10 - Alberi e Grafi

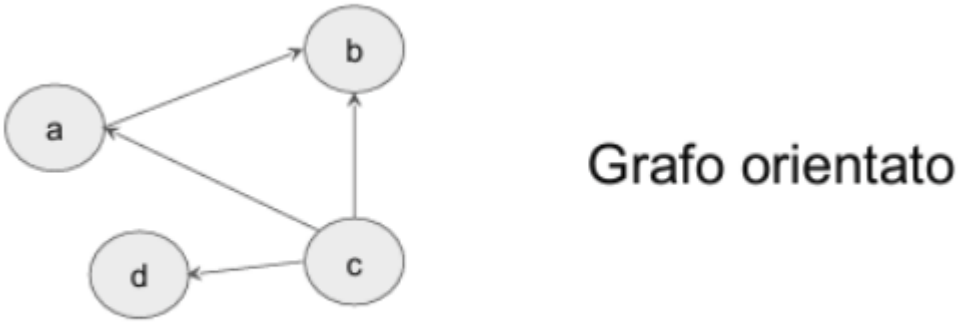
Grafi

Def: Un **grafo** G è definito da una coppia di insiemi $< N, A >$ dove $N = insieme\ dei\ nodi$ e $A = insieme\ degli\ archi$, dove A è definito come sottoinsieme del prodotto cartesiano $N \times N$.

Si dice che se un arco è definito da un nodo $u_i \in N$ ad un nodo $u_j \in N$, allora la coppia $(u_i, u_j) \in A$ ed in questo caso il grafo è **orientato**.

Grafo orientato

In questo tipo di grafo la direzione del collegamento è fondamentale per essere tale.



Visivamente il collegamento si rappresenta con una freccia che parte dal primo nodo e punta al secondo ed è **mono-direzionale**, ovvero che una freccia che va da A in B non implica che valga l'inverso.

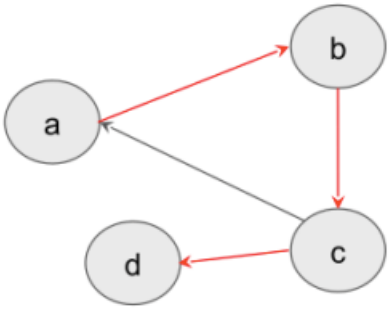
Cammino

In un grafo orientato G un cammino è una sequenza ordinata di nodi $u_0, u_1, \ldots, u_k$ tali che $(u_i, u_{i_+1}) \in A$, per $i = 0, 1, 2, \ldots, k-1$.
Quindi il cammino parte da u_0 e attraversa tutti i nodi, arriva al nodo u_k ed ha lunghezza pari a k stesso.

Non tutti i cammini sono uguali, a seconda di come *saltiamo* tra i nodi, possiamo classificare il cammino in modi differenti:

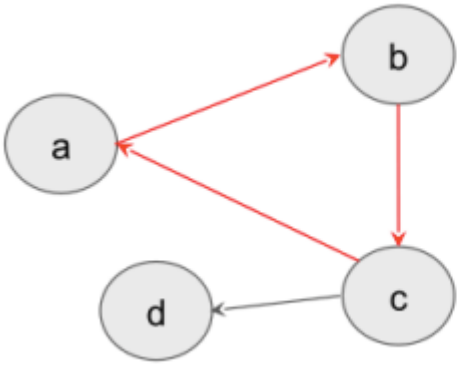
- **cammino semplice**: viene definito tale quando nel percorso non avvengono **nodi ripetuti**. In sintesi, non si torna mai su un nodo già visitato ma si esplorano solo quelli

nuovi e una sola volta e devono essere tutti percorsi.



Cammino semplice: a, b, c, d

- **cammino chiuso**: il cammino viene definito chiuso quando il punto di partenza u_0 coincide con il nodo iniziale u_k, riportandoci all'inizio del percorso.
- **ciclo**: il ciclo è **l'unione tra il cammino semplice e il cammino chiuso**. E' chiuso perché torna al punto di partenza ma è anche semplice perché il ciclo arrivando al punto di partenza come punto di arrivo **ripartirà**, gli altri nodi intermedi sono distinti e non verranno mai ripetuti.

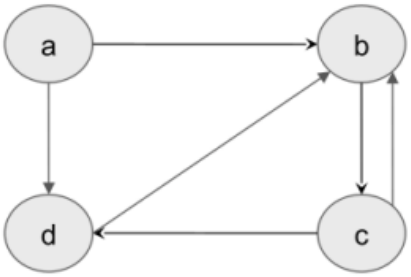


Ciclo: a, b, c, a

Gradi di connessione nei grafi orientati

Quando si analizza un grafo è importante capire non solo le sue frecce ma anche quanto è **lungo**, per questo si studiano tre situazioni importanti in cui si suddivide il suo grado (dalla matematica discreta è la sua lunghezza):

- **Grafo completo**: un grafo si definisce completo se $\forall(u_i, u_j) \in N$ esistere sicuramente un arco che va da u_i a u_j $(A = N \times N)$
- **Grafo connesso**: un grafo si definisce connesso se dati $u \in N \land v \in N$ esiste un cammino da u a v o uno che va da v a u
- **Grafo fortemente connesso**: un grafo si definisce fortemente connesso se $\forall(u, v) \exists\ un\ cammino\ da\ u$ a v ed almeno $un\ cammino\ da\ v$ ad u. Basta che un nodo non soddisfi questa proprietà ed il grafo non è fortemente connesso, inoltre un grafo **completo è sempre fortemente connesso**.

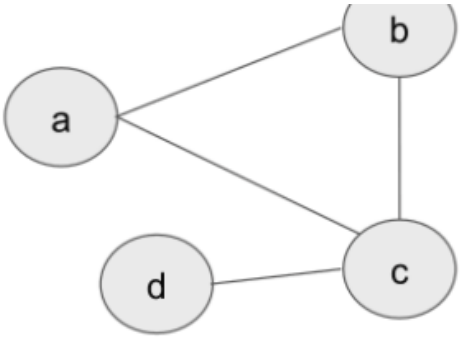


Non è fortemente connesso poiché non esiste un cammino da *c* ad *a*

In pratica, in un grafo fortemente connesso non ci sono strade senza uscita o sensi unici che bloccano il flusso; da qualsiasi punto si sceglie di partire, si può raggiungere qualsiasi altro punto e poi tornare indietro.

Grafo non orientato

In questa tipologia non esiste il concetto di freccia ma quello di linea, avendo un collegamento **simmetrico** (infatti si dice che l'arco **incide** su entrambi i nodi). Se due nodi u_i e u_j sono collegati, la relazione vale in entrambi i sensi.



Grafo non orientato

La loro differenza principale rispetto a quelli orientati sta nella definizione dell'**arco**.
Nel **grafo orientato** la coppia $(A, B) \neq (B, A)$, mentre nel **grafo non orientato** le coppie non sono ordinate quindi scrivere $(A, B) \vee (B, A)$ è la stessa cosa con lo stesso arco.

Due nodi che sono collegati direttamente da un arco si dicono **adiacenti**. Se c'è una linea tra il nodo u e il nodo v, allora u è adiacente a v e viceversa.

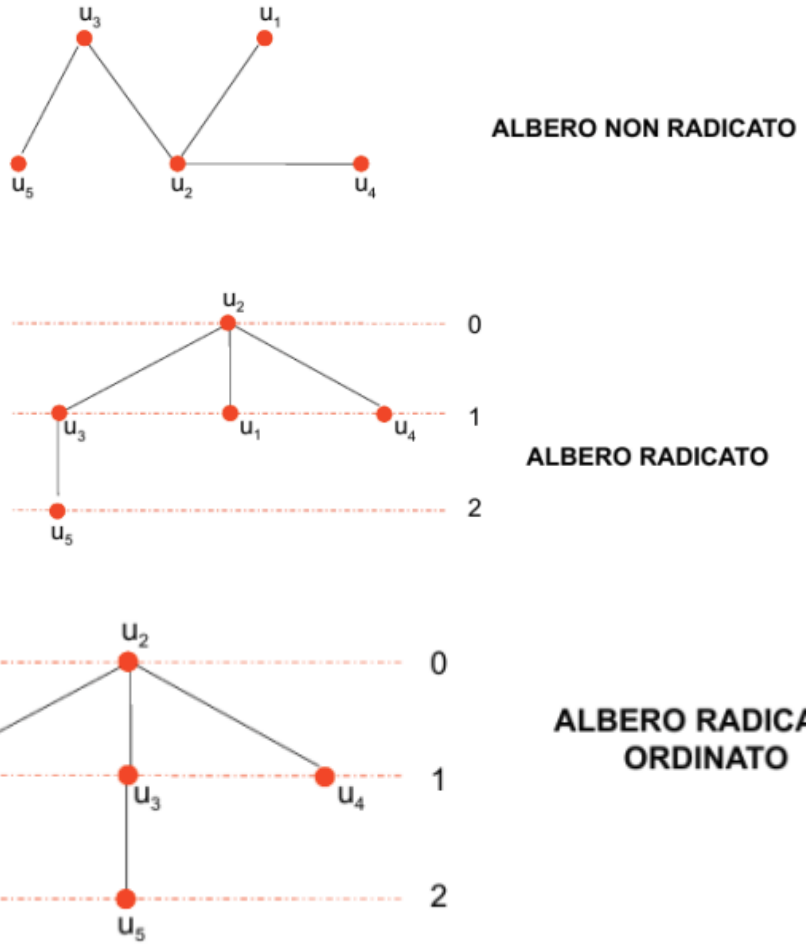
Per muoverci tra i nodi, la logica è la stessa ma con nomi differenti:
quelle che nel grafo orientato prendevano il nome di **cammino** (ovvero la sequenza di nodi da attraversare) qui prende il nome di **catena**, mentre quello che nel grafo orientato era il **ciclo** (il percorso chiuso che ripartiva dall'inizio), qui prende il nome di **circuito**.

Alberi

Un albero è un particolare tipo di grafo; si definisce sempre con la coppia $T < N, A >$, dove N sono i nodi e A sono gli archi.
Tuttavia, riprendendo i concetti dei linguaggi di programmazione e della matematica discreta, un albero per essere tale deve seguire delle regole ben precise:

- **Connesso**: Presi due nodi qualsiasi, esiste sempre una strada che permette di arrivare dall'uno all'altro.

- **Aciclico**: Non devono esiste percorsi chiusi o *anelli*; non bisogna mai tornare ad un punto di partenza mentre ci si muove tra gli archi.
- **Numero di archi**: Il numero degli archi A deve rispettare un determinato numero preciso dettato dalla formula $A = N - 1$.
- **Radice**: La radice definisce la struttura grafica che assumerà l'albero. Si sceglie un nodo r che da quel momento sarà fissato in alto e sarà denominato radice, da esso partiranno a **cascata** tutti gli altri nodi tramite i loro archi. Questo ci permette di ordinare i nodi per **livelli** e creare una gerarchia che verrà usata nelle **strutture dati**.

