ALBERI

Struttura dati astratta Albero

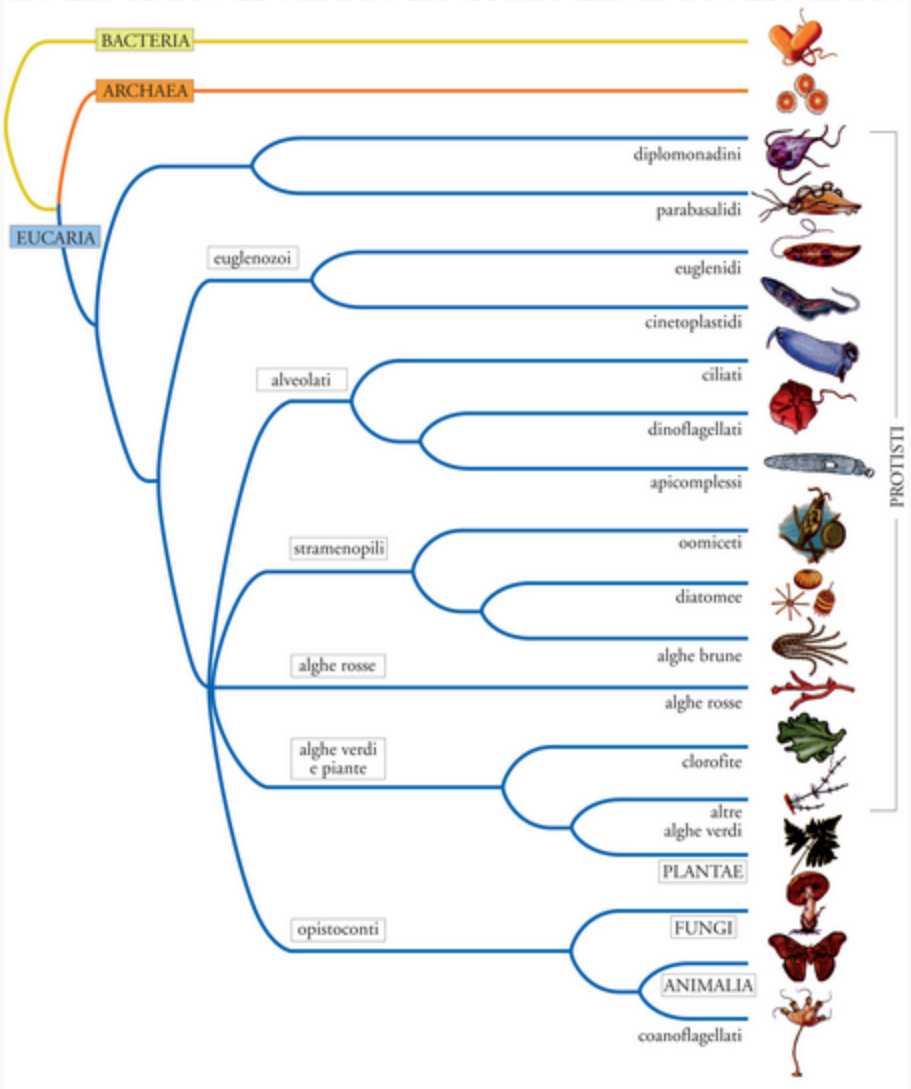
Le liste risolvono le carenze dei vettori (dimensione statica) ma sono strutture dati **sequenziali** e quindi le operazioni sui suoi elementi implicano sempre un accesso sequenziale di costo O(n), insostenibile per grandi moli di dati.

Strutture **non sequenziali** possono rendere nettamente più efficienti gli algoritmi di gestione degli elementi.

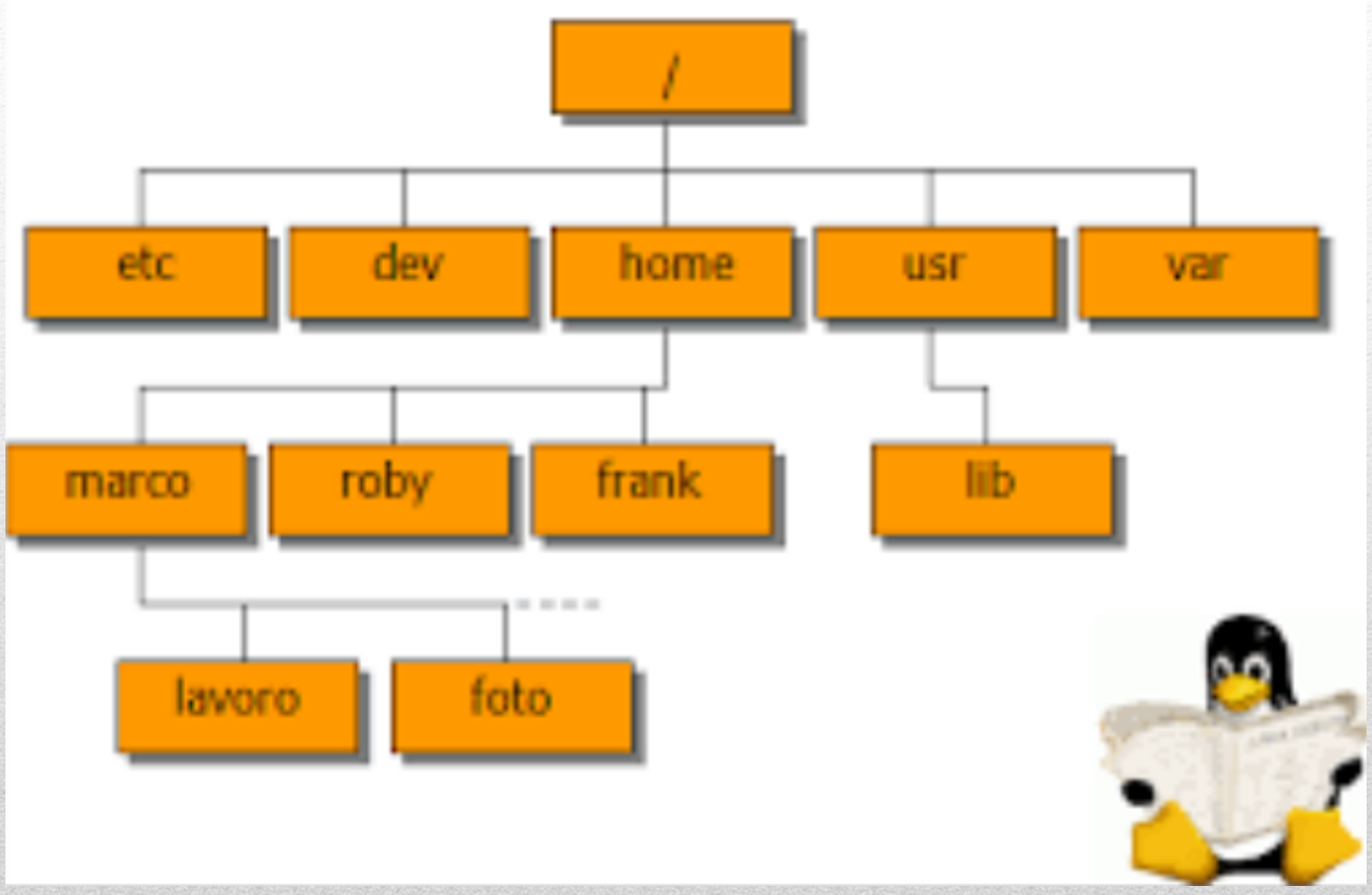
Gli **alberi** sono il caso più usato e rilevante di strutture dati non sequenziali.

Inoltre, su alcune collezioni di oggetti sono definite in modo naturale relazioni gerarchiche. Ciò avviene ad esempio per la successione delle chiamate ricorsive della funzione Fibonacci(n) e negli esempi dei lucidi successivi.

