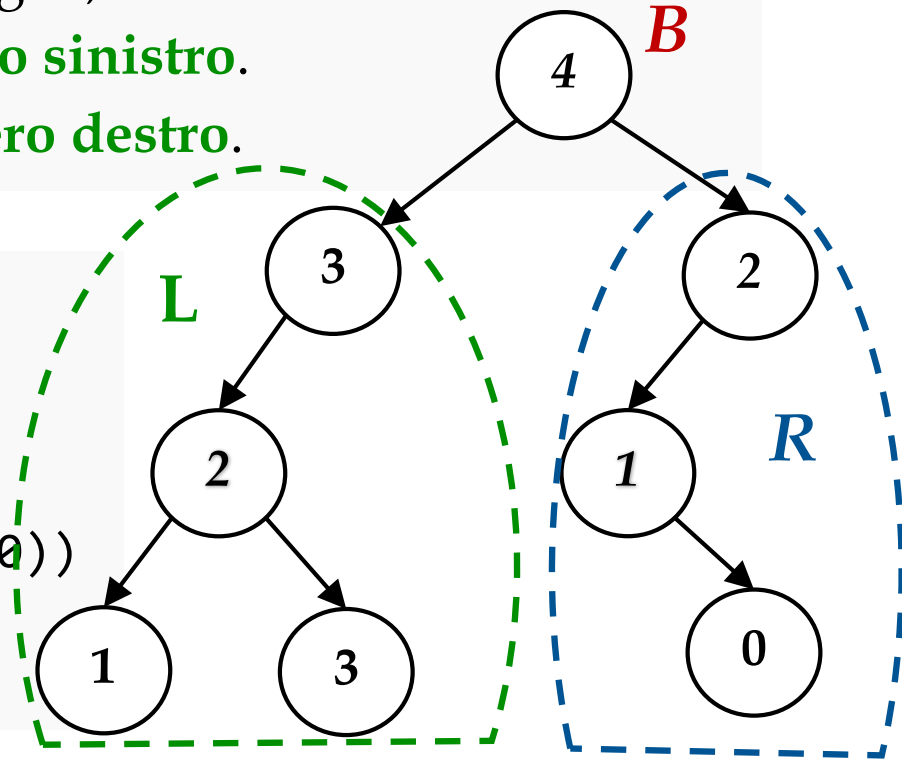
Operazioni del Tipo Albero Binario

Le operazioni che ci aspettiamo sempre disponibili dal **tipo di dato albero binario** sono:

- `B = mkEmpty([n])`: **crea** un albero binario vuoto [con *n* posti, **opzionale**].
- `B = mkBT(r, L, R)`: **costruisce** un albero *B* con radice *r* e sottoalbero sinistro *L* e sottoalbero destro *R*.
- `B = mkLf(r)`: **costruisce** un albero foglia, con etichetta *r*.
- `L = left(B)`: restituisce il **sottoalbero sinistro**.
- `L = right(B)`: restituisce il **sottoalbero destro**.

Ad esempio, l'albero *B* in figura si può costruire con l'espressione:

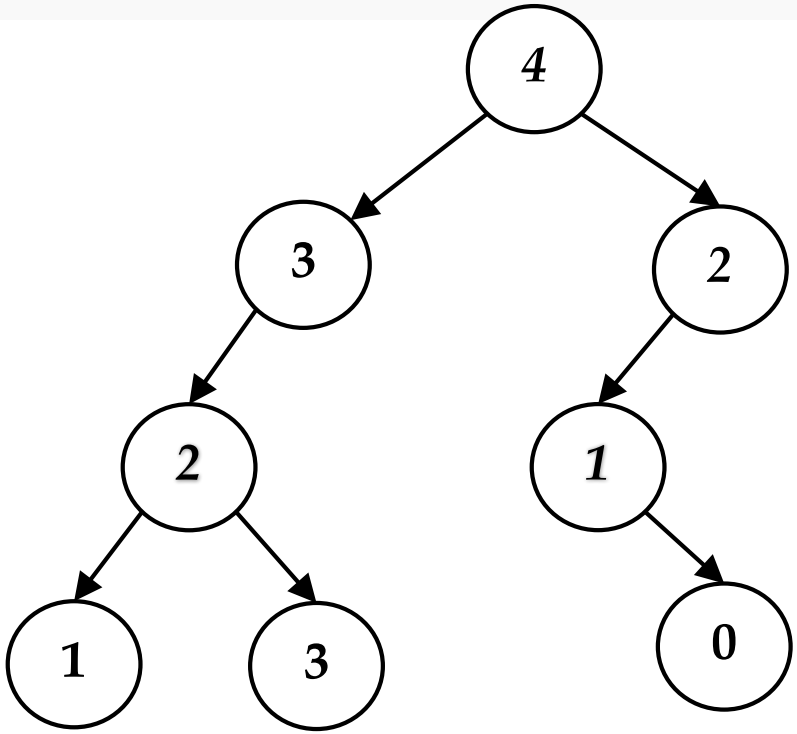
```
L = mkBT(3, mkEmpty(),
         mkBT(2, mkLf(1), mkLf(3))
R = mkBT(2, mkBT(1, mkEmpty(), mkLf(0))
B = mkBT(4, L, R)
```



Scrittura "lineare" di alberi

Supponendo di avere semplicemente un albero di interi, vediamo dapprima una **rappresentazione lineare** di un albero che permette ad esempio di leggerlo o scriverlo come una **sequenza di caratteri**.