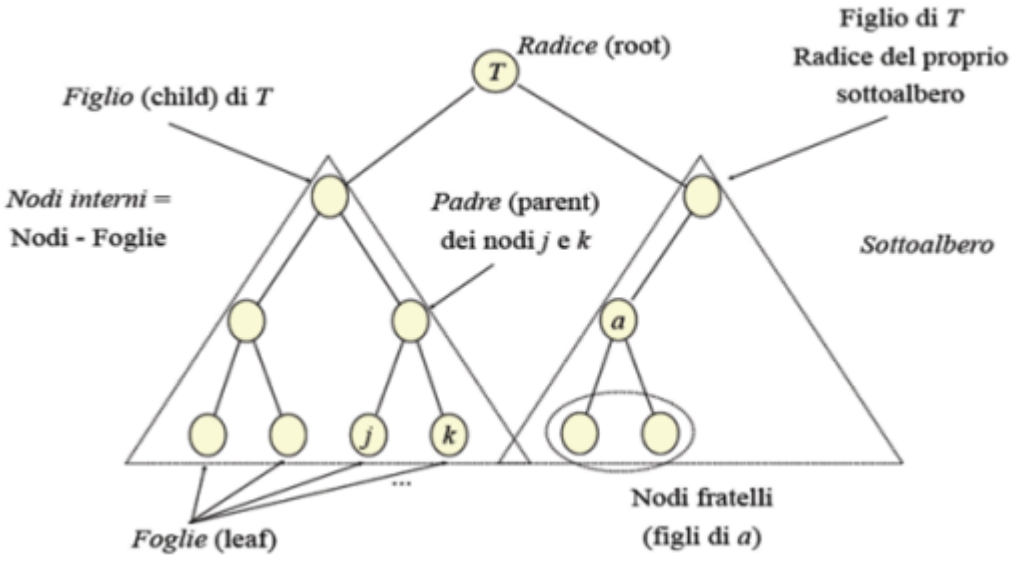


Analizziamo queste strutture e vediamo che i **nodi con lo stesso padre sono fratelli**, come u_3, u_4, u_1. I nodi **terminali senza figli** sono detti **foglie dell'albero**, come u_5.

Proprietà per il suo movimento

1. Ogni nodo ha un solo **padre** ad eccezione della radice che è il capostipite.
2. Essendo l'albero connesso ma aciclico, ne consegue che, se si vuole tornare alla radice esiste **solo un unico cammino** e se ci fossero due modi diversi per arrivare allo stesso nodo, ci sarebbe un ciclo e quindi si va ad **infrangere** la regola che definisce un albero.
3. Un albero è composto da tanti alberi più piccoli l'uno nell'altro. Infatti se si prende un nodo u, che non sia la radice, e ne vedessimo la struttura sottostante vedremo un **sottoalbero**, con *radice* il nodo che abbiamo selezionato.

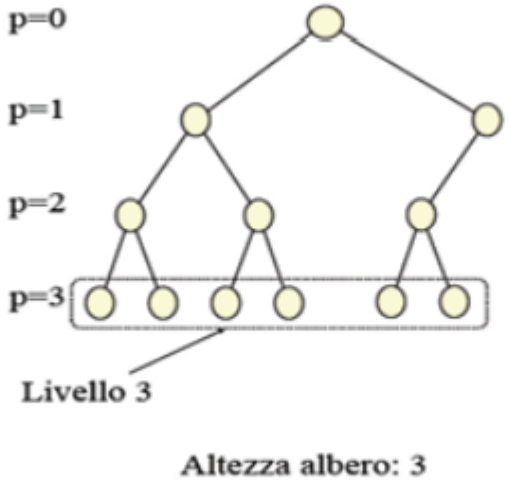
Questo fenomeno prende nome di **struttura ricorsiva**.



Definizioni importanti

- **Albero ricorsivo**: Un albero è un grafo non orientato che o è vuoto oppure esiste un nodo r detto radice, senza predecessori, con $n >= 0$ nodi successori $a_1, a_2, \ldots, a_n$. Tutti gli altri nodi sono ripartiti in n sottoalberi mutuamente disgiunti $T_1, T_2, \ldots, T_n$ aventi rispettivamente $a_1, a_2, \ldots, a_n$ come loro radice.

L'albero è spesso utilizzato per rappresentare relazioni gerarchiche tra oggetti; la definizione data presuppone che sui figli sia definita una relazione d'ordine, infatti:



Altezza albero: 3

- La **profondità di un nodo** è la lunghezza del percorso dalla radice al nodo.
- Il **livello** è l'insieme dei nodi allo stesso livello di profondità p.
- **L'altezza dell'albero** è il massimo livello delle sue foglie.
- **Ordine**: definiamo un albero di ordine k, un albero in cui ogni nodo ha al massimo k figli.

Specifica sintattica e semantica

I tipi di dati che useremo per le operazioni sono:

- **albero**: insieme degli alberi ordinati $T =< N, A >$ in cui ad ogni nodo n in N è associato il livello(n).
- **boolean**: insieme dei valori di verità.
- **nodo**: insieme qualsiasi che non sia infinito.

Gli operatori che useremo sono:

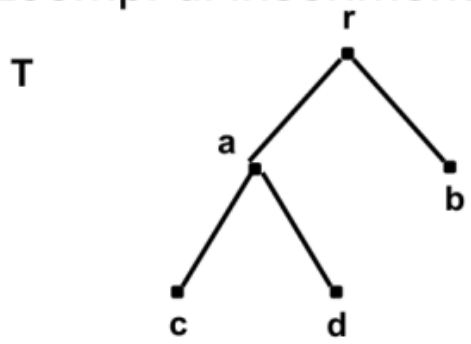
Operatore	Input (Dominio)	Output (Codominio)	Descrizione
`CREAALBERO`	`()`	`albero`	Crea un nuovo albero vuoto.
`INSRADICE`	`(nodo, albero)`	`albero`	Questo comando serve per inserire il primissimo nodo nell'albero.
`ALBEROVUOTO`	`(albero)`	`boolean`	Serve a controllare se c'è qualcosa nell'albero.
`RADICE`	`(albero)`	`nodo`	Restituisce il nodo radice dell'albero.
`FOGLIA`	`(nodo, albero)`	`boolean`	Verifica se il nodo non ha figli.
`ULTIMOFRATELLO`	`(nodo, albero)`	`boolean`	Verifica se il nodo è l'ultimo dei suoi fratelli.
`PADRE`	`(nodo, albero)`	`nodo`	Restituisce il genitore del nodo specificato.
`PRIMOFIGLIO`	`(nodo, albero)`	`nodo`	Restituisce il primo figlio del nodo.
`SUCCFRATELLO`	`(nodo, albero)`	`nodo`	Restituisce il fratello successivo (immediatamente a destra).
`INSPRIMOSOTTOALBERO`	`(nodo, albero, albero)`	`albero`	Inserisce un intero sottoalbero come primo figlio del nodo dato.
`INSSOTTOALBERO`	`(nodo, albero, albero)`	`albero`	Inserisce un sottoalbero come fratello successivo del nodo dato.
`CANCSOTTOALBERO`	`(nodo, albero)`	`albero`	Rimuove il sottoalbero che ha radice nel nodo specificato.

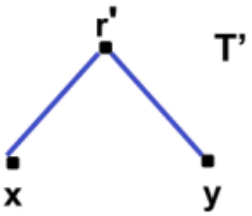
Semantica delle operazioni:

Operatore	Specifica Semantica (Pre e Post Condizioni)
Costruttori	
`CREAALBERO = T'`	**POST:** T' = ($\emptyset$, $\emptyset$) = Λ *(albero vuoto)*
`INSRADICE(u, T) = T'`	**PRE:** T = Λ **POST:** T' = (N, A), dove N={u}, livello(u)=0, A=$\emptyset$
Ispezione	
`ALBEROVUOTO(T) = b`	**POST:** b = VERO se T = Λ; b = FALSO, altrimenti
`RADICE(T) = u`	**PRE:** T $\neq \Lambda \Rightarrow \exists$ v $\in$ N t.c. livello(v) = 0 **POST:** u = v
`FOGLIA(u, T) = b`	**PRE:** T $\neq \Lambda$, u $\in$ N **POST:** b = VERO se $\nexists$ v $\in$ N t.c. <u,v> $\in$ A AND livello(v) = livello(u) + 1
`ULTIMOFRATELLO(u, T) = b`	**PRE:** T $\neq \Lambda$, u $\in$ N **POST:** b = VERO se non esistono altri fratelli di u che lo seguono
Navigazione	
`PADRE(u, T) = v`	**PRE:** T $\neq \Lambda$, u $\in$ N, livello(u) > 0 **POST:** v = x t.c. <x, u> $\in$ A AND livello(x) = livello(u) - 1
`PRIMOFIGLIO(u, T) = v`	**PRE:** T $\neq \Lambda$, u $\in$ N, FOGLIA(u, T) = FALSO **POST:** v è il primo figlio secondo la relazione d'ordine
`SUCCFRATELLO(u, T) = v`	**PRE:** T $\neq \Lambda$, u $\in$ N, ULTIMOFRATELLO(u, T) = FALSO **POST:** v è il fratello di u successivo

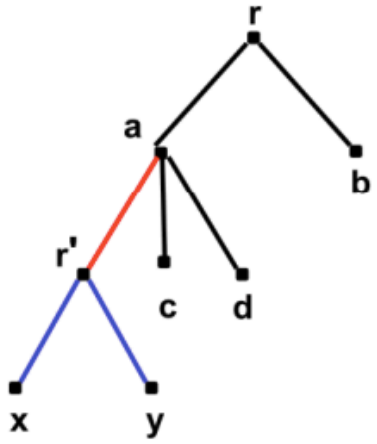
Operatore	Specifica Semantica (Pre e Post Condizioni)
Modifica Strutturale	
`INSPRIMOSOTTOALBERO`	**POST:** T'' è ottenuto da T aggiungendo l'albero T' come primo figlio di u
`INSSOTTOALBERO`	**POST:** T'' è ottenuto aggiungendo T' come fratello successivo di u
`CANCSOTTOALBERO`	**POST:** T' è ottenuto rimuovendo u e i suoi discendenti

Esempi di inserimento

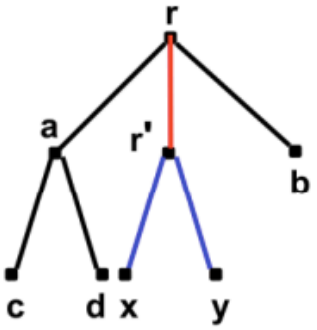




INSPRIMOSOTTOALBERO(a,T,T')