Struttura dati astratta Albero



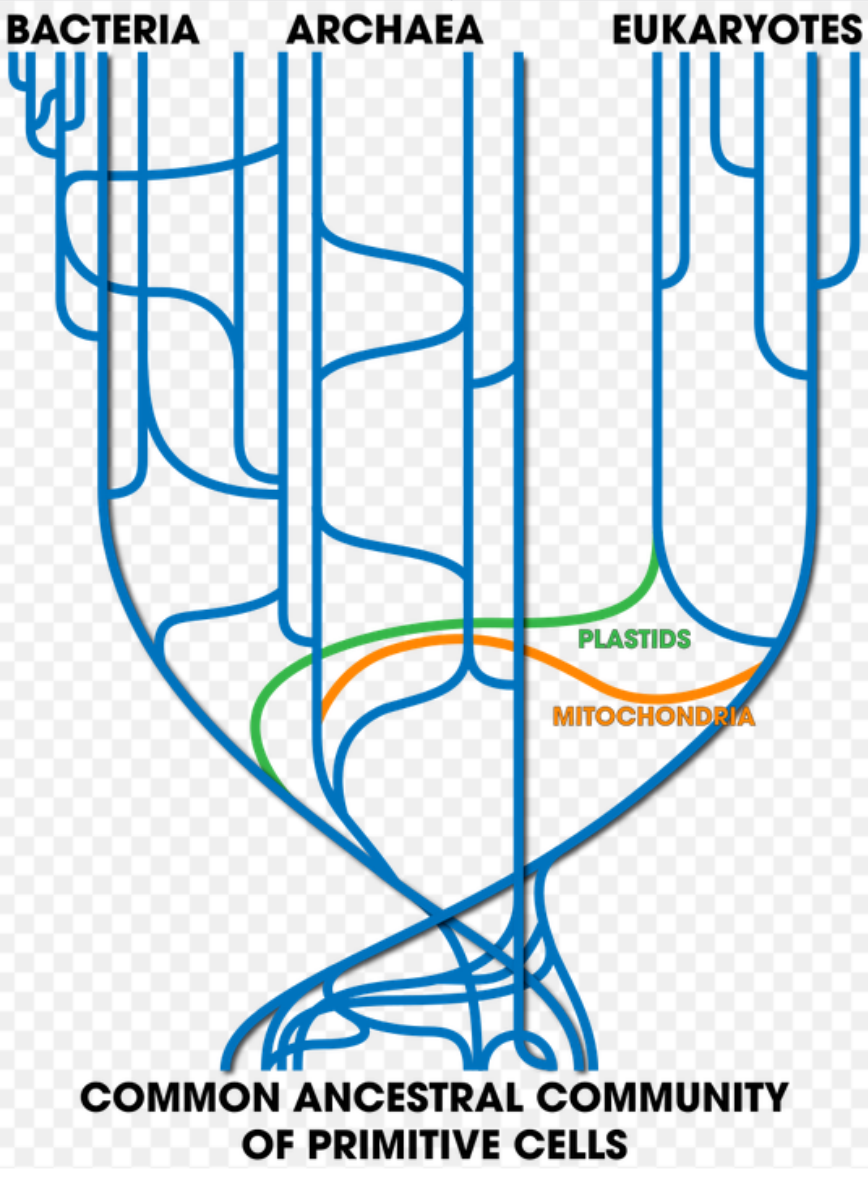
Struttura dati astratta Albero



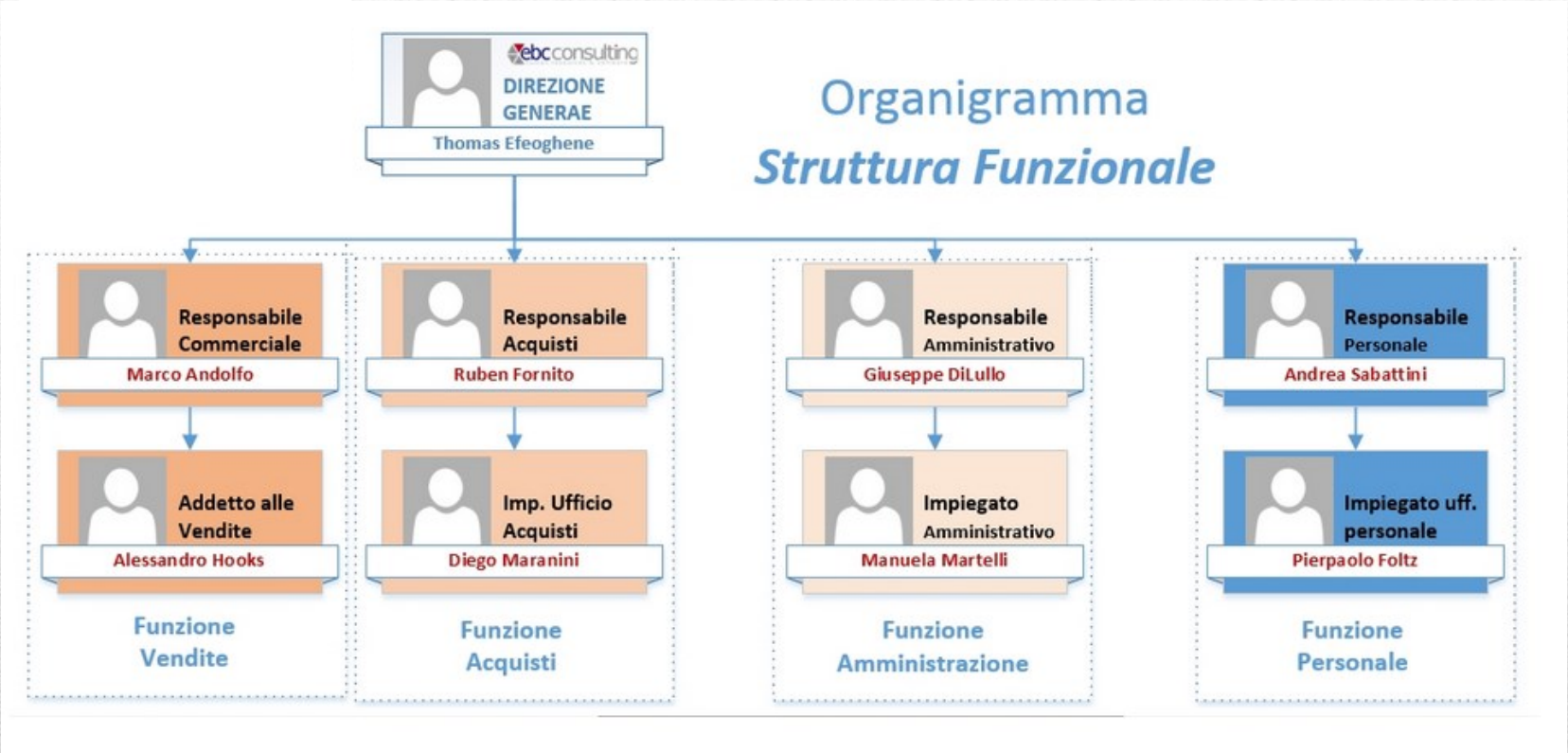
Struttura dati astratta Albero

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>La sezione 'body' della pagina HTML</title>
5    </head>
6    <body>
7      <p>
8        Questa e' la sezione body della pagina HTML,
9        delimitata dai tag di apertura e chiusura omonimi.
10     </p>
11     <p>
12       Quello sopra e' un paragrafo,
13       e io sono un secondo paragrafo.
14     </p>
15   </body>
16 </html>
```

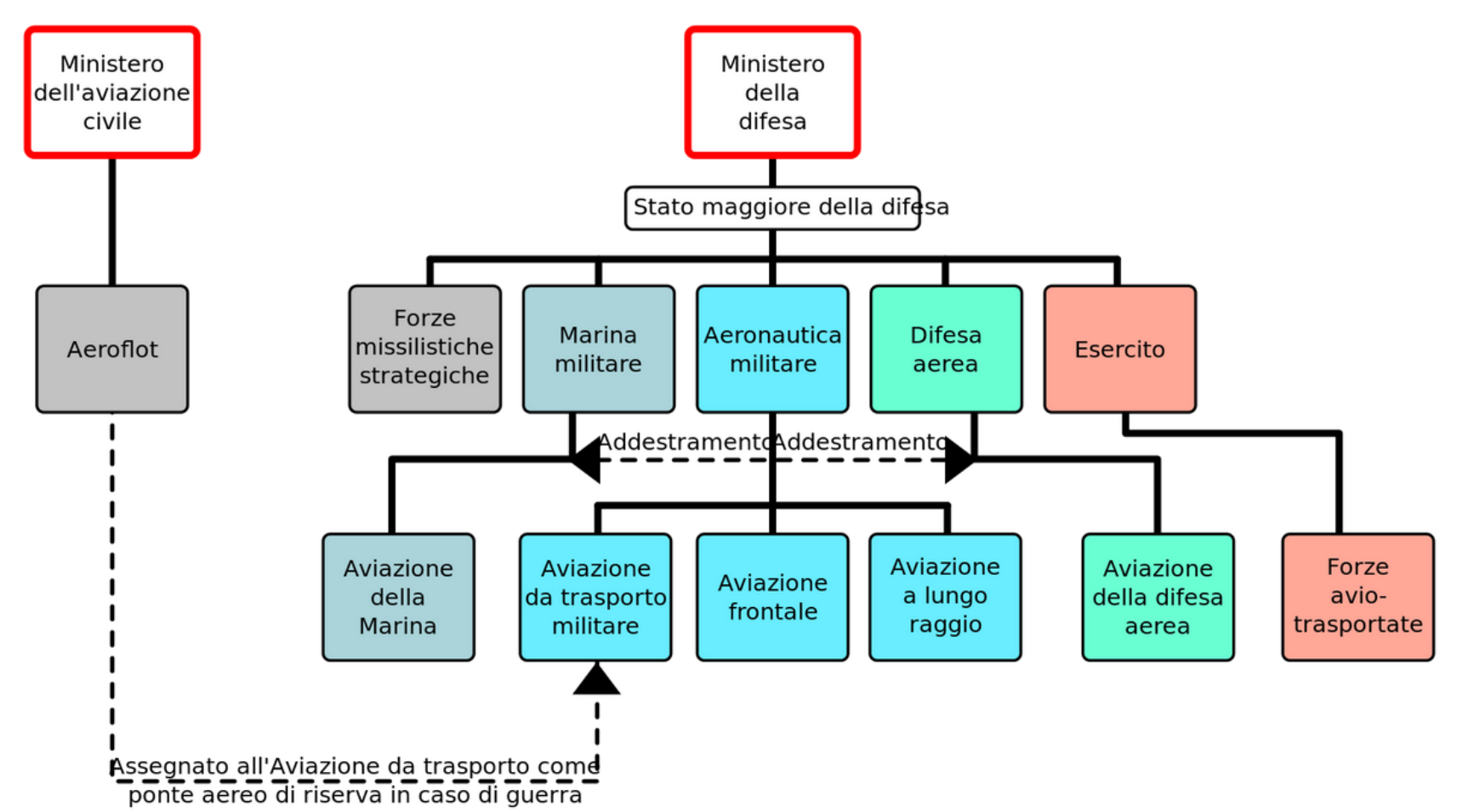
Struttura dati astratta Albero



Struttura dati astratta Albero



Struttura dati astratta Albero



Definizione Albero

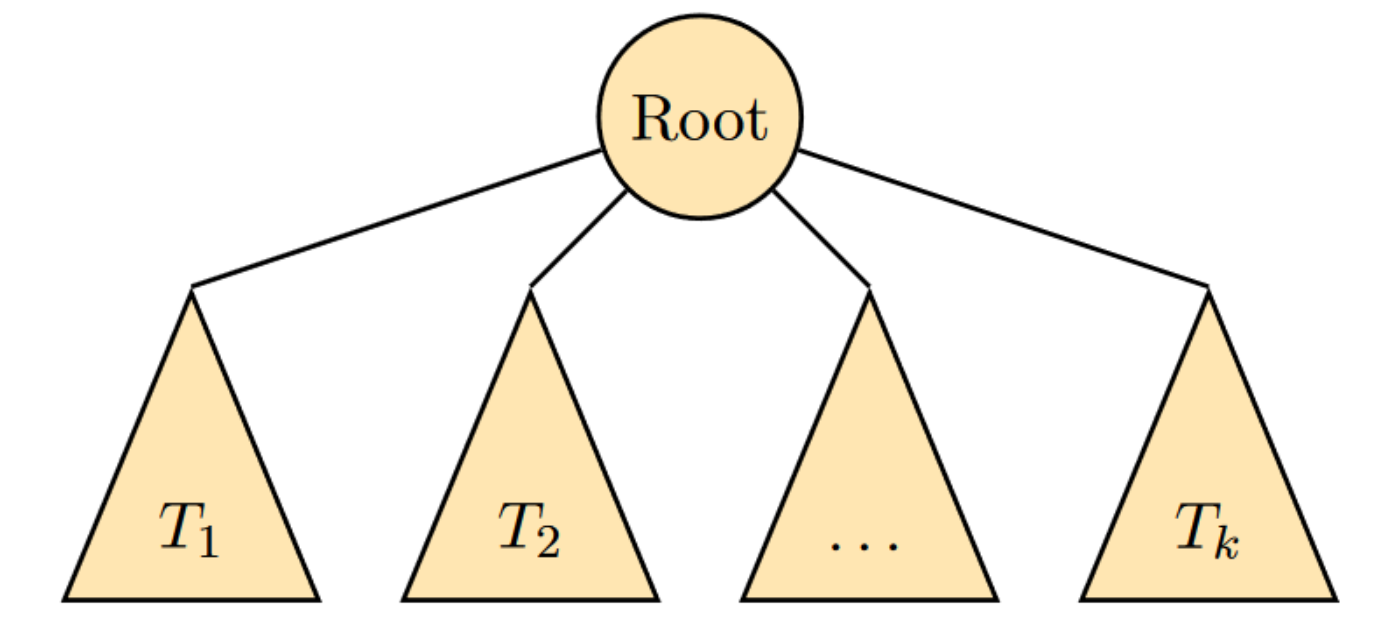
Un albero è un insieme di elementi chiamati **nodi** e un insieme di **relazioni** tra i nodi stessi tali che:

1. Un albero che non contiene nodi si dice **vuoto**
2. Se l'albero non è vuoto, tra i nodi ne esiste uno che prende il nome di **radice**; tale elemento deve esistere necessariamente e deve essere **unico**
3. Per tutti i nodi dell'albero tranne la radice è sempre possibile individuare **uno e un solo** nodo **genitore**.
4. L'albero è **connesso**, nel senso che preso un qualunque nodo tranne la radice e procedendo verso i suoi genitori, si arriva sempre alla radice.

Definizione ricorsiva di Albero

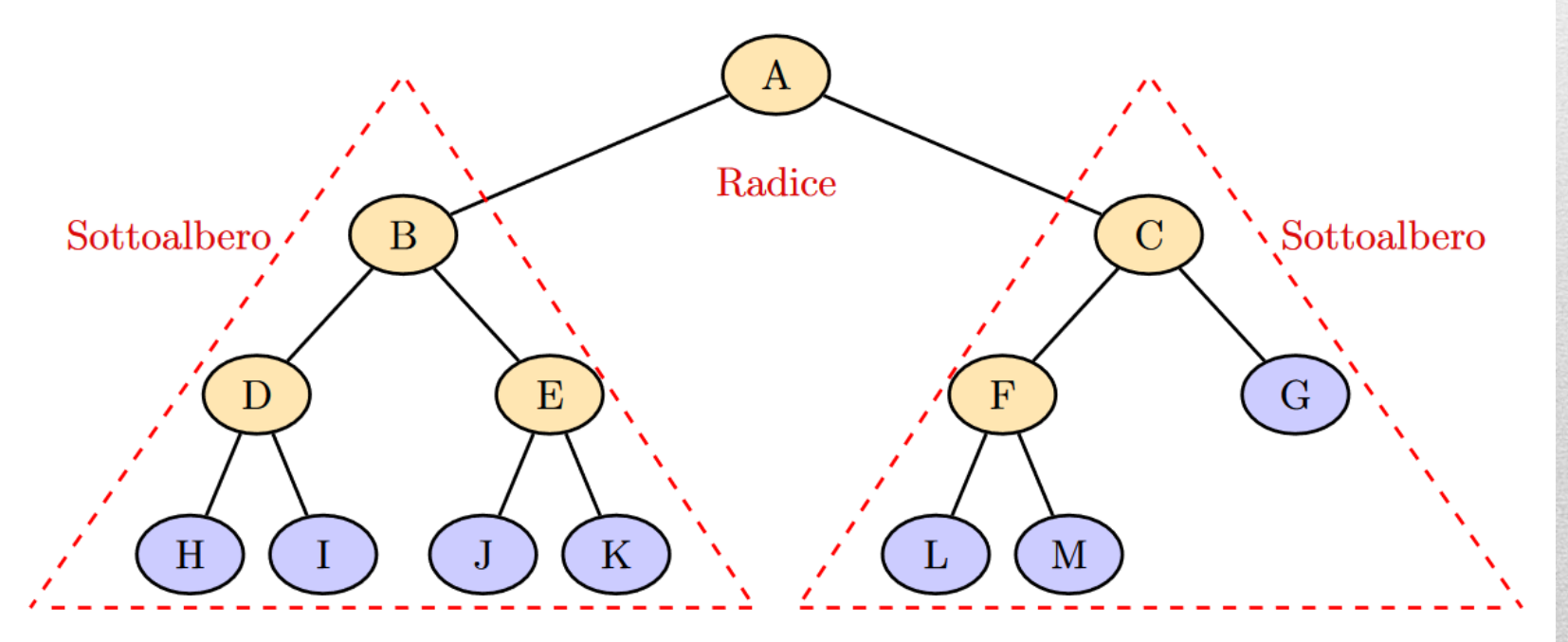
Un albero è dato da:

1. un insieme **vuoto**, oppure
2. un nodo **radice** e zero o più **sottoalberi**, ognuno dei quali è un **albero**; la radice è connessa alla radice di ogni sottoalbero con un arco orientato.

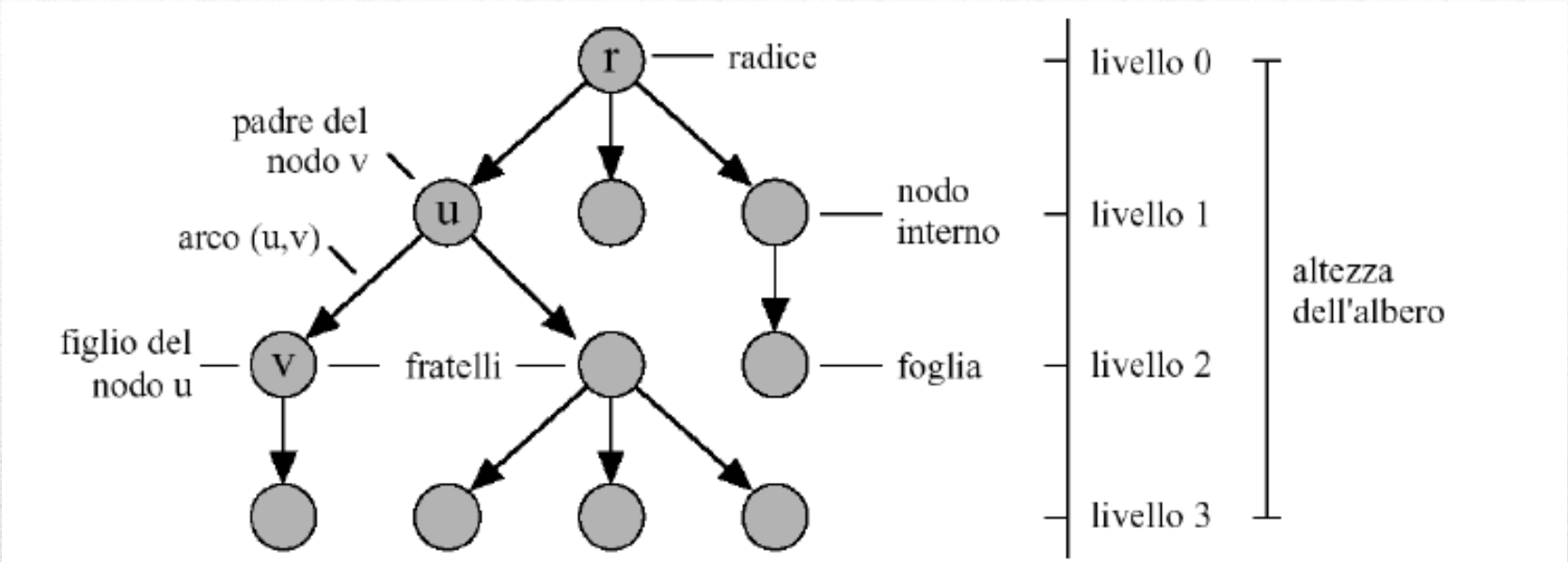


Terminologia

- A è la **radice**
- B, C sono radici dei sottoalberi
- D, E sono **fratelli**
- D, E sono **figli** di B
- B è il **padre** di D, E
- I nodi viola sono **foglie**
- Gli altri nodi sono **nodi interni**.



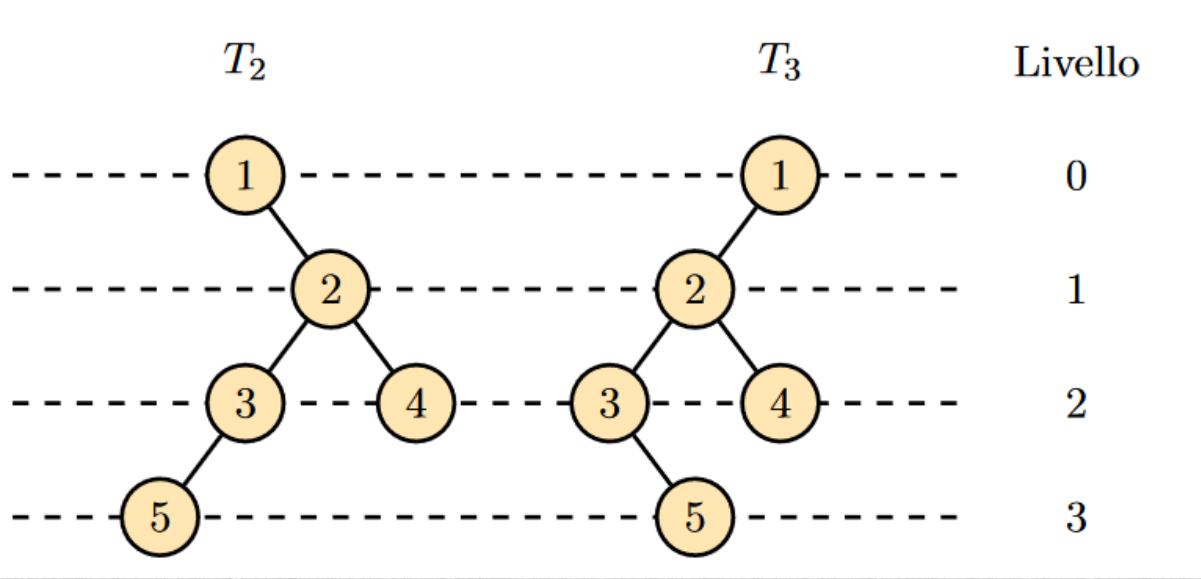
Terminologia



Profondità o livello di un nodo: la lunghezza del cammino semplice dalla radice al nodo (misurato in numero di archi)
Livello: l'insieme di nodi alla stessa profondità
Altezza dell'albero: la profondità massima delle sue foglie

Albero binario

Un **albero binario** è un albero radicato in cui ogni nodo ha al massimo due figli, identificati come **figlio sinistro** e **figlio destro**.
Nota: Due alberi T e U che hanno gli stessi nodi, gli stessi figli per ogni nodo e la stessa radice, sono distinti qualora un nodo u sia designato come figlio sinistro di v in T e come figlio destro di v in U.