Possiamo rappresentare un albero semplicemente **usando le parentesi**, nella forma ***r*** **(*L*, *R*)**, dove ***r*** è l'**etichetta della radice**, e **ricorsivamente** ***L*** è la **rappresentazione del sottoalbero sinistro** e ***R*** è la **rappresentazione** di quello **destro**.



Ad esempio, albero in figura si può rappresentare come:

4(3(2(1, 3), •), 2(1(•, 0), •)

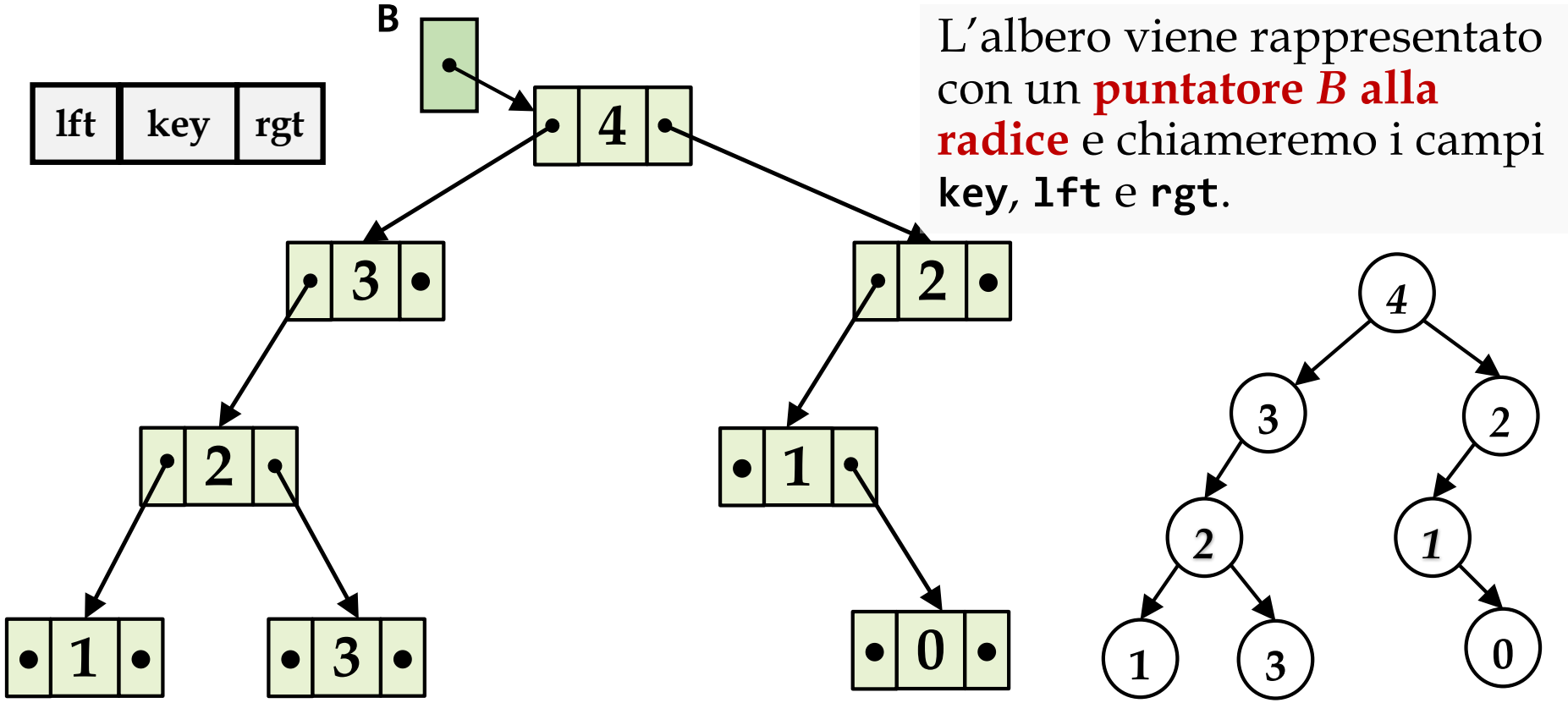
dove • rappresenta l'albero vuoto.