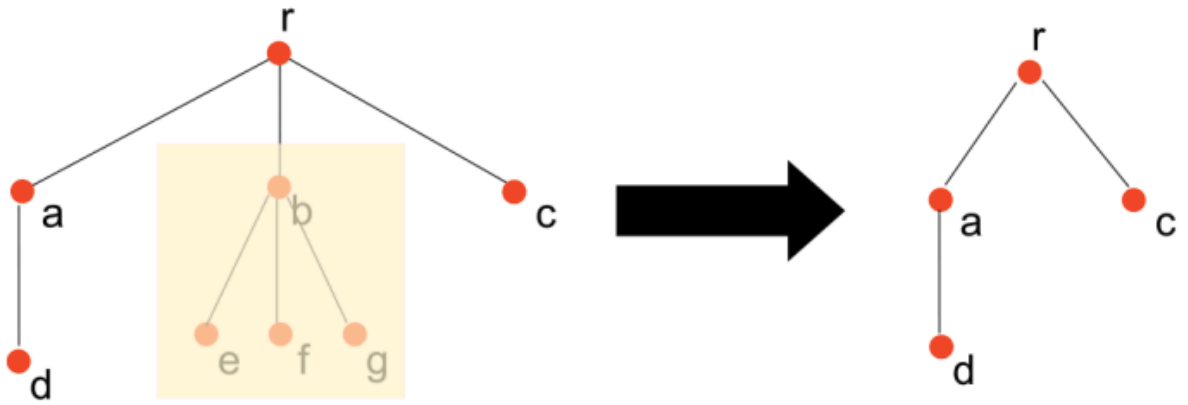
INSSOTTOALBERO(a,T,T')



2

Esempio di cancellazione

CANCSOTTOALBERO(b, T)



Visita di alberi

La **visita di alberi** è un algoritmo che ci permette di pianificare una *rotta* che consente di **esaminare ogni nodo dell'albero** esattamente una **singola volta**.
A seconda delle esigenze possiamo scegliere diverse strategie per tracciare la rotta, in base alla struttura del nostro albero. La differenza tra i metodi di visita, sta proprio nell'ordine in cui decidiamo di seguire la struttura:

- se scendere subito in profondità lungo un ramo
- se vogliamo guardare prima tutti i nodi vicini sullo stesso livello

DFS (Depth First Search)

Questa strategia di visita scende in **profondità** ed è nota anche come visita a **scandaglio**. La strategia è di partire da un nodo e percorrerlo fino in fondo, fino a trovare una foglia, e si sceglie di percorrere un determinato ramo e si prosegue con il medesimo prima di tornare indietro e vedere gli altri. I rami vengono visitati **uno dopo l'altro** ed esistono tre **varianti** di questa visita che dipendono dal momento esatto in cui decidiamo di *leggere* il nodo rispetto ai suoi figli.

BFS (Breadth First Search)

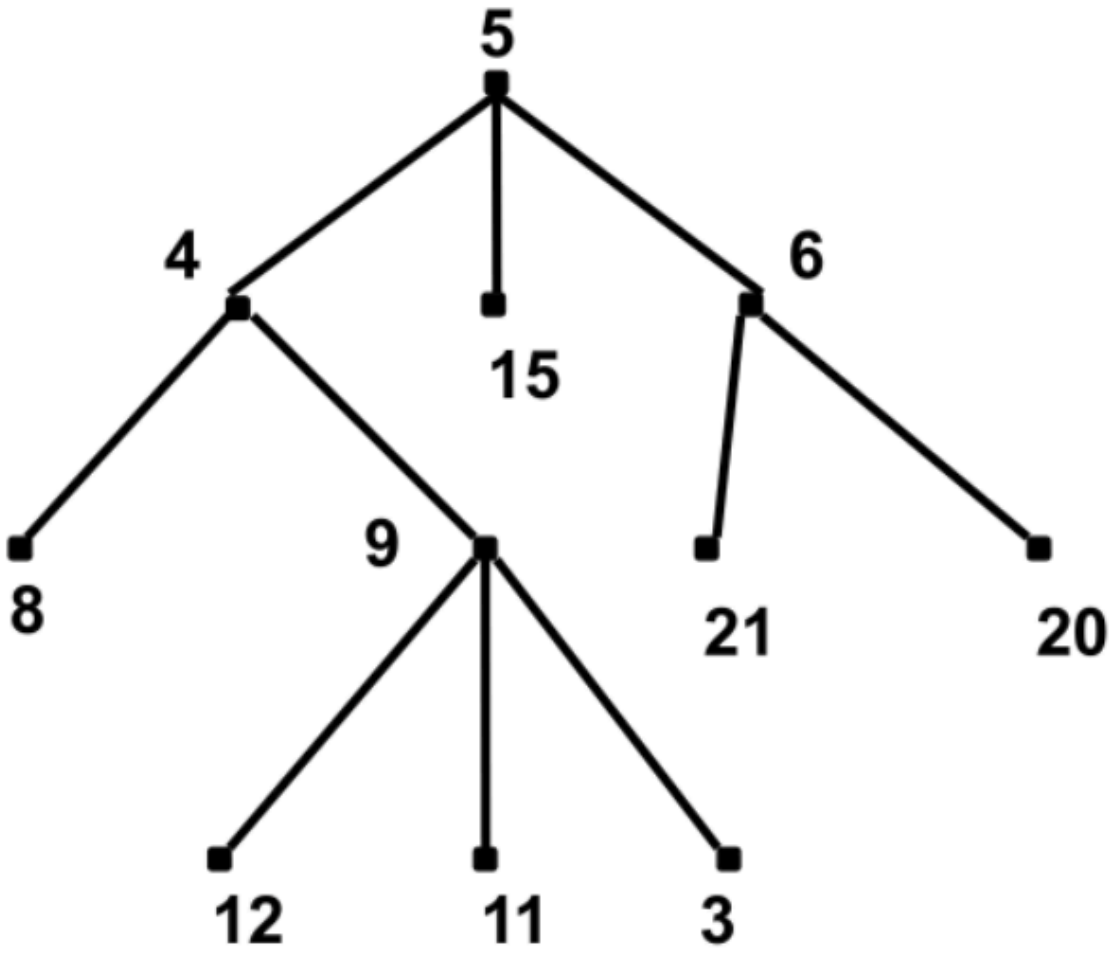
Questa strategia di visita scende in **ampiezza** ed è nota anche come visita a **ventaglio**. L'approccio utilizzato è di tipo **orizzontale**, non si scorre più fino in fondo ma si esplora l'albero per **livelli**:

1. Si parte dalla **root**
2. Si visualizzano tutti i figli della radice di livello 1
3. Dopo averli visti tutti si scende per effettuare la stessa operazione ai nodi del livello 2
4. andare fino alla fine

Gli ordini

A seconda della metodologia scelta, in base al caso più adeguato, i nodi vengono processati in **ordine**, ed esistono quattro tipi di ordinamento:

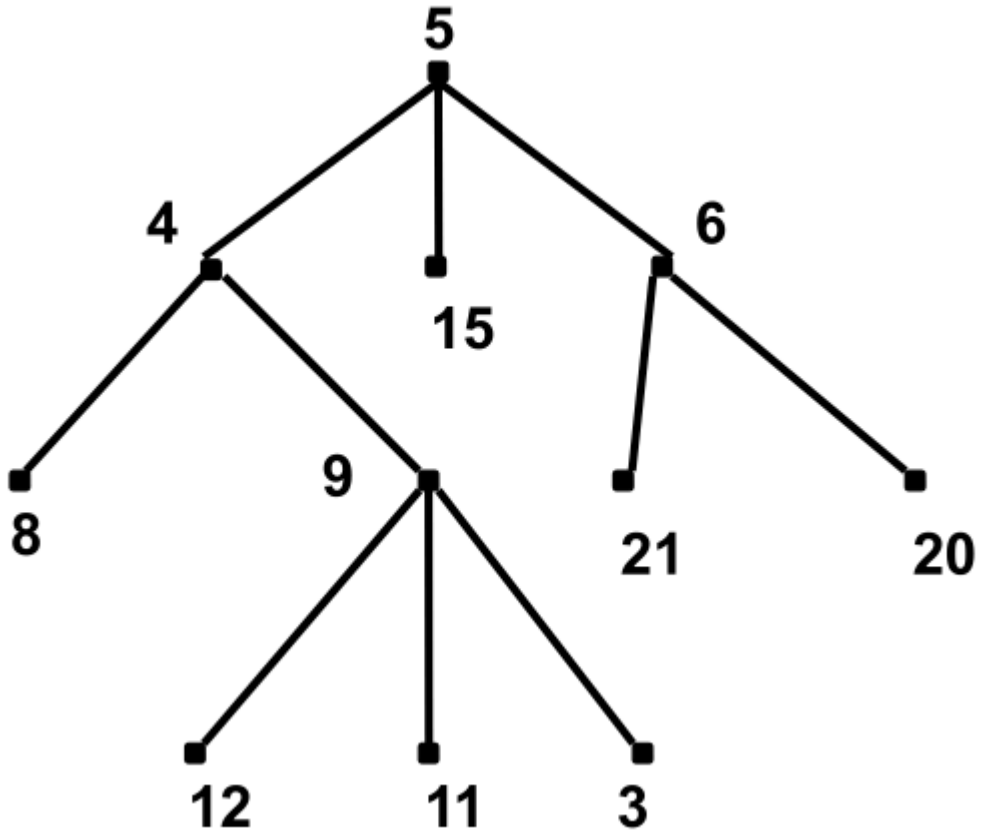
- **Previsita**:
  In questa modalità si parte da r e si visualizzano i sottoalberi T_1, T_2 da sinistra a destra.
  Il prefisso **pre** indica proprio che si visiti prima la radice e poi si vede cosa c'è sotto di lei.



Si parte dalla cima 5, per poi visitare i figli da sinistra a destra, quindi si visualizza 4 poi il suo sottoalbero, partendo sempre da sinistra a destra si visualizza 8 e poi 9, dato che 8 non ha figli e da 9 si visitano le foglie.
Dato che un sottoalbero è terminato, si risale a 4 e si segue la stessa operazione con 15, ed essendo che non ha figli si termina così anche un altro sottoalbero.
Si procede così via via con il figlio 6 e si visualizzano i suoi figli in ordine 21 e 20.
Il risultato finale è:

$$5, 4, 8, 9, 12, 11, 4, 15, 6, 21, 20$$

- **Postvisita**:
  In questa modalità si visitano prima i sottoalberi e poi si sale verso la radice.
  Il prefisso **post** indica proprio che la radice venga visitata dopo aver visto cosa sta sotto di lei.