Dati e operazioni su alberi

Tipo di **dato astratto** Albero. In base all'implementazione concreta che si sceglierà, le operazioni saranno realizzate in maniera più o meno efficiente.

tipo `Albero`:
dati:
un insieme di nodi (di tipo *nodo*) e un insieme di archi.

operazioni:

`numNodi`() → *intero*
restituisce il numero di nodi presenti nell'albero.

`grado`(*nodo* v) → *intero*
restituisce il numero di figli del nodo v.

`padre`(*nodo* v) → *nodo*
restituisce il padre del nodo v nell'albero, o `null` se v è la radice.

`figli`(*nodo* v) → $\langle$*nodo*, *nodo*, ..., *nodo*$\rangle$
restituisce, uno dopo l'altro, i figli del nodo v.

`aggiungiNodo`(*nodo* u) → *nodo*
inserisce un nuovo nodo v come figlio di u nell'albero e lo restituisce. Se v è il primo nodo ad essere inserito nell'albero, esso diventa la radice (e u viene ignorato).

`aggiungiSottoalbero`(`Albero` a, *nodo* u)
inserisce nell'albero il sottoalbero a in modo che la radice di a diventi figlia di u.

`rimuoviSottoalbero`(*nodo* v) → `Albero`
stacca e restituisce l'intero sottoalbero radicato in v. L'operazione cancella dall'albero il nodo v e tutti i suoi discendenti.

Operazioni elementari sugli alberi

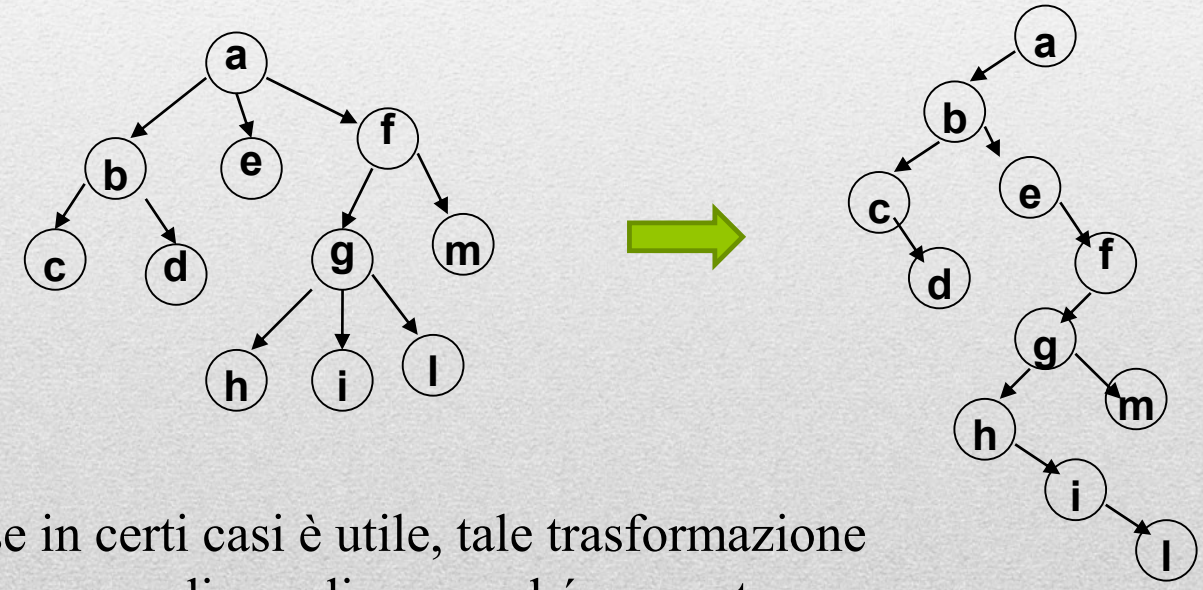
Operazioni elementari che possiamo implementare sugli alberi (in questo caso se binari, ma possono essere generalizzate):

- **Vuoto**: fa il test di albero vuoto;
- **Accedi**: recupera il valore contenuto nel nodo radice;
- **FiglioSx**: restituisce il sottoalbero sinistro;
- **FiglioDx**: restituisce il sottoalbero destro.
- **Genitore**: restituisce il sottoalbero di cui il nodo dato è figlio;
- **InserisciSottoAlberoSx**: inserisce un sottoalbero sinistro;
- **InserisciSottoAlberoDx**: inserisce un sottoalbero destro;
- **CancellaSottoAlberoSx**: cancella un sottoalbero sinistro;
- **CancellaSottoAlberoDx**: cancella un sottoalbero destro;
- **CreaFoglia**: crea un nuovo nodo senza figli.
- **CancellaFoglia**: dealloca un nodo.

Trasformazione di alberi

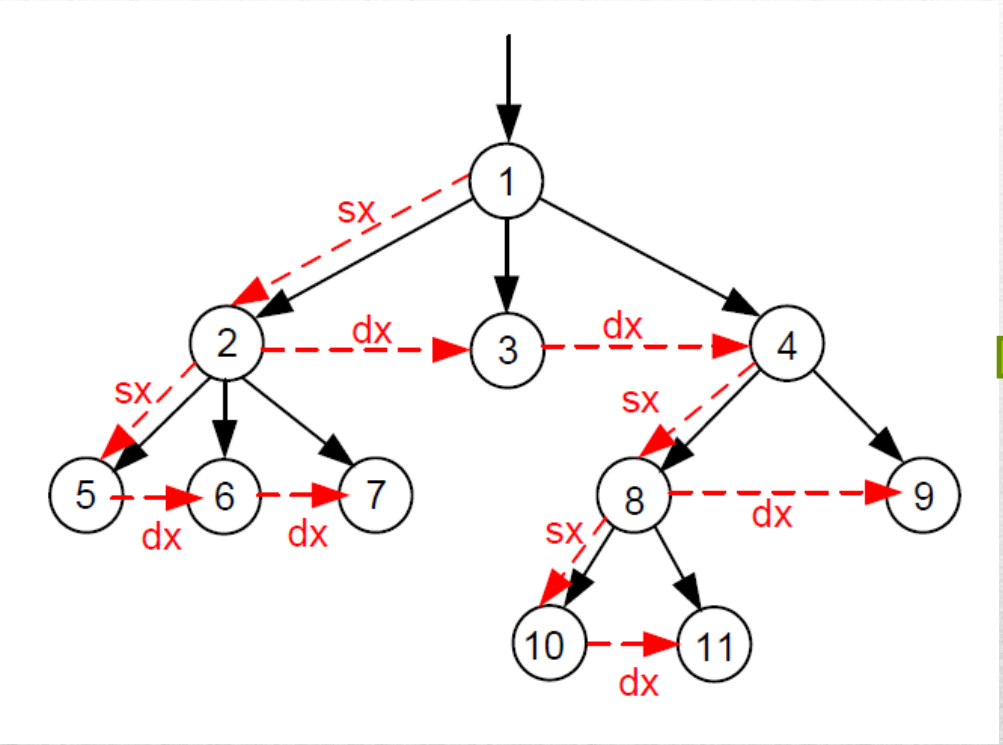
Da un albero ordinato (da sinistra verso destra) A di n nodi è possibile ricavare un equivalente albero **binario** B di n nodi con la regola:

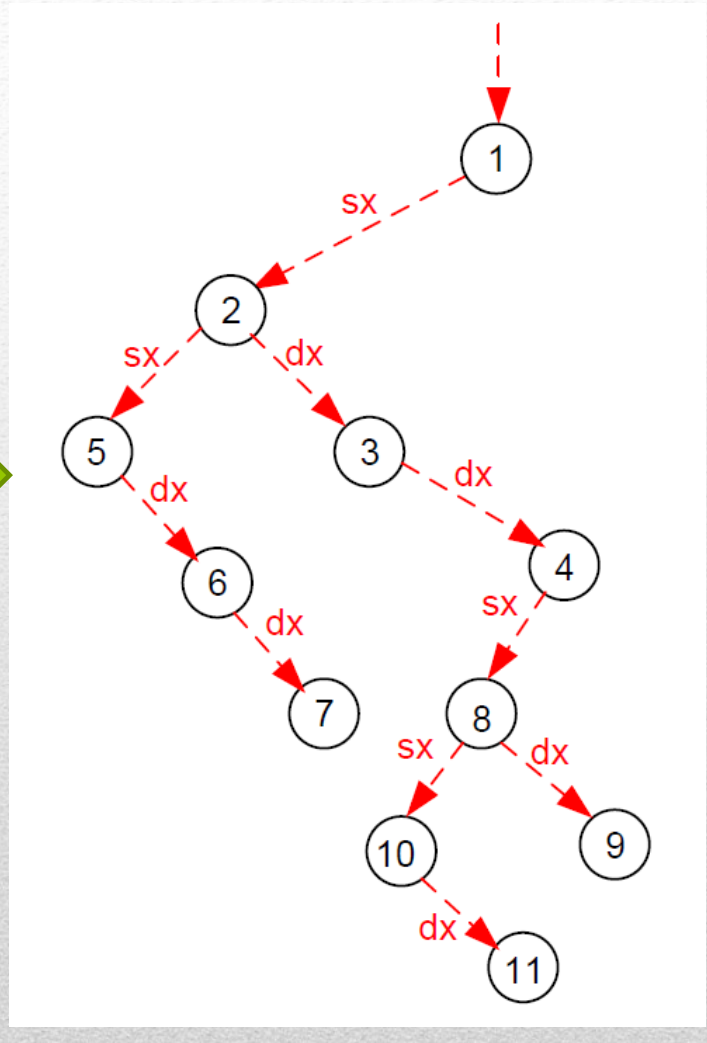
- La radice di A coincide con la radice di B;
- Ogni nodo b di B ha come radice del sottoalbero sinistro il primo figlio di b in A e come sottoalbero destro il fratello successivo di b in A.