Rappresentaz. a record & puntatori

Dovendo memorizzarlo nella **memoria di un calcolatore**, un albero binario viene spesso rappresentato a **record è puntatori**.

In figura • rappresenta il pointer `NULL`, cioè un sottoalbero vuoto.

Osservate che (concretamente) la **struttura dati è isomorfa alle liste doppiamente concatenate**, ma è diverso l'uso che si fa dei puntatori.

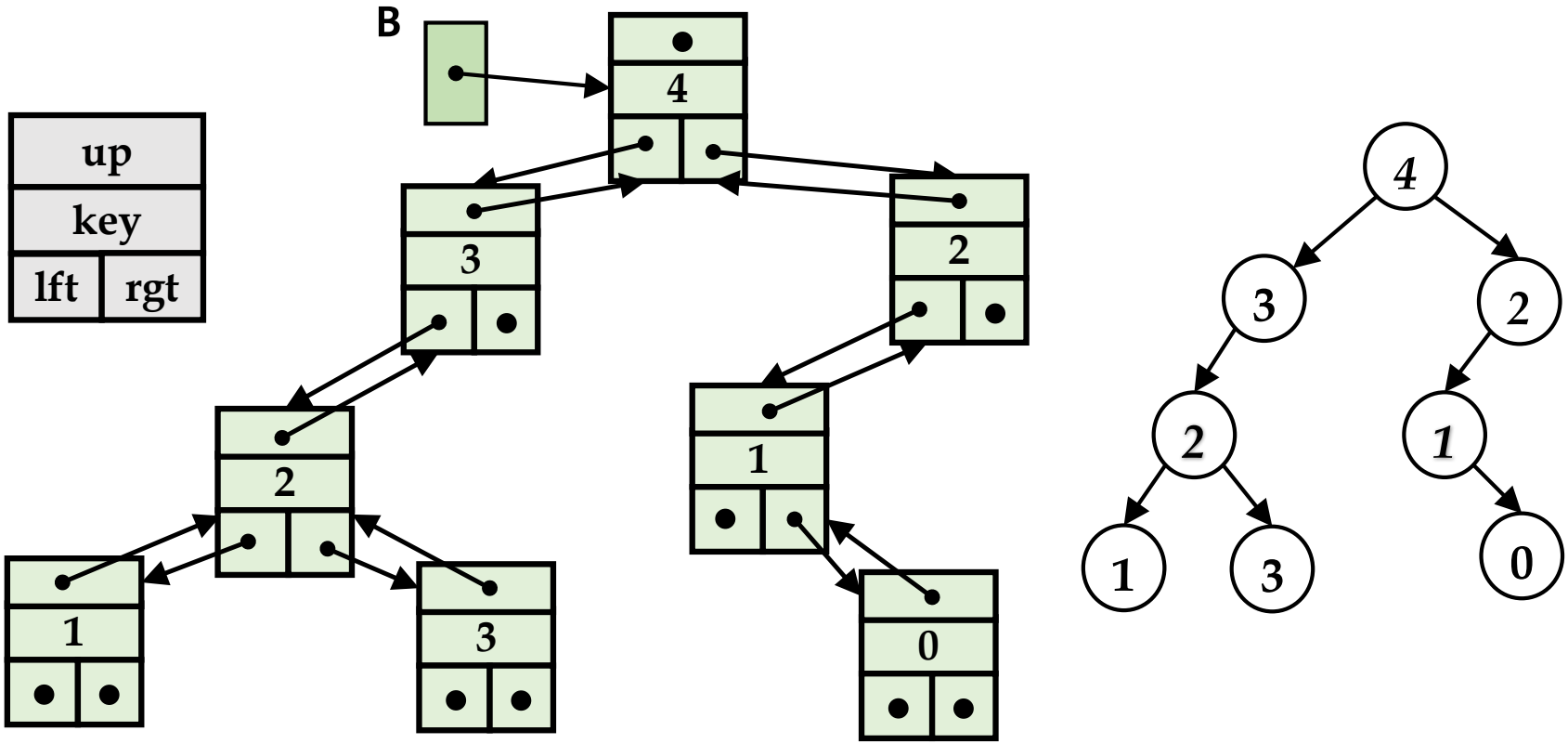


L'albero viene rappresentato con un **puntatore *B* alla radice** e chiameremo i campi `key`, `lft` e `rgt`.

Puntatore parent (up)

Esattamente come nelle liste, potrebbe essere comodo avere anche un **puntatore al padre** (chiamato usualmente **parent** o **up**. Questo è abbastanza standard negli **alberi binari di ricerca** (prossimamente).

Al solito, questo presenta lo **svantaggio** di un ulteriore **overhead di memoria** dovuto al puntatore in più, e a rendere leggermente **più laboriose** operazioni come **inserimento** e **cancellazione**.