Si parte dalla cima ma la lasciamo in disparte, prima si analizzano i figli, quindi si andrà al 4, che ha figli e quindi si trascura e andiamo ai figli di esso, quindi 8 che è una foglia e si scrive e 9, esso ha tre foglie e si prendono in ordine da sinistra a destra. Dopo aver registrato i figli di 9 si può registrare 9 stesso e salendo più su anche il 4. Si procede fino a risalire al radice con questo modus operandi avendo come risultato finale:

$$8, 12, 11, 3, 9, 4, 15, 21, 20, 6, 5$$

L'implementazione di previsita e postvisita è il seguente:

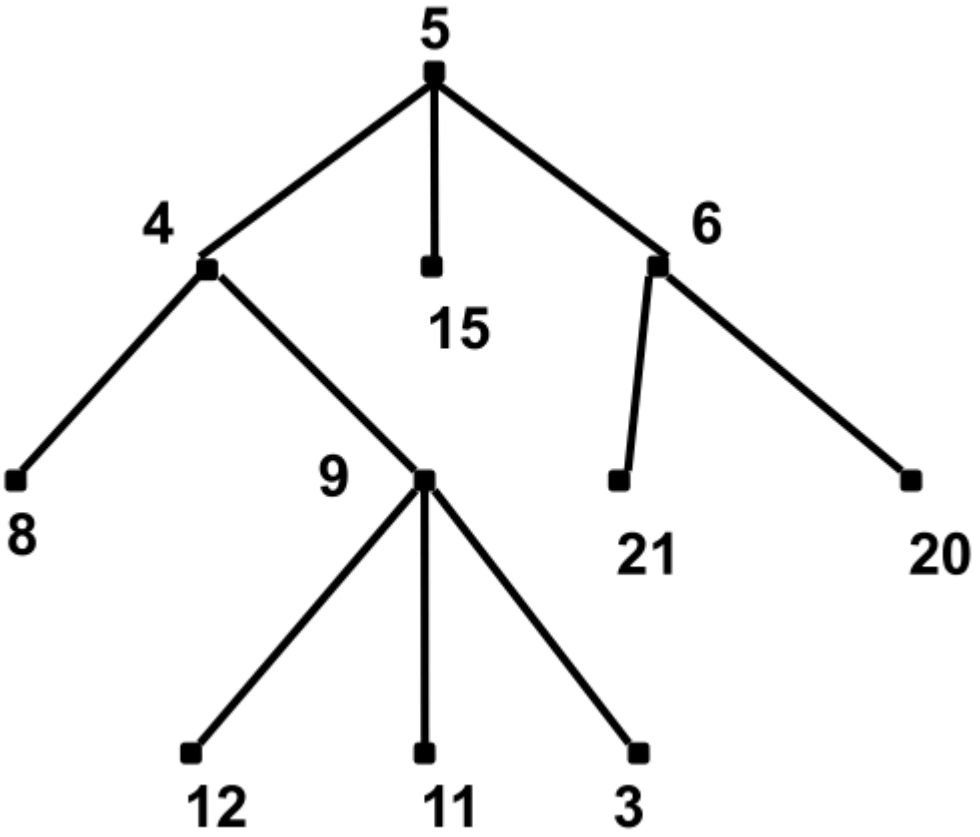
```
PREVISITA(var T:albero; U:nodo){
  nodo C;
  {esamina nodo U}; //1
  if(!FOGLIA(U,T)){ //2
      C=PRIMOFIGLIO(U,T);
      while(!ULTIMOFRATELLO(C,T)){
          PREVISITA(T,C);
          C=SUCCFRATELLO(C,T);
      }
      PREVISITA(T,C);
  }
}
```

La prima azione che esegue l'algoritmo è visitare subito il nodo su cui arriva l'istruzione, dopo di che controlla se il nodo ha figli, se è foglia, l'algoritmo si ferma, altrimenti entra nel blocco dell'if per gestire i discendenti, ripetendo le operazioni fino all'ultimo fratello presente. All'uscita del ciclo, la funzione chiama se stessa per vedere gli altri fratelli.

Scambiando l'ordine dell'istruzione 1 con la 2 si otterrà la Postvisita.
Come si può notare, strutturalmente l'algoritmo di navigazione è identico, cambia solo il momento in cui si decide di stampare il dato.

- **Invisita**:
  Più arzigogolato è la struttura **in**, dove il prefisso stesso definisce l'interno, ovvero la radice viene visualizzata internamente durante la visita ai sottoalberi iniziali e quelli finali. Si parte principalmente dal sottoalbero T_1, successivamente si visualizza r ed in fine, in ordine, i sottoalberi restanti.

  Per applicare questa visita si sceglie un **punto di taglio** i, solitamente impostato ad uno. La struttura diventa la seguente



Si analizzano totalmente prima gli i sottoalberi, dopo di che si studia la **radice corrente** ed in fine, si riparte con gli altri sottoalberi mancanti.
Seguendo l'esempio avremo quindi come risultato finale:

$$8, 4, 12, 9, 11, 3, 5, 15, 21, 6, 20$$

L'implementazione è simile alla precedente, con l'aggiunta della nuova clausola i:

```
INVISITA(var T:albero,U:nodo,i:int){
if(FOGLIA(U,T))
```

```
      {esamina nodo U}
else{
    nodo C=PRIMOFIGLIO(U,T);
    int k=0;
    while(!ULTIMOFRATELLO(C,T)&&k<i){
        k=k+1;
        INVISITA(T,C,i);
        C=SUCCFRATELLO(C,T);
    }
    if(ULTIMOFRATELLO(C,T)&&k<i) //ultimofratello==true si esce
        INVISITA(T,C,i);
    {esamina nodo U};
    while(!ULTIMOFRATELLO(C,T)){
        INVISITA(T,C,i);
        C=SUCCFRATELLO(C,T);
    }
    if(k==i)
        INVISITA(T,C,i);
}
}
```

- **Ampiezza**:
  Analizzando sempre l'esempio precedente, qui non andremo a studiare i rami ma i nodi saranno visitati per livello, partendo dalla root fino all'ultima foglia a destra.

  Il termine **Ampiezza** indica che ci allarghiamo su tutto l'orizzonte del livello corrente prima di scendere più in profondità nel livello successivo. Questa strategia è completamente diversa dalle tre precedenti. Se prima seguivamo i rami e le parentele, qui **ignoriamo** le parentele e guardiamo l'albero come se fosse fatto a strati.
  L'algoritmo non scende n profondità, ma si muove orizzontalmente.

  Al livello 0 c'è la radice, il valore 5.
  Al livello 1 sono presenti i figli della radice, quindi $4, 15, 6$.
  Al livello 2 sono presenti i nipoti, sarebbero i sottoalberi, quindi qui si registrerà sotto il 4, l'$8, 9$; sotto la root di 9 si avranno i **pronipoti** e così via, avendo in fine:

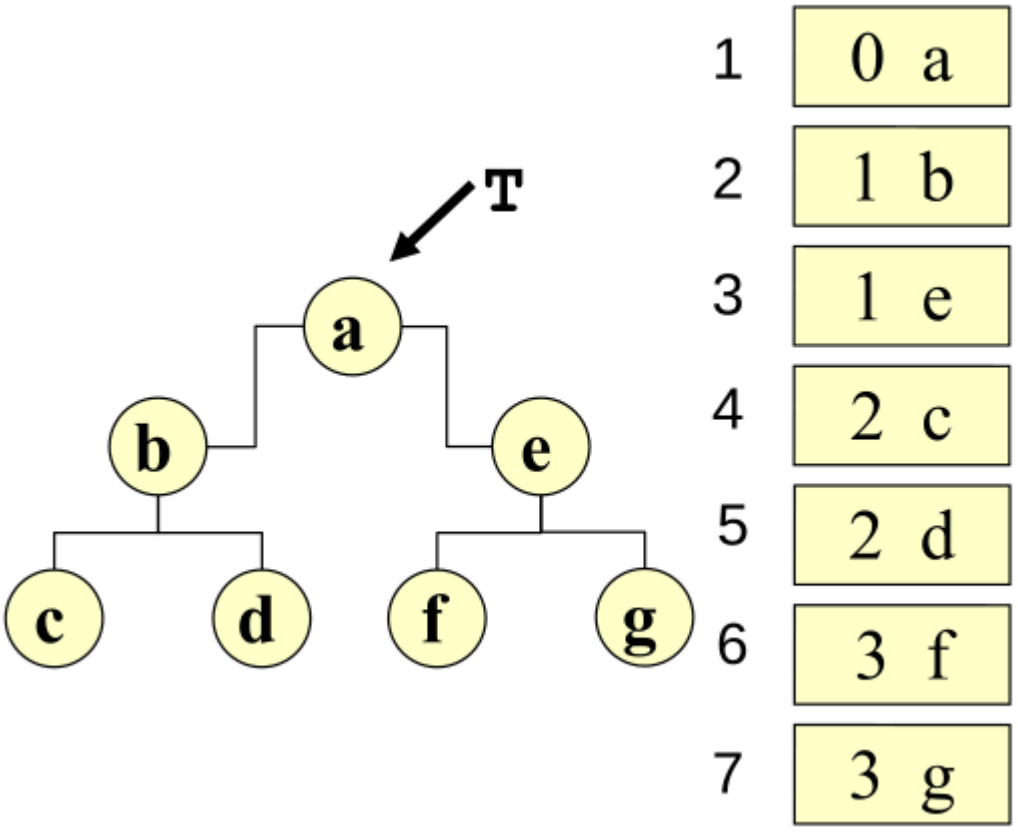
$$5, 4, 15, 6, 8, 9, 21, 20, 12, 11, 3$$

Realizzazione

Realizzazione vettori padri

Questo modello di rappresentazione si basa sull'inserire la struttura dell'albero in un vettore sequenziale, sfruttando le relazioni parentali.
La lunghezza del vettore deve essere pari ad n, dove n è anche il numero dei nodi dell'albero. La logica di memorizzazione dei dati si basa su due regole:

- **Identificazione del nodo**: Ogni posizione i dell'array corrisponde ad uno specifico nodo.
- **Riferimento al padre**: Il valore presente nella posizione i è l'indice al padre.



> Analizzando il disegno e prendendo il nodo c, notiamo come sia posizionato nell'indice 4 e il contenuto della cella 4 è il valore 2 che corrisponde alla cella in cui è presente il nodo padre b.

A livello di C.C. questa struttura presenta un ottimo **vantaggio**, ovvero la possibilità di poter risalire velocemente ai padri, tuttavia presenta anche un grande **svantaggio**, se bisogna cancellare o inserire un intero sottoalbero, la via più facile sarebbe quella di ricostruirlo totalmente da zero.