Anche se in certi casi è utile, tale trasformazione potrebbe essere dispendiosa perché aumenta l'altezza dell'albero.

Trasformazione di alberi





Rappresentazioni indicizzate di alberi

Ogni cella dell'**array** contiene

- le informazioni di un nodo
- eventualmente gli indici per raggiungere altri nodi

Spazio O(n) per albero con n nodi.

1 - Vettore dei padri

Sia T=(N,A) un albero con n nodi numerati da 0 a (n-1)

Per un albero con n nodi usiamo un array P di dimensione n

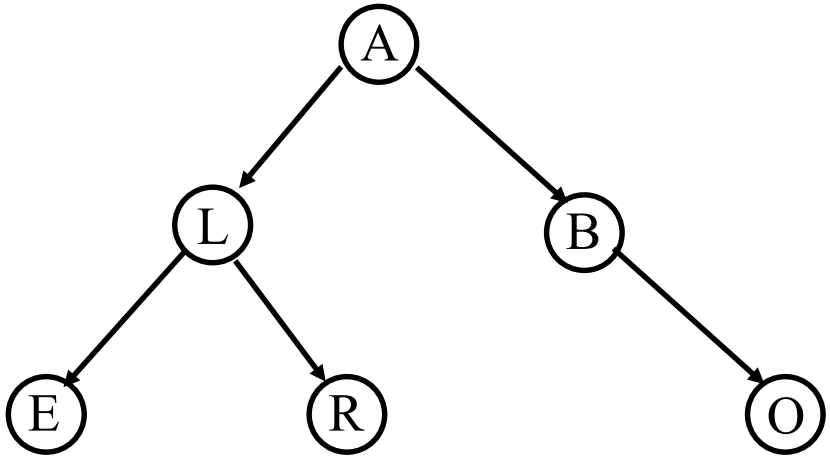
Una generica cella i contiene una coppia (info, parent), dove:

- info: contenuto informativo del nodo i
- parent: indice (nell'array) del nodo padre di i

E' possibile risalire in tempo O(1) al padre di ciascun nodo, mentre trovare un figlio richiede una scansione dell'array in tempo O(n).

2 - Vettore posizionale (per alberi d-ari (quasi) completi)

Vettore dei padri



P

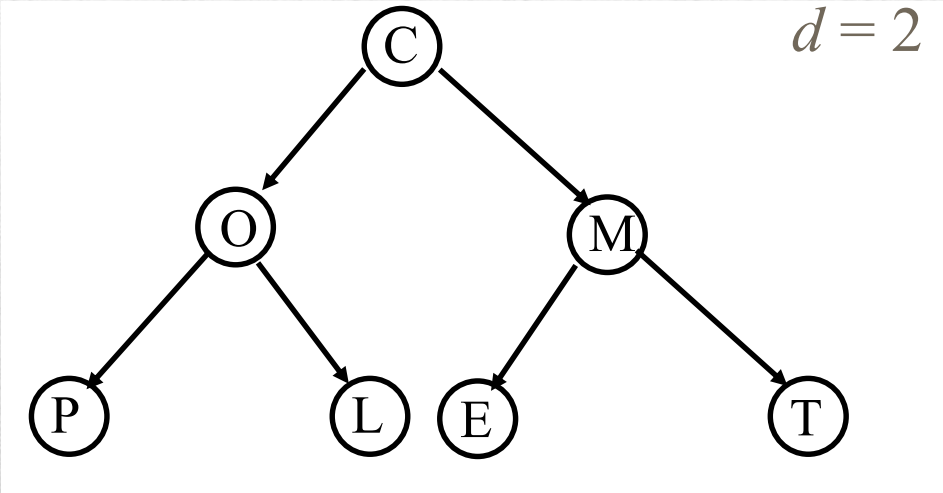
(L,2)	(B,2)	(A,null)	(O,1)	(E,0)	(R,0)
0	1	2	3	4	5

P[i].info: contenuto informativo nodo

P[i].parent: indice del nodo padre

Vettore posizionale

- nodi organizzati nell'array "per livelli"
- un array P di dimensione n tale che P[v] contiene l'informazione di v
- l'informazione dell'i-esimo figlio ($i \in \{0,\ldots,d-1\}$) di v è in posizione $P[d \cdot v+i]$
- il padre di i è in posizione $\lfloor v/d \rfloor$



Per alberi d-ari completi:

Tutti i nodi tranne le foglie hanno grado d

tutte le foglie sono sullo stesso livello

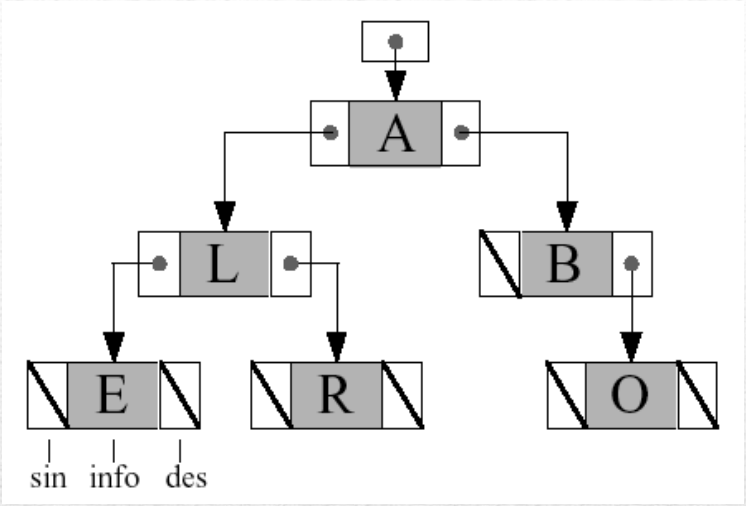
Dato un nodo *v* in un albero binario completo, il figlio **sinistro** sarà in posizione $2v$ e quello **destro** in posizione $2v+1$.

P

C	O	M	P	L	E	T
1	2	3	4	5	6	7

Da ogni nodo v si risale in tempo O(1) sia al padre (indice $\lfloor v/d \rfloor$ se v non è la radice) che a uno qualunque dei figli

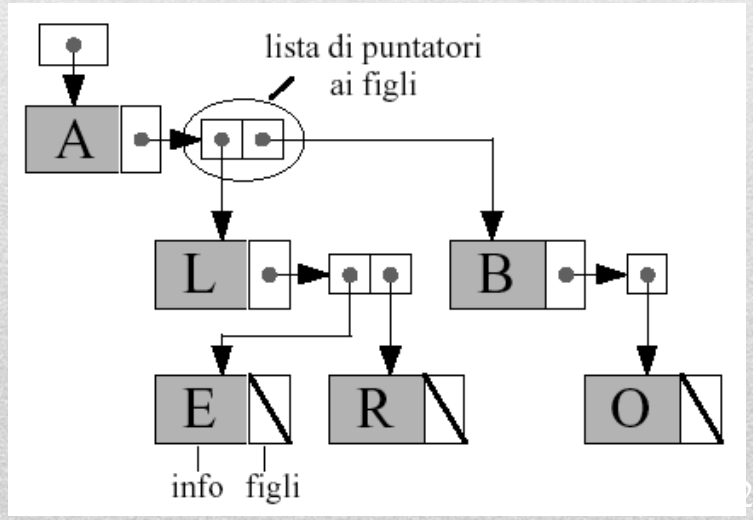
Rappresentazioni collegate di alberi



Rappresentazione con **puntatori ai figli** (nodi con numero limitato di figli). Lo spazio richiesto sarà *O(n·d)* che per *d* costante è *O(n)*

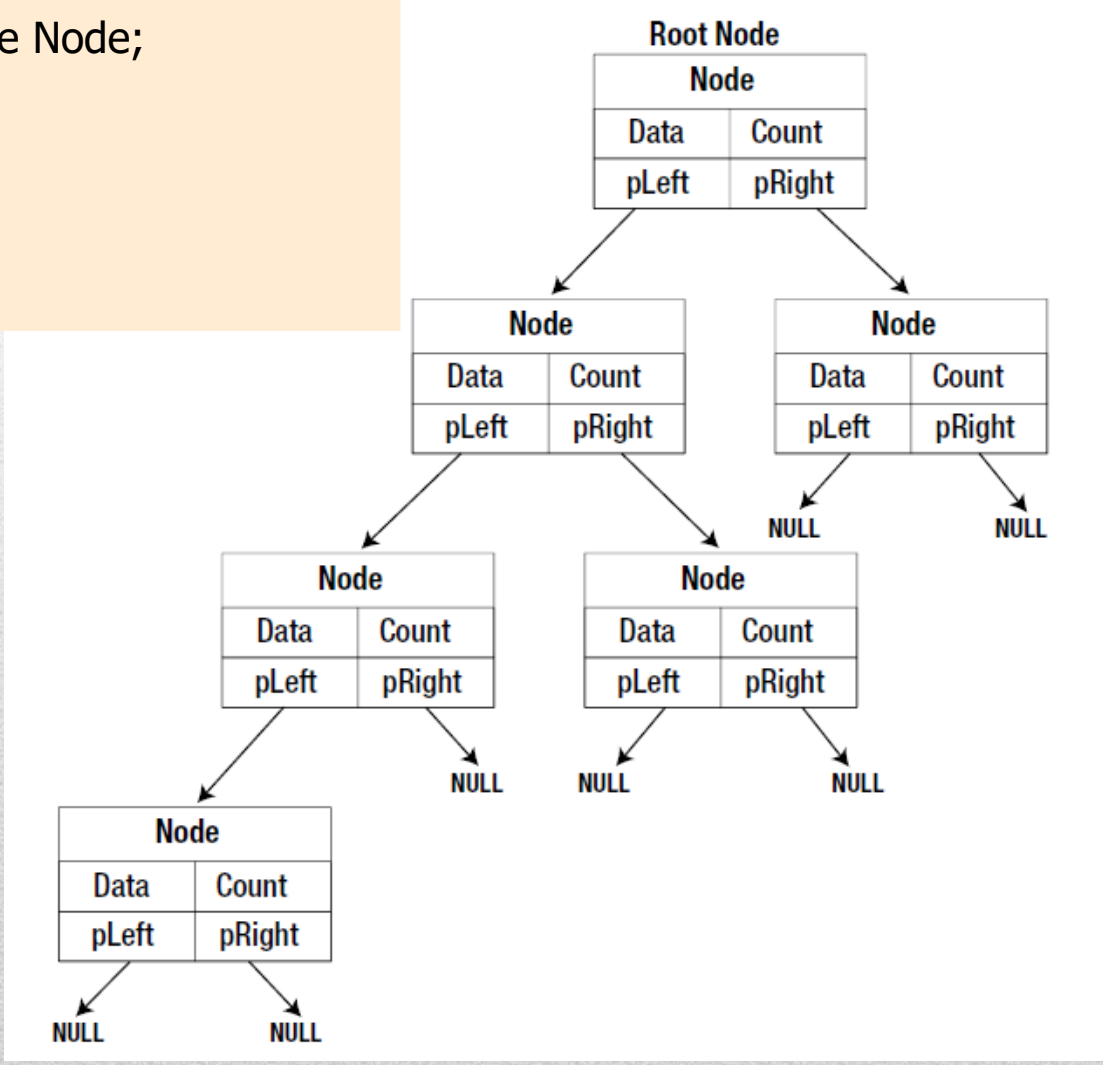
```
struct treeNode {
   int data;
   struct treeNode *pLeft;
   struct treeNode *pRight;
};
```

Rappresentazione con **liste di puntatori ai figli** (nodi con numero arbitrario di figli). La lista può essere rappresentata in modo indicizzato o collegato.
Lo spazio richiesto per rappresentare un albero con n nodi sarà *O(n)*.



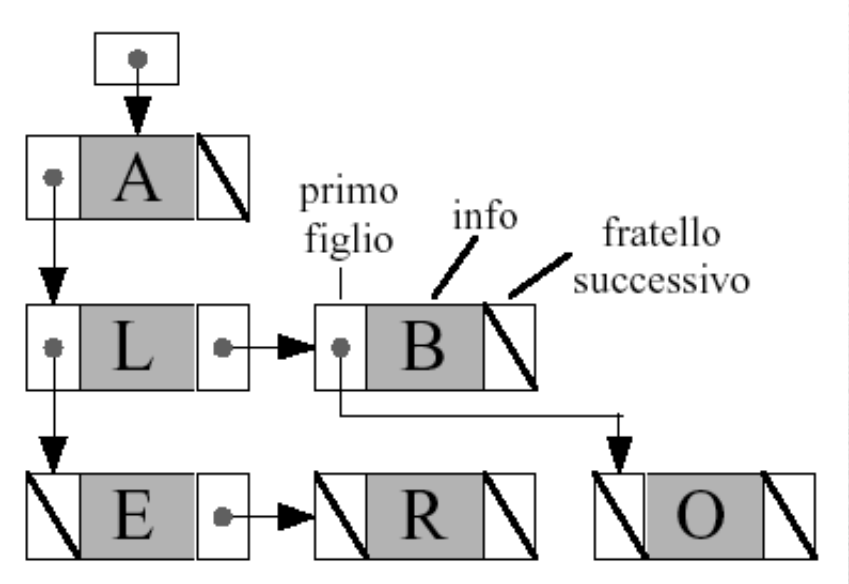
Rappresentazioni collegate di alberi