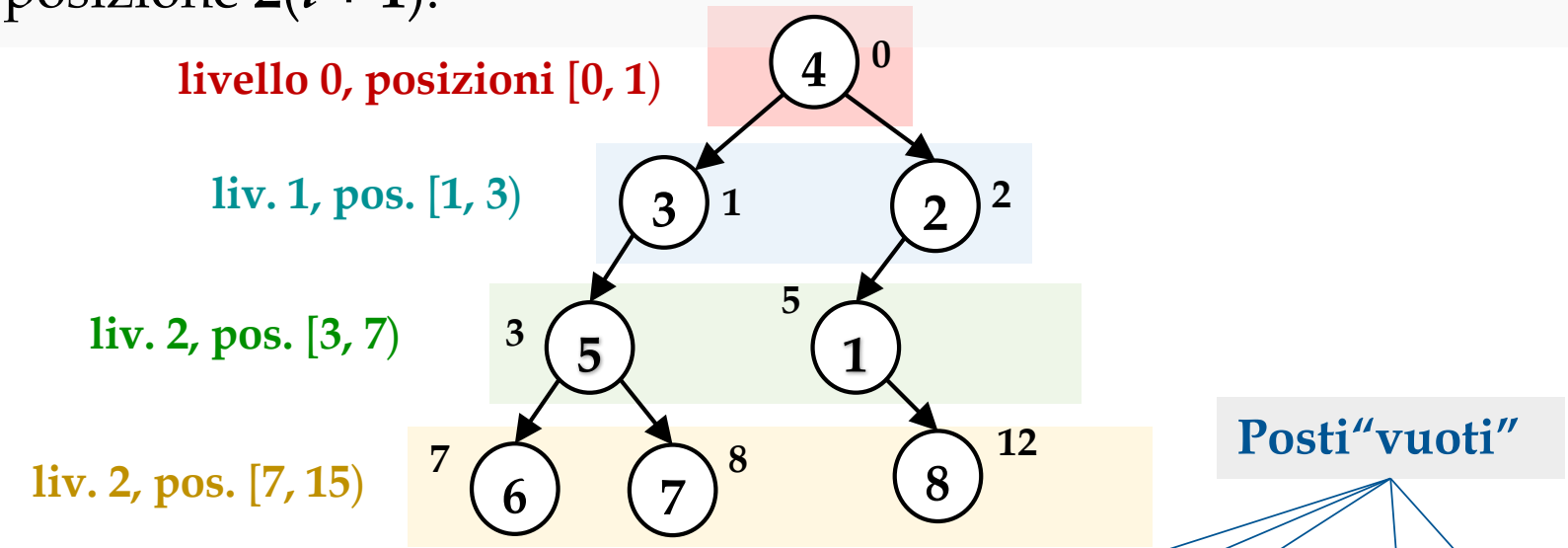


Rappresentazione posizionale

Anche gli **alberi binari** si possono rappresentare con vettori.

Un vettore di altezza h si può rappresentare in un vettore di lunghezza $2^{h+1} - 1$.

La **radice** è in **posizione 0**, e per ogni generico nodo memorizzato in posizione i, il **figlio sinistro** in posizione $2(i + 1) - 1$ e il figlio **destro** in posizione $2(i + 1)$.



4	3	2	5	•	1	•	6	7	•	•	•	8	•	•
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Primi confronti

La rappresentazione a **record è puntatori**:

- ha la consueta **flessibilità** delle strutture dinamiche;
- ha un **overhead di memorizzazione** dovuto ai **puntatori** (ne servono $2n$ per un albero di n nodi e di questi, esattamente $n+1$ sono NULL).

La **rappresentazione posizionale**:

- ha uno **spreco di memoria notevole** quando gli **alberi non sono completi** (o **quasi completi**). Se l'altezza è h, necessita di avere un vettore di almeno $2^{h+1}-1$ elementi.
- In particolare, l'**albero** degenere (**filiforme**) ha **solo h nodi**.
- È **problematica** quando **non è noto** fin dal momento dell'**allocazione** quale sia l'**altezza massima** dell'albero.

➧ **Osservazione**: operazioni come `makeBT` e `makeLf` non hanno una naturale implementazione con la rappresentazione posizionale.

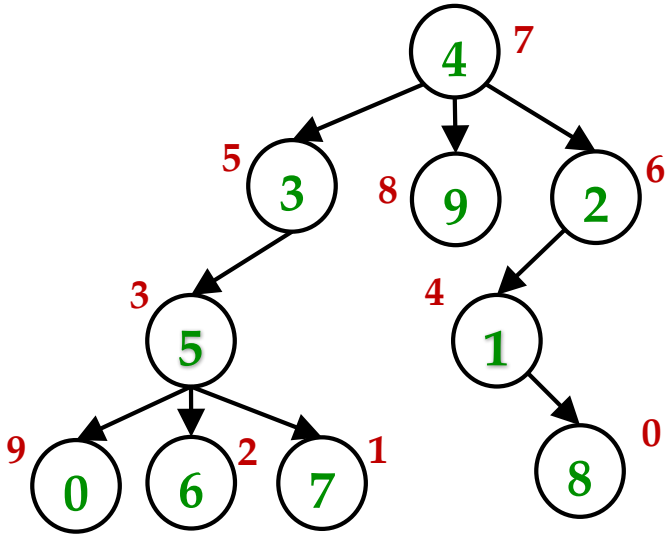
In tal caso, gli alberi si costruiscono allocando lo spazio necessario e iterando operazioni del tipo `insert(B, x)`.