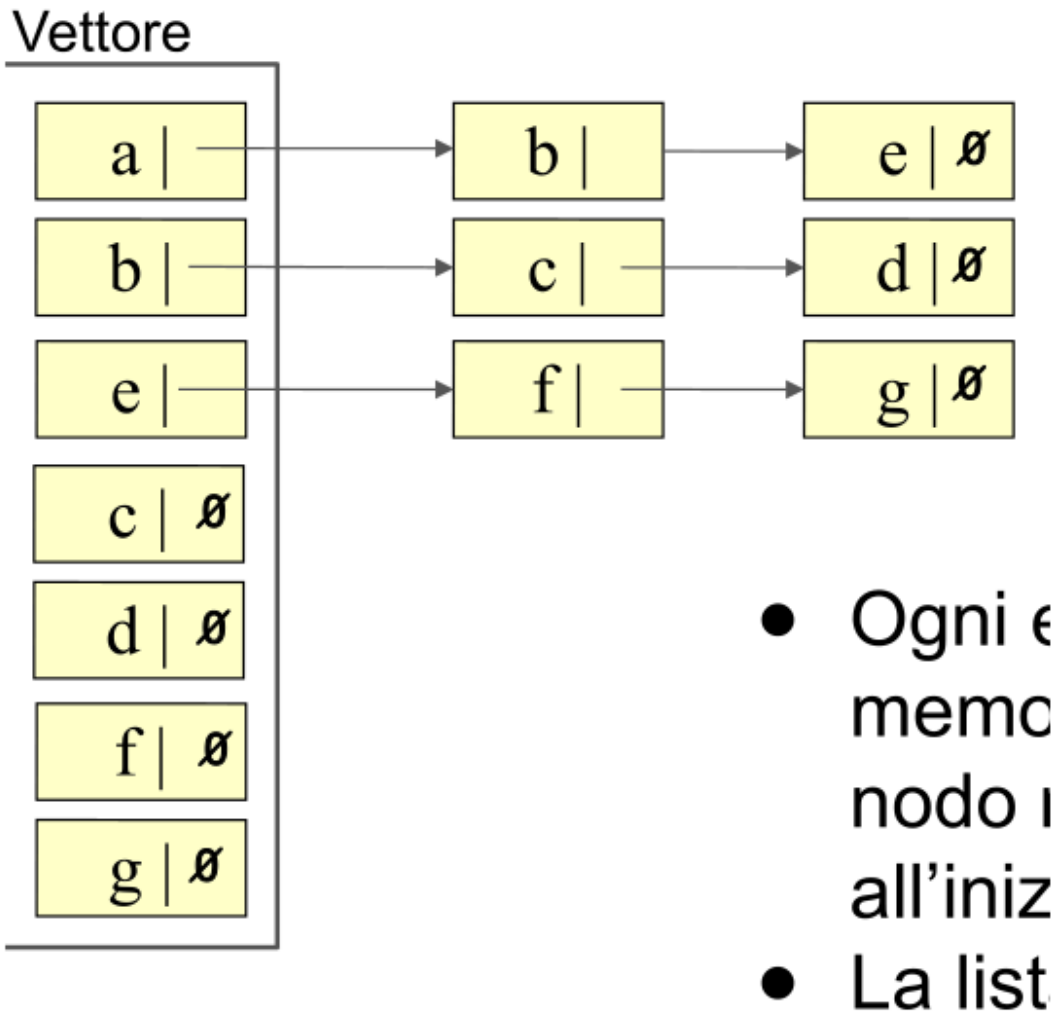
Realizzazione con liste di figli

Questa implementazione non presenta migliorie ma semplicemente privilegia la memorizzazione esplicita dei figli piuttosto che contare i padri, tramite l'uso delle liste.

Ogni cella del vettore rappresenta uno specifico nodo contenente due informazioni:

- **Informazione del nodo**
- **Puntatore**
  Il puntatore presente nel vettore indirizza ad una **lista** che contiene tanti elementi quanti

sono i successori del nodo corrente; se l'elemento corrente non ha figli, quindi è una foglia, il valore del puntatore sarà **nullo**.



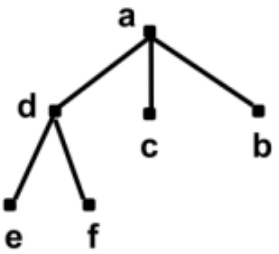
Realizzazione con lista primo figlio/ primo fratello

Questa soluzione è una delle migliori e una delle più eleganti per rappresentare alberi con un numero arbitrario di figli, utilizzando una struttura a dimensione fissa per ogni nodo. Invece di usare una struttura dinamica per i figli, che porta a gran spreco di memoria, ogni nodo mantiene esattamente due **puntatori**:

- **Puntatore al primo figlio**
- **Puntatore al fratello adiacente successivo**
  La sua realizzazione si manifesta tramite l'uso di un array di record, dove ogni riga rappresenta un nodo con tre campi fondamentali:

	FIGLIO	NODO	FRATELLO
1	0	e	2
2	0	f	0
3	0	c	5
4	7	a	0
5	0	b	0
6			
7	1	d	3
8			

4

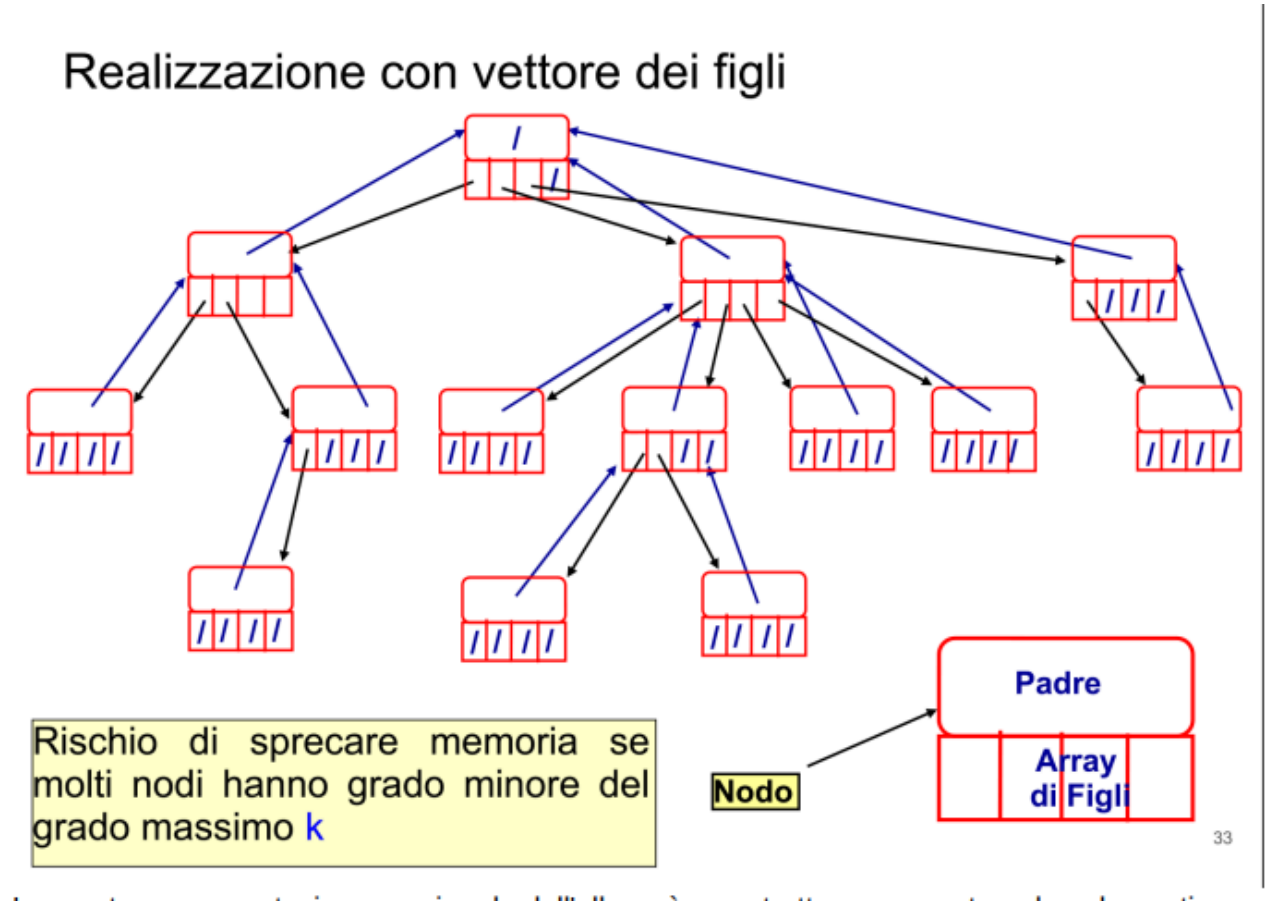


Analizzando l'esempio:

Possiamo comprendere da questo schema che partendo dalla cella 4 in cui c'è il padre si controlla qual è il puntatore al suo **primo figlio diretto** (colonna a sinistra), in questo caso 7, per questo nella cella 7 visualizziamo il nodo di d, con le sue caratteristiche. Tornando ad a (nella colonna di destra) si nota il puntatore al **fratello diretto**, che è nullo poiché non esiste, caso contrario in d in cui esso è presente e punta alla cella 3 in cui è presente c, e si procede in questo modo.

I **vantaggi** di questa struttura sono la possibilità di usare la **stessa quantità di memoria** indipendentemente da quanti figli esso possieda e inoltre si possono rappresentare alberi arbitrario tramite la **logica delle liste concatenate**.