```
typedef struct Node Node;
struct Node {
   int data;
   Node *pLeft;
   Node *pRight;
};
```



Rappresentazioni collegate di alberi

Rappresentazione
di tipo **primo figlio-fratello successivo** (nodi con numero arbitrario di figli). Non serve una struttura dati addizionale (la lista dei figli) per ogni nodo.
Lo spazio richiesto per rappresentare un albero con n nodi sarà *O(n)*.
Per scandire tutti i figli di un nodo servirà scendere al primo figlio e passare a tutti i suoi fratelli.



Tutte le rappresentazioni possono essere arricchite per avere in ogni nodo anche un puntatore al padre, per supportare l'operazione *padre(v)* in tempo costante.

Visite di alberi

La visita di un albero è un'operazione di attraversamento della struttura dell'albero.
Applicazioni:

- scrivere l'albero in output
- ricercare un'informazione in un albero
- confrontare se due alberi contengono le stesse informazioni
- ...

La visita stabilisce un ordinamento lineare dei nodi dell'albero.

Gli algoritmi di visita si distinguono in base al particolare ordine di accesso ai nodi.

Algoritmo di visita generica

visitaGenerica visita il nodo r e tutti i suoi discendenti in un albero

Idea: parto dalla radice e raggiungo gli altri nodi muovendomi lungo gli archi dell'albero (cioè visito un nodo solo dopo aver visitato suo padre).
Non compio "salti" verso nodi non direttamente collegati ad almeno un nodo visitato.

1. visita la radice
2. **scegli** un nodo *v* tra quelli visitati che abbia figli non visitati
3. se non esiste un tale nodo la visita è terminata
4. **scegli** alcuni dei figli di *v* non visitati e visitali

I vari tipi di visita si differenziano per i criteri di scelta usati in 2) e 4).

Algoritmo di visita generica