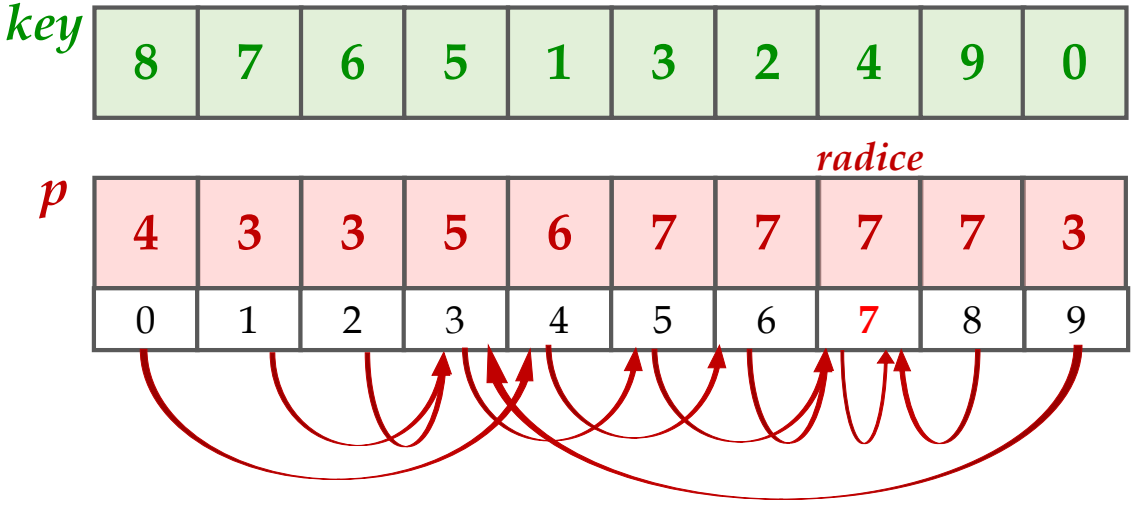
Vettore dei padri

In un albero **un nodo** può avere un numero variabili di figli, ma **ha** sempre **un solo padre**. Ciò suggerisce la seguente codifica, che risulta molto comoda in alcune applicazione (la vedremo nelle **visite di grafi**) e che permette di rappresentare **alberi anche non binari**.

Un albero di n nodi viene rappresentato da **due vettori** di n **celle**, che chiameremo *key* e p. Ogni nodo viene rappresentato (implicitamente) da un naturale (arbitrario) nell'intervallo $[0, n)$.

- $key[i]$ è la chiave del nodo i
- $p[i]$ è l'indice del padre. Se i è l'indice della radice, poniamo $p[i] = i$ per riconoscere l'unico nodo senza padre.





Confronto tra Rappresentazioni (I)

La struttura dati che permette di rappresentare in **modo naturale** il **tipo di dato albero** è senz'altro quella **a record e puntatori** (**R&P**)

Tuttavia gli alberi servono a molte cose, e a volte **risultano comode le altre** rappresentazioni:

- rappresentazione **posizionale** nell'implementazione delle **code con priorità** e nell'**algoritmo di ordinamento** ottimo **heapSort**
- **vettore dei padri** nelle **visite di grafi**.

Operazioni di **interrogazione** come sotto-albero destro, sinistro, radice sono ugualmente facili con R&P e Posiz., mentre **richiedono di scorrere tutto il vettore** nel caso di vettore dei padri.

```
def right(B):
# posizionale, REQ: B≠EMPTY
    return 2*(B+1)
```

```
def left(B):
# R&P, REQ: B≠EMPTY
    return lft[B]
```

```
def isLeaf(B):
# a R&P e posizionale
    if B ≠ EMPTY:
        if left(B)==EMPTY
            and right(B)==EMPTY:
            return True
    return False
```

Rappresentazione di alberi k-ari

Col vettore dei padri si rappresentano senza problemi alberi con un numero arbitrario di figli. Essendo una rappresentazione a vettore, è efficace soprattutto **quando è noto il numero di nodi in anticipo**.

Alberi *k*-ari (con *k* fissato) si possono rappresentare sostituendo i puntatori **`lft`** e **`rgt`** con un **vettore lungo *k* di puntatori** ai figli.

Se non c'è un *k* fissato a priori, è sempre possibile fare una **lista di sottoalberi** (che in letteratura viene a volte chiamata **foresta**).

Un **albero *k*-ario** con $k > 2$ **fissato o meno** può essere **codificato** con un **albero binario**: si dice rappresentazione figlio-fratello: il pointer **`lft`** punta al primo figlio, il pointer **`rgt`** al fratello.