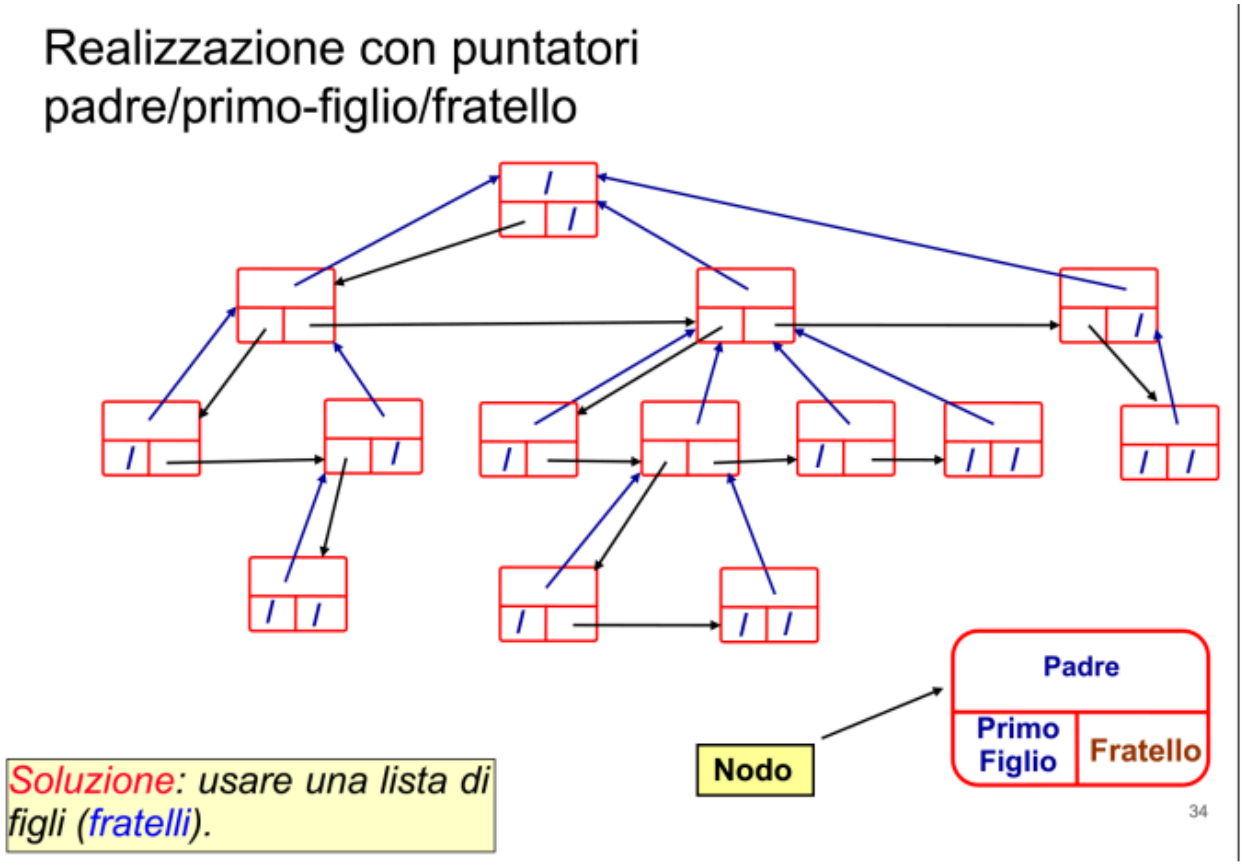
Realizzazione con vettore dei figli



In questa rappresentazione, ogni nodo dell'albero è una **struttura composta a due**

elementi, un riferimento al nodo del genitore (zona blu) e un array contenente i puntatori ai nodi dei figli (zona nera).
La caratteristica principale di questa strategia è che la dimensione dell'array per arrivare ai figli dev'essere **uguale al grado massimo** k; questo porta però ad un enorme **svantaggio**, esiste un alto rischio di **spreco di memoria** qualora la topologia dell'albero sia altamente disomogenea.
Questo avviene se molti nodi hanno un grado altamente inferiore al grado massimo, la maggior parte delle celle del vettore rimarrà vuota e **inutilizzata**. Questo è un problema particolarmente evidente nelle **foglie**, che hanno grado 0 ma **occupano comunque lo spazio per un vettore** di dimensione k.

Realizzazione con puntatori padre - primo figlio - fratello



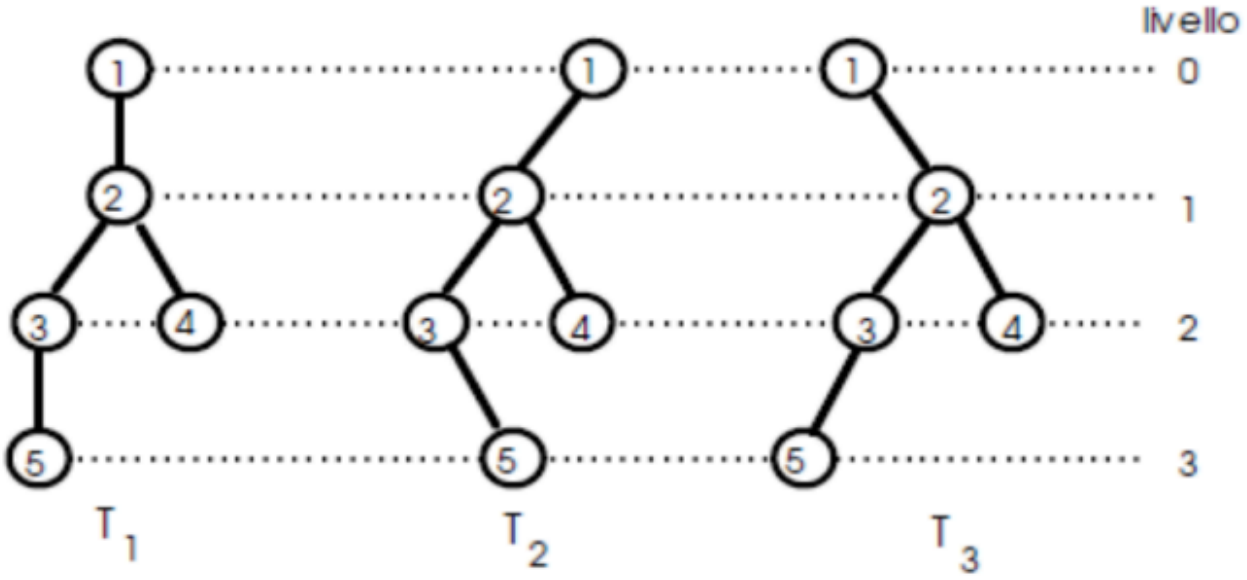
Questa soluzione risolve il problema dello spreco di memoria. Invece di prevedere un numero variabile di puntatori, ogni nodo della struttura possiede esattamente **tre campi fissi**, indipendentemente dalla sua posizione nell'albero:

- Un puntatore al padre
- Un puntatore al **primo** figlio
- Un puntatore che riferisce al fratello **immediatamente successivo**
  Questa implementazione garantisce che ogni nodo occupi una quantità di **memoria costante**, eliminando gli sprechi delle celle vuote tipiche dei vettori a dimensione fissa.

Alberi binari

Un albero binario è un particolare tipo di albero ordinato che deve rispettare due regole per essere tale:

- Ogni nodo può avere **al massimo due figli**, da qui il nome binario.
- I figli vengono esclusivamente indicati come **figlio sinistro** e **figlio destro**. La posizione dei figli infatti definisce un aspetto cruciale che identifica questa nuova tipologia di albero.

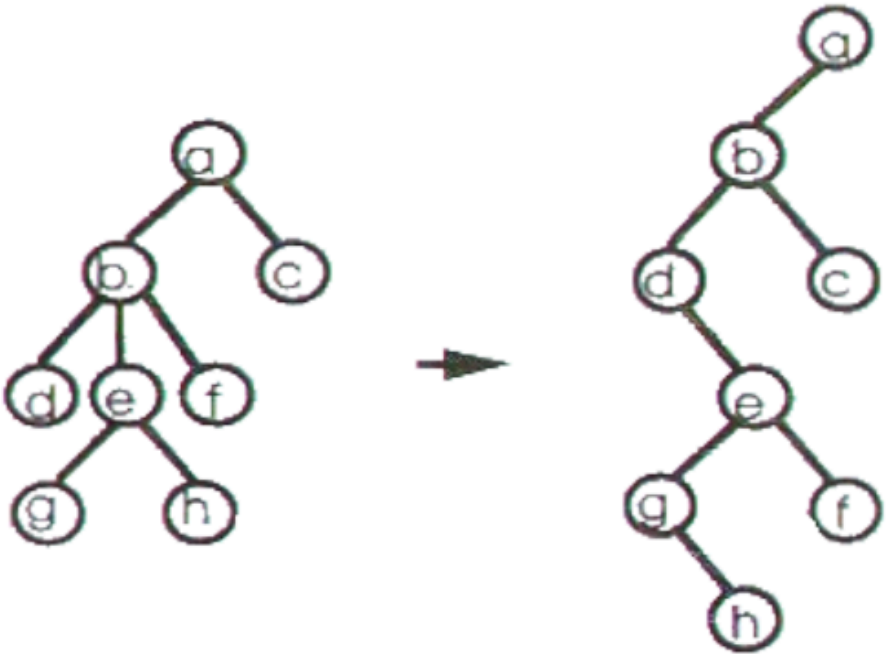


Vedendo questi tre alberi si nota chiaramente che sono strutturalmente simili ma a livello di significato e di informazioni sono totalmente differenti, si vede chiaramente che il nodo 2 in T_2 è a **sinistra**, mentre in T_3 è a **destra**, andando a cambiare **matematicamente** totalmente il significato.

Anche gli alberi binari seguono le caratteristiche della **struttura ricorsiva** tipica degli alberi base per quanto riguarda i sottoalberi.
Nelle implementazioni come `C++ e JAVA` spesso si accede alla foglia finale di un nodo se a sinistra con `a.left()` altrimenti con `a.right()`. Per accedere invece al padre da cui discendono questi due nodi, l'operazione usata è `j.parent()`, con `j` nome del nodo figlio.

Equivalenza tra alberi n-ari e alberi binari

E' sempre possibile trasformare un generico albero n-ario T in un albero binario corrispondente, avente stessi nodi e radice.



La regola di conversione si basa sui puntatori:
Il puntatore **sinistro** diviene una discendenza, quindi un **figlio**, il puntatore **destro** diviene un elemento di orizzontalità, quindi un **fratello**.

Analizzando lo schema seguente possiamo capire con questo esempio il corretto funzionamento:
Albero generico:

Per trasformare questo albero generico in binario dobbiamo ricordarci che, un nodo di un albero binario non può avere più di due figli e che la conversione si basa sulla regola che ciò che sta a sinistra verticalmente diventa figlio sinistro, ciò che sta a destra orizzontalmente diventa figlio destro.

In questo caso partendo dalla radice A il primo figlio che appare a sinistra è B, notiamo come B ha un fratello, esso diventerà suo figlio destro, mentre il figlio E diventa figlio sinistro. E a sua volta ha un fratello che diventerà sempre suo figlio destro.
Tornando sopra a C si nota come esso abbia solo un figlio, quindi G rimane al suo posto sinistro e il fratello successivo di C, ovvero D, diventerà suo figlio destro.

Infine si otterrà questo **albero binario**:

Specifica sintattica e semantica alberi binari

Categoria	Operatore / Concetto	Input / Condizioni	Descrizione e Risultat (Post-condizione)
Definizioni	**ALBEROBIN**	N/A	Insieme matematico degli alberi binari $T = (N, A)$, dove a ogn nodo è associato un livello.
Definizioni	**CREABINALBERO**	Nessuno	Crea l'albero vuoto. **Risultato:** L'albero risultante T' è uguale a (insieme vuoto).
Definizioni	**BINALBEROVUOTO**	Albero T	Controlla l'esistenza dell'albero. **Risultato:** Restituisce *Vero* se $T = \Lambda$, altrimer *Falso*.
Gerarchia	**BINRADICE**	$T \neq \Lambda$	Restituisce il nodo u ch ha livello 0 ($livello(u) =$).
Gerarchia	**BINPADRE**	Nodo u, T ($livello(u) > 0$)	Restituisce il nodo v collegato direttamente a u al livello superiore ($livello(v) = livello(u) -$).
Navigazione	**SINISTROVUOTO**	Nodo u, T	**Ispezione:** Restituisce *Vero* se il nodo u non h un figlio sinistro, *Falso* altrimenti.