```
    algoritmo visitaGenerica(nodo r)
1.      S ← {r}
2.      while (S ≠ ∅) do
3.          estrai un nodo u da S
4.          visita il nodo u
5.          S ← S ∪ { figli di u }
```

Richiede tempo O(n) per visitare un albero con n nodi a partire dalla radice.

Algoritmo di visita in profondità

L'algoritmo di visita in profondità (**depth first search**, **DFS**) parte da *r* e procede visitando i nodi di figlio in figlio fino a raggiungere una foglia. Retrocede poi al primo antenato che ha ancora figli non visitati (se esiste) e ripete il procedimento a partire da uno di quei figli.

Criteri di **scelta**:

- scelgo l'ultimo (in ordine di tempo) nodo visitato che abbia qualche figlio non visitato.
- scelgo solo un figlio (es. il più a sx) tra quelli non visitati.

Rappresento l'insieme dei **nodi aperti** (nodi che rappresentano i punti di ramificazione rimasti in sospeso e da cui la visita deve proseguire) S mediante il tipo di dato **Pila**.

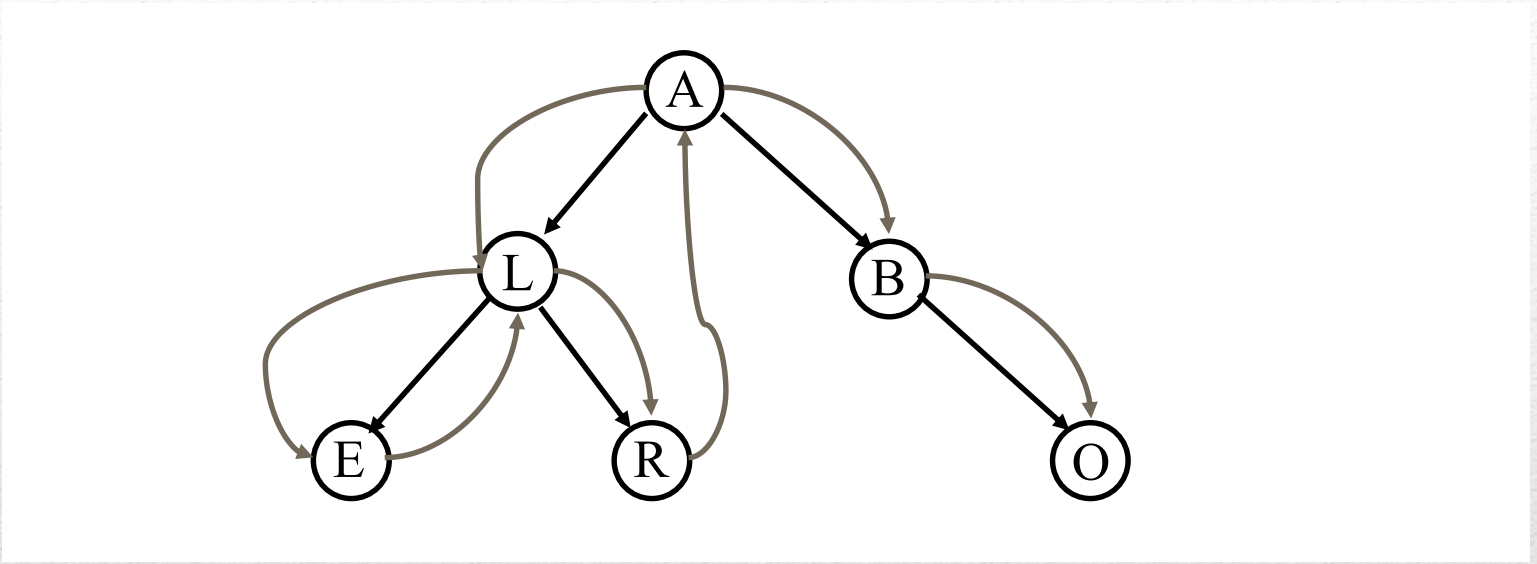
Poiché mettiamo in Pila prima il figlio destro di ogni nodo poi il sinistro, tenderemo a seguire tutti i figli sinistri andando in profondità fino a che non si raggiunge la prima foglia sinistra. In generale, si passerà a visitare ogni sottoalbero destro di un nodo solo quando quello sinistro è stato completamente visitato.

Algoritmo di visita in profondità

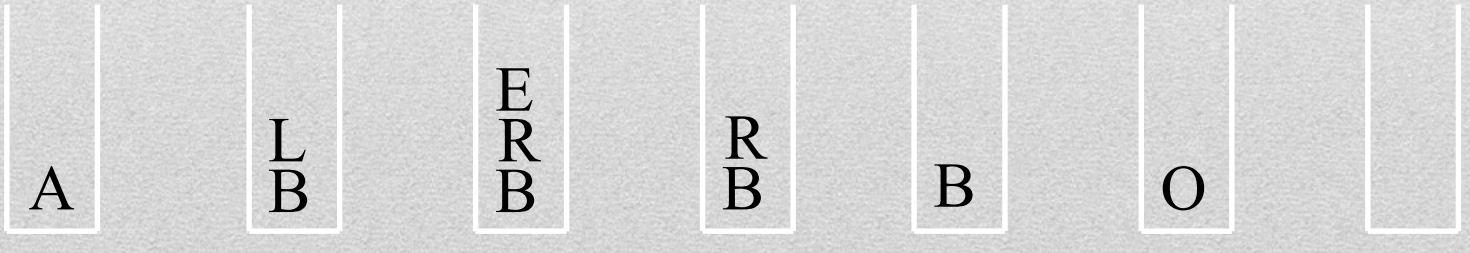
Versione **iterativa** (per alberi binari):

```
void VisitaProfondità(albero t)
{
   pila p=CreaPila();
   Push(t,p);
   albero u;
   while (!PilaVuota(p))
   {
     u = Pop(p);
     if (!Vuoto(FiglioDx(u)))
         Push(FiglioDx(u),p);
      if (!Vuoto(FiglioSx(u)))
         Push(FiglioSx(u),p);
      ;
   }
}
```

Algoritmo di visita in profondità



Ordine di visita: A L E R B O

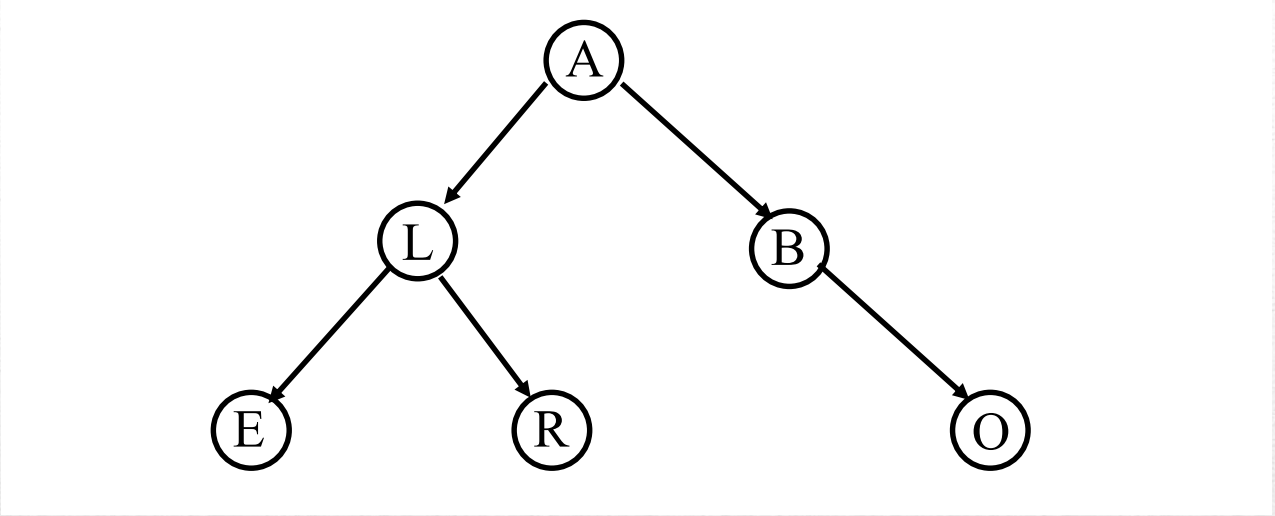


Algoritmo di visita in profondità

Versione **ricorsiva** (per alberi binari):