Categoria	Operatore / Concetto	Input / Condizioni	Descrizione e Risultat (Post-condizione)
Navigazione	**DESTROVUOTO**	Nodo u, T	**Ispezione:** Restituisce *Vero* se il nodo u non h un figlio destro, *Falso* altrimenti.
Navigazione	**FIGLIOSINISTRO**	Nodo u, T (*SINISTROVUOTO* = Falso)	Restituisce il nodo v ch è specificamente il figlio sinistro di u.
Navigazione	**FIGLIODESTRO**	Nodo u, T (*DESTROVUOTO* = Falso)	Restituisce il nodo v ch è specificamente il figlio destro di u.
Modifica	**COSTRBINALBERO**	T (sinistro), T' (destro)	Unisce due alberi esistenti sotto una nuov radice. **Risultato:** Crea una nuova radice r''. T diventa il sottoalbero sinistro, T' diventa il sottoalbero destro.
Modifica	**CANCSOTTOBINALBERO**	Nodo u, T	Operatore di "potatura". **Risultato:** Rimuove il nodo u e, a cascata, elimina l'intero sottoalbero che ha u come radice.

Per gestire i dati, dobbiamo aggiornare il nostro vocabolario introducendo un nuovo tipo e due nuove funzioni per quanto riguardando gli alberi n-ari:

Categoria	Operatore / Tipo	Sintassi (Input $\rightarrow$ Output)	Specifica Semantica e Descrizione
Nuovi Tipi	**TIPOELEM**	N/A	Rappresenta il tipo di informazione salvata nel nodo (l'etichetta). Può essere un *int*, una *String* o un oggetto complesso.
Dati	**LEGGINODO** *(Lettura)*	(NODO, ALBEROBIN) $\rightarrow$ TIPOELEM	Funzione per recuperare il dato contenuto in un nodo. **PRE:** Il nodo n deve esistere in T. **POST:** Restituisce esattamente il valore a associato al nodo.

Categoria	Operatore / Tipo	Sintassi (Input $\rightarrow$ Output)	Specifica Semantica e Descrizione
Dati	**SCRIVINODO** *(Scrittura)*	(TIPOELEM, NODO, ALBEROBIN) $\rightarrow$ ALBEROBIN	Funzione per aggiornare il dato in un nodo. **PRE:** Il nodo n deve esistere in T. **POST:** Produce un nuovo stato dell'albero T'. La struttura rimane identica, cambia solo il valore nel nodo n.

L'algebra finora studiata non era casuale, ci permetteva di determinare una precisa **scelta progettuale** durante le costruzione degli alberi, ovvero quella di partire dalle foglie per arrivare alla radice collegando tutti i sottoalberi ma non in tutti i casi è la scelta progettuale favorevole. Infatti molto spesso è utile creare gli alberi partendo proprio dalla radice fino alle foglie.
La nuova logica si basa sulla creazione di un **contenitore vuoto** dove verrà posta la radice e poi via via si scenderà verso il basso.

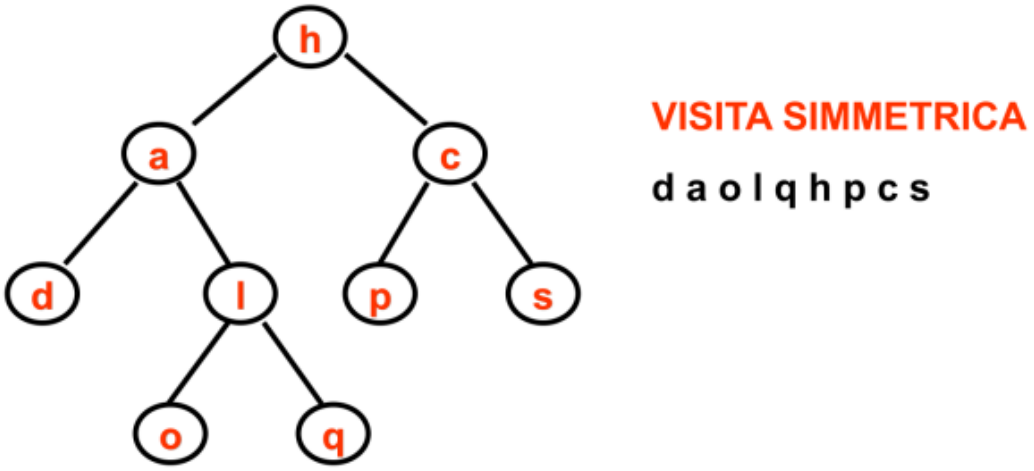
Questo cambio di progettazione non modifica gli operatori esistenti, rimangono validi quelli di **cancellazione, navigazione, creazione, controllo e test**, l'operatore di costruzione `COSTRBINALBERO` sarà l'unico operatore ad essere eliminato e sostituito da **tre nuovi operatori**, progettati per inserire i nodi uno alla volta:

Operatore	Specifica Sintattica (Input $\rightarrow$ Output)	Specifica Semantica e Descrizione
INSBINRADICE	(NODO, ALBEROBIN) $\rightarrow$ ALBEROBIN	**Descrizione:** Inserisce il nodo u come radice. **PRE:** L'albero T deve essere necessariamente vuoto ($T = \Lambda$). **POST:** Il nuovo albero T' è composto da un solo nodo (u) con livello 0.
INSFIGLIOSINISTRO	(NODO, ALBEROBIN) $\rightarrow$ ALBEROBIN	**Descrizione:** Aggiunge un nodo come figlio sinistro di u. **PRE:** L'albero T non è vuoto, u esiste e NON ha già un figlio sinistro. **POST:** L'insieme dei nodi si arricchisce del nuovo nodo collegato a sinistra di u.
INSFIGLIODESTRO	(NODO, ALBEROBIN) $\rightarrow$ ALBEROBIN	**Descrizione:** Aggiunge un nodo come figlio destro di u. **PRE:** L'albero T non è vuoto, u esiste e NON ha già un figlio destro. **POST:** L'insieme dei nodi si arricchisce del nuovo nodo collegato a destra di u.

Invisita per alberi binari

Data la struttura sinistra-destra degli alberi binari, la selezione simmetrica risulta più semplice rispetto a quella degli alberi n-ari dove bisognava decidere *dove* andare a tagliare. Anche qui viene posto per convenienza $i = 1$ e l'algoritmo alla base esegue rigorosamente in sequenza queste tre operazioni in maniera ricorsiva:

1. Si scende ricorsivamente attuando tutta la parte a **sinistra** del sottoalbero.
2. Si **legge** il nodo corrente.
3. Si scende ricorsivamente completando tutta la parte a **destra** del sottoalbero.



Questa foto spiega il procedimento adottato:
Si parte dalla radice e bisogna completare prima tutti i sottoalberi a sinistra, poi si può leggere la radice ed infine ciò che sta a destra, salendo poi per tutti gli alberi più grandi che li racchiudono.
Infatti partendo da h si va verso sinistra dove c'è a ma da a si può ancora andare a sinistra verso d, che essendo foglia verrà letto. Non essendoci altri valori a sinistra di d si può leggere anche a e ora si analizza il sottoalbero alla sua destra. Anche qui l possiede figli a sinistra e sarà letto prima o poi la radice l e infine la sua foglia a destra q. Dopo aver completato tutto il grande sottoalbero a sinistra della radice madre si può leggere la radice madre stessa e analizzare la sua parte destra con lo stesso algoritmo ricorsivo eseguito precedentemente, arrivando al valore finale mostrato in figura.

La sua implementazione (**da imparare a memoria**) sul C++ è la seguente:

```
visitaSimmetrica(ALBEROBIN T, NODO u){
    if(!SINISTROVUOTO(u,t)){
        visitaSimmetrica(T,FIGLIOSINISTRO(u,T));
    }
    {esamina nodo u}
    if(!DESTROVUOTO(u,T)){
        visitaSimmetrica(T,FIGLIODESTRO(u,T));
```

```
    }
}
```

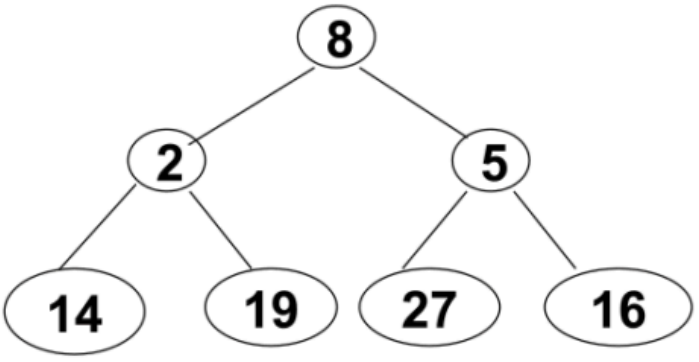
L'algoritmo non procede in modo lineare, ma orchestra il flusso attraverso tre fasi rigorose per ogni nodo visitato. Inizialmente, dà priorità assoluta alla **discesa a sinistra**: se esiste un sottoalbero sinistro, l'algoritmo "*congela*" il nodo corrente e scende ricorsivamente fino a toccare il fondo. Solo quando ritorna da quella profondità si attiva la **fase centrale**, in cui il nodo viene effettivamente esaminato o stampato. Conclusa questa operazione, il flusso si sposta sulla **discesa a destra**: se esiste un sottoalbero destro, l'algoritmo scende nuovamente per esplorarlo, risalendo definitivamente al padre solo dopo aver completato anche questo percorso.
Se il nodo u è una foglia, le due condizioni `if` saranno **false**; l'algoritmo eseguirà solo l'esame del nodo (fase centrale) e terminerà quella specifica istanza ricorsiva, permettendo al processo di risalire al padre.

Rappresentazione sequenziale degli alberi binari tramite vettore

Questa tecnica, definita come **implementazione implicita**, non sfrutta più i puntatori tra i nodi ma le **proprietà matematiche** degli array.
L'idea centrale è associare univocamente la posizione di un nodo ad un indice preciso del vettore senza salvare la struttura dell'albero nei dati ma intrinsecamente negli indici.