```
void VisitaDFSPreOrdine(albero t)
{
   if (!Vuoto(t))
   {
      Accedi(t);
      VisitaDFSPreOrdine (FiglioSx(t));
      VisitaDFSPreOrdine (FiglioDx(t));
   }
}
```

La Pila S non appare esplicitamente nell'algoritmo; l'uso della ricorsione permette di usare la pila dei record di attivazione delle chiamate ricorsive per mantenere i nodi aperti.



Visita in **preordine**: radice, sottoalbero sin, sottoalbero destro

Preordine: A L E R B O

Visita **simmetrica**: sottoalbero sin, radice, sottoalbero destro (scambia riga 2 con 3)

Simmetrica: E L R A B O

Visita in **postordine**: sottoalbero sin, sottoalbero destro, radice (sposta riga 2 dopo 4)

Postordine: E R L O B A

Attraversamento degli alberi binari

Sono funzioni ricorsive:

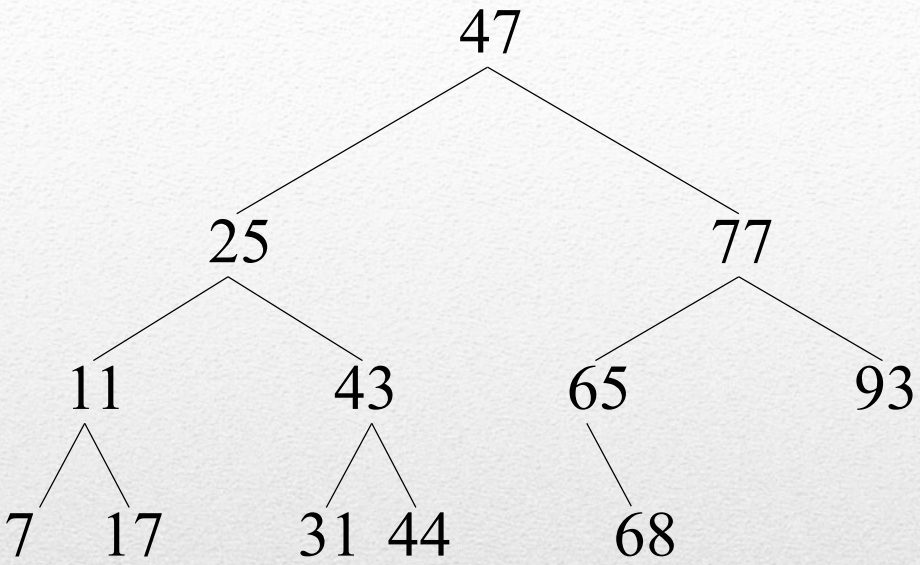
- Attraversamento **simmetrico**:
  1. Si attraversa il sottoalbero sinistro in ordine simmetrico.
  2. Si elabora il valore del nodo (ad es., si stampa).
  3. Si attraversa il sottoalbero destro in ordine simmetrico.
- Attraversamento **anticipato**:
  1. Si elabora il valore del nodo (ad es., si stampa).
  2. Si attraversa il sottoalbero sinistro in ordine anticipato.
  3. Si attraversa il sottoalbero destro in ordine anticipato.
- Attraversamento **differito**:
  1. Si attraversa il sottoalbero sinistro in ordine differito.
  2. Si attraversa il sottoalbero destro in ordine differito.
  3. Si elabora il valore del nodo (ad es., si stampa).

Algoritmo di visita in profondità

```
void VisitaDFSSimmetrica(albero t)
{
   if (!Vuoto(t))
   {
      VisitaOrdineCentrale(FiglioSx(t));
      Accedi(t);
      VisitaOrdineCentrale(FiglioDx(t));
   }
}

void VisitaDFSPostOrdine(albero t)
{
   if (!Vuoto(t))
   {
      VisitaOrdinePosticipato(FiglioSx(t));
      VisitaOrdinePosticipato(FiglioDx(t));
      Accedi(t);
   }
}
```

Esempio



Simmetrico: 7 11 17 25 31 43 44 47 65 68 77 93
Anticipato: 47 25 11 7 17 43 31 44 77 65 68 93
Differito: 7 17 11 31 44 43 25 68 65 93 77 47

L'albero mostrato in figura è di tipo particolare, i suoi nodi sono infatti disposti in modo speciale rispetto al loro valore, quale?

Algoritmo di visita in ampiezza

L'algoritmo di visita in ampiezza (**Breadth first search, BFS**) parte da *r* e procede visitando nodi per livelli successivi. Un nodo sul livello *i* può essere visitato solo se tutti i nodi sul livello *i-1* sono stati visitati.

Idea:

visito prima la radice (nodo a livello 0), poi tutti i figli della radice (nodi a livello 1), poi tutti i figli dei figli della radice (livello 2), e così via, ...

Mi espando per livelli successivi.

Criteri di scelta:

- scelgo il primo (in ordine di tempo) nodo visitato che abbia figli non visitati.
- scelgo tutti i suoi figli per visitarli.

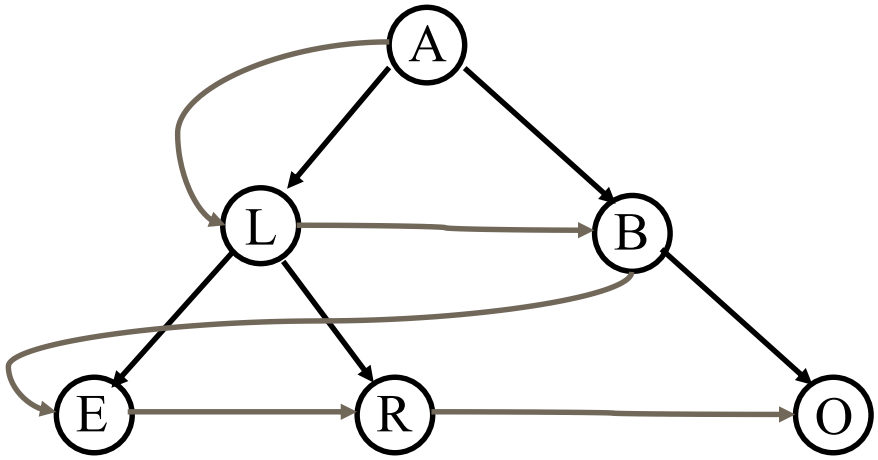
Rappresento l'insieme dei **nodi aperti** (nodi che rappresentano i punti di ramificazione rimasti in sospeso e da cui la visita deve proseguire) S mediante il tipo di dato **Coda**.

Algoritmo di visita in ampiezza

Versione **iterativa** (per alberi binari):

```
void VisitaLarghezza(albero t)
{
   coda c=CreaCoda();
   Push(t,c);
   albero t1;
   while (!CodaVuota(c))
   {
      t1= Front(c);
      Accedi(t1);
      if (!Vuoto(FiglioSx(t1)))
         Push(FiglioSx(t1),c);
      if (!Vuoto(FiglioDx(t1)))
         Push(FiglioDx(t1),c);
      Pop(c);
   }
}
```

Algoritmo di visita in ampiezza



Ordine di visita: A L B E R O

Implementazione funzioni

```
typedef int elemento;
struct nodo {
   elemento valore;
   nodo* sx;
   nodo* dx;
};
typedef struct nodo nodo;
```

```
int Accedi(albero t)
{
   return t->valore;
}

albero FiglioSx(albero t)
{
   return t->sx;
}

albero FiglioDx(albero t)
{
   return t->dx;
}

bool Vuoto(albero t)
{
  return (t==NULL);
}
```

Implementazione funzioni

```
albero CreaFoglia(elemento valore)
{
   nodo* temp=malloc(sizeof(nodo)); //controlli
   temp->valore=valore;
   temp->sx=NULL;
   temp->dx=NULL;
   return temp;
}

void CancellaFoglia(albero &t)
{
   free(t);
}
```

Implementazione funzioni

```
void InserisciSottoAlberoSx(albero pos_alb, albero s_alb)
{
   pos_alb->sx=s_alb;
}

void InserisciSottoAlberoDx(albero pos_alb, albero s_alb)
{
   pos_alb->dx=s_alb;
}

void CancellaSottoAlberoSx(albero &pos_alb)
{
   CancellaAlbero((*pos_alb)->sx);
   free(pos_alb->sx);
}

void CancellaSottoAlberoDx(albero &pos_alb)
{
   CancellaAlbero((*pos_alb)->dx);
   free(pos_alb->dx);
}
```

Implementazione funzioni

```
void Genitore(albero pos_alb, albero t, albero & pos_gen)
{
   if (!Vuoto(t))
   {
      if (FiglioSx(t)==pos_alb || FiglioDx(t)==pos_alb)
         pos_gen=t;
      Genitore(pos_alb,FiglioSx(t),pos_gen);
      Genitore(pos_alb,FiglioDx(t),pos_gen);
   }
}
```

La funzione Genitore è complicata per questa rappresentazione. Ha un tempo di esecuzione lineare nel numero di nodi.

- Il parametro pos_alb rappresenta il puntatore al nodo di cui bisogna trovare il genitore
- il parametro t rappresenta la radice dell'albero
- il parametro pos_gen conterrà il puntatore al genitore trovato.

Non si può richiamare la funzione passando pos_alb uguale alla radice t.

Implementazione funzioni

```
void CancellaAlbero(albero t)
{
   if (!Vuoto(t))
   {
      CancellaAlbero(FiglioSx(t));
      CancellaAlbero(FiglioDx(t));
      CancellaFoglia(t);
   }
}
```

Funzione che dealloca un albero binario. La funzione si basa su una **visita in ordine posticipato**, perché non si può deallocare un nodo genitore prima di aver deallocato i suoi figli.

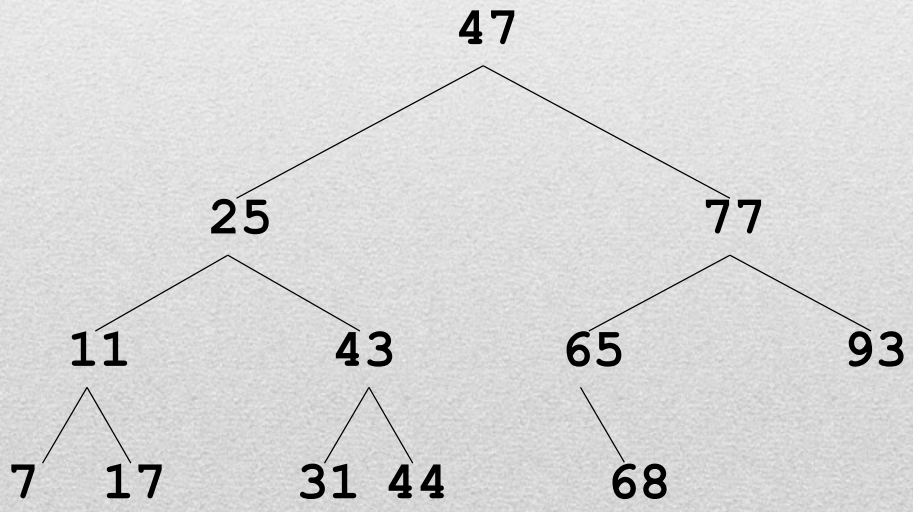
Alberi binari di ricerca

Binary search tree: albero binario che soddisfa le seguenti proprietà

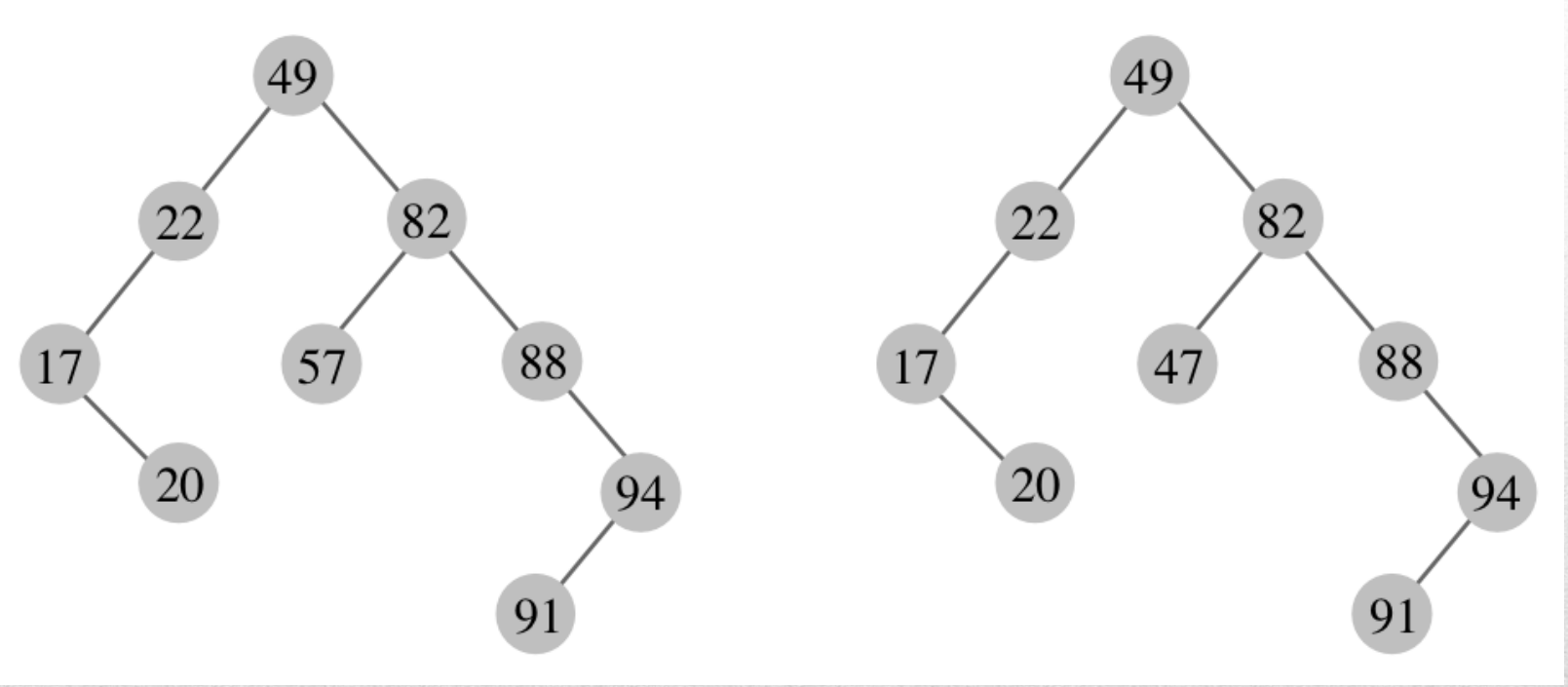
- ogni nodo *v* contiene un elemento *elem(v)* cui è associata una chiave *chiave(v)* presa da un dominio totalmente ordinato.

Per ogni nodo v vale che:

- le chiavi nel sottoalbero sinistro di v sono ≤ chiave(v)
- le chiavi nel sottoalbero destro di v sono > chiave(*v*)



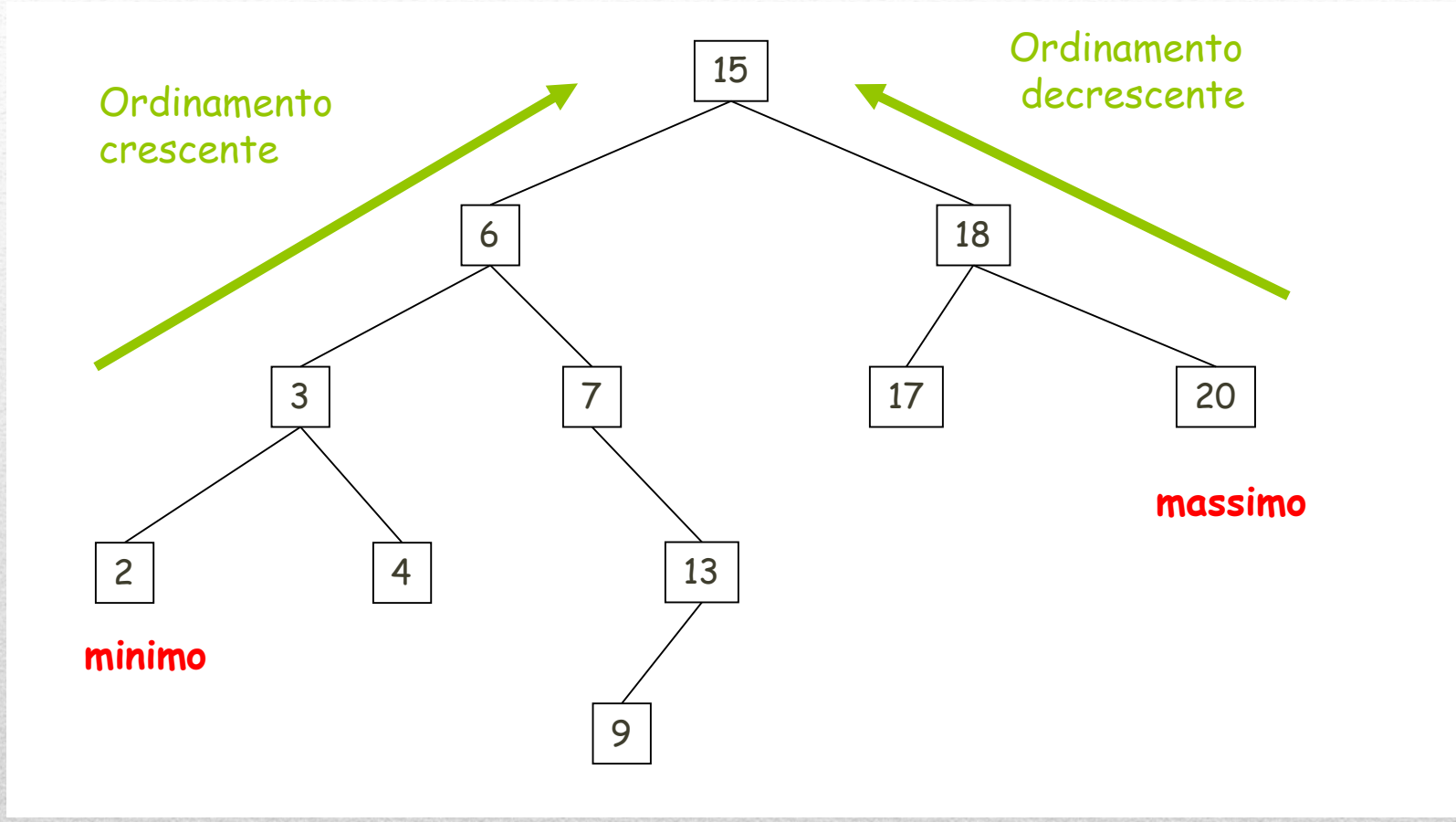
Esempio



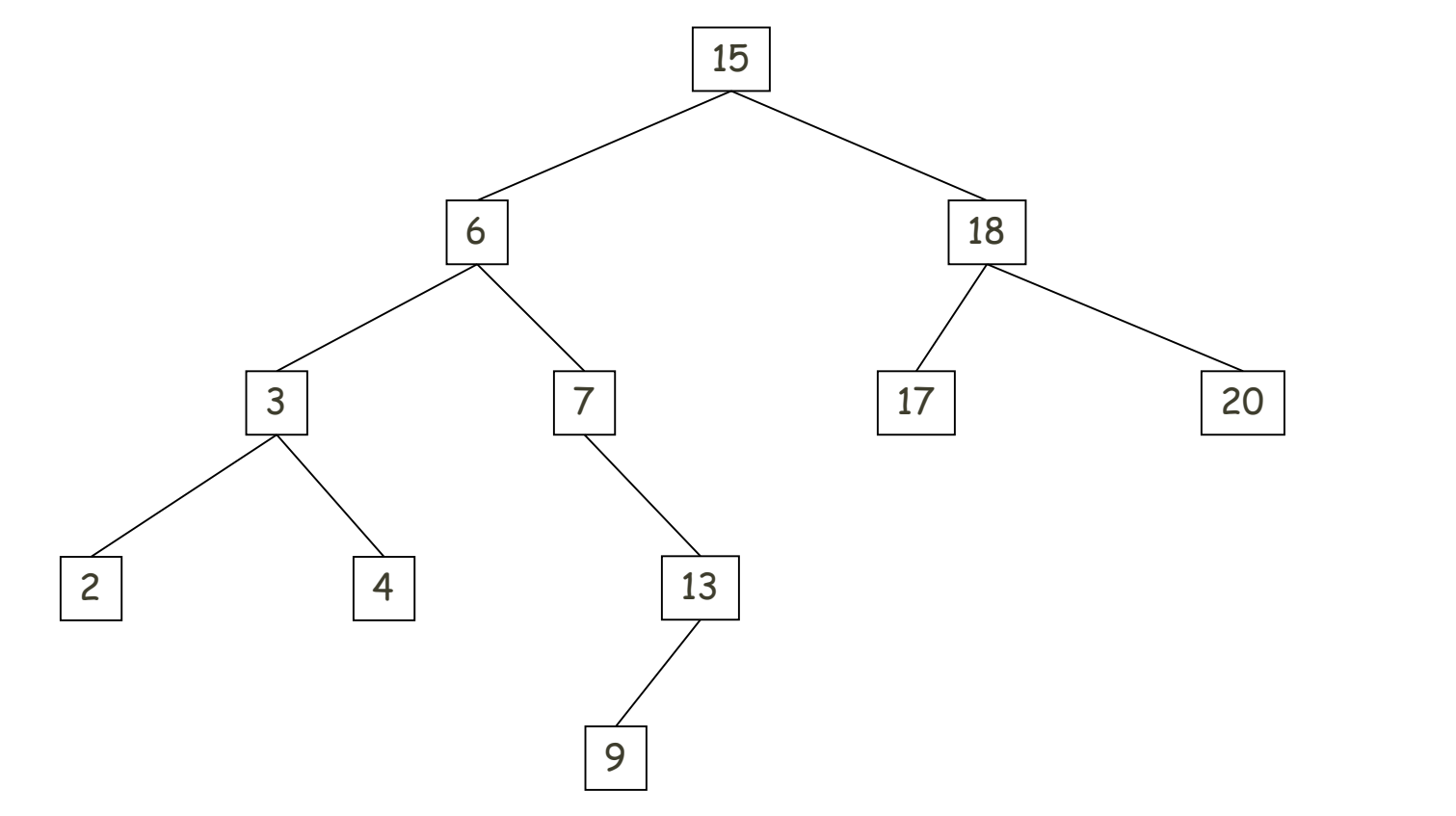
Albero binario di ricerca

Albero binario non di ricerca

Esempio


Ordinamento
crescente
Ordinamento
decrescente
15
6
18
3
7
17
20
2
4
13
9
minimo
massimo

Visita simmetrica di un BST



2 3 4 6 7 9 13 15 17 18 20

visito i nodi in ordine crescente di chiave!