Per quanto riguarda le **regole di posizionamento** dei dati, la radice sarà sempre posizionata nell'indice 1, i figli di un determinato nodo in posizione i invece si troveranno obbligatoriamente a:

- **sinistra** quando la loro cella è data da $\text{posizione padre } i \times 2$
- **destra** quando la loro cella è data da $(\text{posizione padre } i \times 2) + 1$



1	8
2	2
3	5
4	14
5	19
6	27
7	16

Possiamo notare questa regola con l'immagine presente, in cui la radice 8 è nell'indice del vettore 1 ed i figli del nodo 3 sono rispettivamente nella posizione di indice 6 per il figlio sinistro, che è dato dal livello 2 (in cui risiede il nodo dal valore 5) $\times$ 3 che è l'indice

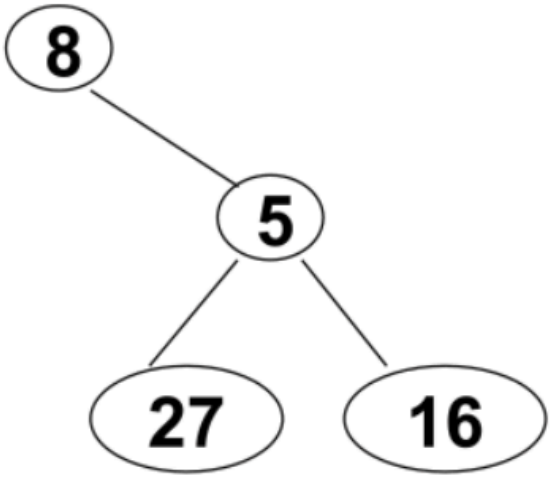
del padre, ed il figlio destro in posizione 7, dato dal livello 2 (in cui risiede il nodo dal valore 5) $\times$ 3 che è l'indice del padre $+1$, come la formula vuole.

Questa metodologia funziona perfettamente con gli alberi privi di **buchi** ed ha una C.C. pari a $O(1)$, l'accesso è garantito a qualsiasi parente dato un indice.

Buchi

I buchi si presentano quando un albero che non è **completo** utilizza la metodologia precedente per gestire i nodi ed è molto rischioso in alcuni linguaggi gestire delle celle non inizializzate.

Per gestire questo problema, si arricchisce la struttura di ogni elemento del vettore, ogni posizione diventa un record composto dall'**informazione** e un **flag**.
Il flag **vero** viene settato al momento in cui si riconosce una posizione con un nodo valido dell'albero mentre sarà settato a **falso** se il nodo è mancante e quindi sarà ignorato.



1	8
2	-
3	5
4	-
5	-
6	27
7	16

Tuttavia questo tipo di implementazione è raramente usata per gli alberi dinamici a causa di tre gravi inefficienze:

1. **Spreco di memoria**: le celle marcate a **falso** occupano comunque spazio fisico nella RAM.
2. **Costo di modifica**: spostare un nodo eseguendo la logica matematica legata all'indice porta al ricalcolo fisico di tutti i suoi discendenti.
3. **Rigidità**: la dimensione dev'essere definita a priori, essendo la struttura basata su un array, perciò si necessità di ricreare l'albero interamente da zero se si sfora il limite definito inizialmente.

Realizzazione con la lista

A differenza della realizzazione con vettori, la realizzazione con la lista si concentra sull'uso totale della memoria, in cui ogni nodo è allocato in una zona qualsiasi della memoria **heap** e i collegamenti sono mantenuti tramite **puntatori**.

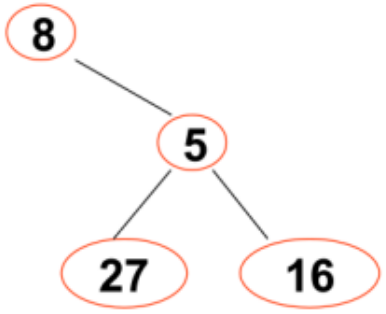
In questa implementazione ogni nodo è un record che contiene, dato e puntatore ai parenti. Ogni nodo u è composto da tre campi:

- **KEY/VALUE**: il dato memorizzato
- **LEFT**: un puntatore che contiene l'id di memoria del figlio sinistro
- **RIGHT**: un puntatore che contiene l'id di memoria del figlio destro
  In alcune implementazioni è presente il campo **PARENT** che facilità la risalita al padre ,ma non è sempre presente.

La radice è l'unico punto di **accesso** all'albero. Il programma mantiene un puntatore esterno che punta al primo nodo. Se un nodo u ha un figlio sinistro, il campo `u.left` conterrà l'indirizzo di quel figlio. Se non ha un figlio sinistro, il campo `u.left` conterrà un valore speciale NULL. Lo stesso vale per il figlio destro. Le foglie, sono nodi che hanno entrambi i puntatori (`left` e `right`) impostati a NULL.

Questa implementazione **risolve** tutti i problemi elencati dall'uso del vettore ma presenta dei nuovi problemi, ogni nodo **occupa più spazio** perché oltre al dato contiene gli indirizzi di memoria e non si potrà più **calcolare matematicamente** la posizione di un nodo, bisognerà partire dalla radice e seguire tutta la mappatura sequenziale data dai puntatori.

Esempio:



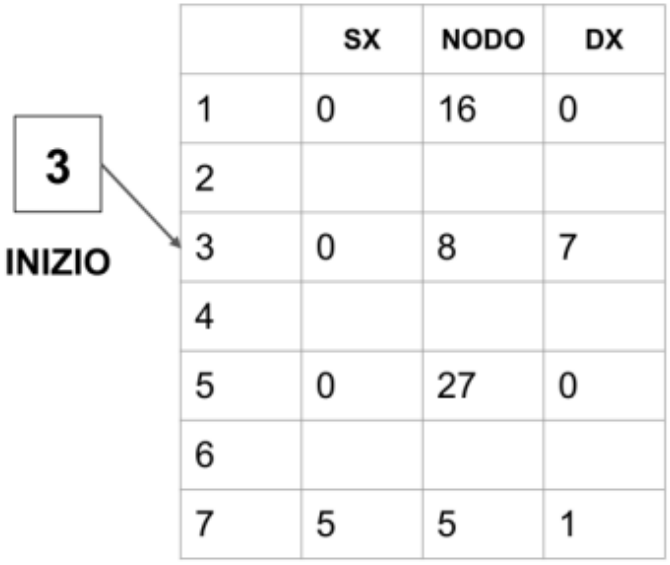
(8 () (5 (27 () ()) (16 () ())))

Per comprendere questo esempio ed il risultato finale, bisogna spiegare la mappatura come viene gestita, se sono presenti le **parentesi vuote** $()$ allora il dato è vuoto altrimenti se è presente $(valore\ ()())$ vuol dire che è presente **un valore foglia** e altri figli vuoti, mentre $(\dots)$ raffigurano una **parentela in corso**.

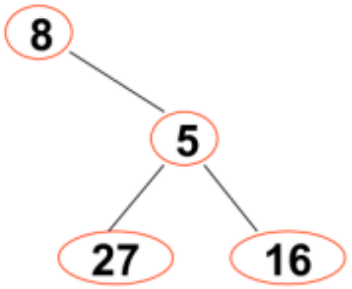
Partendo dalla radice 8 si avrà $(8\dots)$ e si aggiungerà una parentesi vuota poiché a sinistra non c'è nessun figlio, aggiungendo quindi $(8()\dots)$ mentre a destra è presente un figlio che a sua volta è un sottoalbero quindi si aprirà una nuova parentesi $(8()(5(\dots)))$.
Il 5 ha sia figlio sinistro che destro che sono entrambi **foglie** quindi si rappresenterà il 27 tramite $valore\ (\text{sinistro mancante})(\text{destro mancante})$, ovvero $(8()(5(27()())\dots))$, stessa cosa per il 16, avendo come risultato finale:

$(8()(5(27()())(16()())))$

In questo modo l'albero potrà essere rappresentato **dinamicamente** in memoria heap tramite l'uso di puntatori alle celle.



	SX	NODO	DX
1	0	16	0
2			
3	0	8	7
4			
5	0	27	0
6			
7	5	5	1



Alberi di ricerca binaria

Un dizionario è un insieme dinamico di dati che deve soddisfare tre operazioni fondamentali:

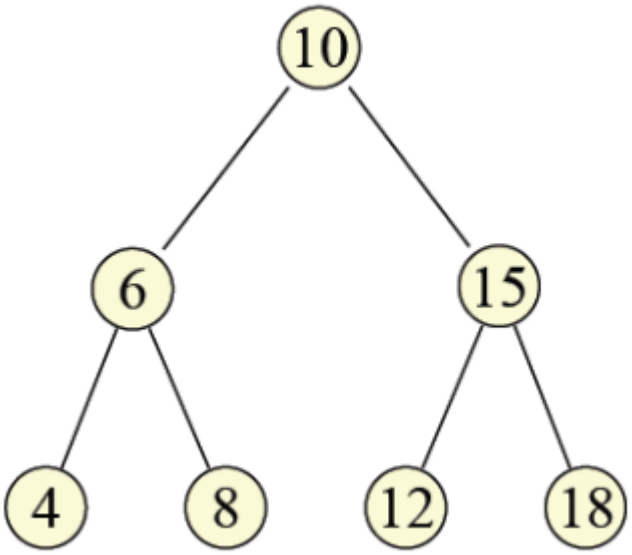
- il **lookup**, ossia trovare l'**item** corrispondente ad una chiave
- l'**insert**, aggiungere una nuova coppia chiave-valore all'insieme
- il **remove**, ovvero rimuovere un elemento associato ad una chiave
  Per eseguire in modo efficiente tutte e tre le operazioni gli array e le liste **non sono sufficienti**, servono gli **alberi**, i quali riescono ad unire il vantaggio della **ricerca veloce** con C.C. pari a $O(\log n)$ degli array ordinati, con l'inserimento istantaneo con C.C. pari a $O(1)$ delle liste non ordinate.

Albero binario di ricerca (ABR)

In un ABR ogni nodo u possiede un informazione composta dalla chiave e dal valore; per le chiavi, un requisito fondamentale è che esse siano poste in un insieme ordinato, ovvero tra u_1 e u_2 posso facilmente comprendere qual è maggiore dell'altra o viceversa.

Affinché un albero binario sia definito anche di **ricerca**, ogni suo nodo deve rispettare le seguenti proprietà:

- Tutte le chiavi contenute nel sottoalbero sinistro devono essere **strettamente minori di** `u.key`.
- Tutte le chiavi contenute nel sottoalbero destro devono essere **strettamente maggiori di** `u.key`.



vedendo questa foto è chiaro comprendere che tutti i nodi nel sottoalbero sinistro della radice 10 sono valori strettamente inferiori della radice stessa, diversamente ciò che sta nel sottoalbero destro possiede valori strettamente maggiore della radice stessa.

Rende subito l'idea dell'immediata **velocità** che si ottiene nella **ricerca**, poiché nel dover cercare un valore basta confrontarlo esclusivamente con la radice, in modo tale da andare a scartare un intero lato dell'albero favorendone solo uno ed anche l'**ordinamento** per ordine crescente non necessita di algoritmi esterni ma basta semplicemente una **visita simmetrica** che vada da sinistra, alla radice ed in fine a destra.

Per supportare questo tipo di operazioni, specialmente la navigazione in todo, il nodo deve essere strutturato tassativamente da **cinque campi**, a loro volta suddivisi in **tre gruppi**, rendendo il nodo di fatto complesso.

- **Navigazione Ascendente**: `Parent` è essenziale per risalire al padre e percorrere indietro l'albero.
- **Dati**:
  1. `key`
  2. `value`
- **Navigazione Discendente**:
  1. `left` (figlio sinistro)

2. `right` (figlio destro)

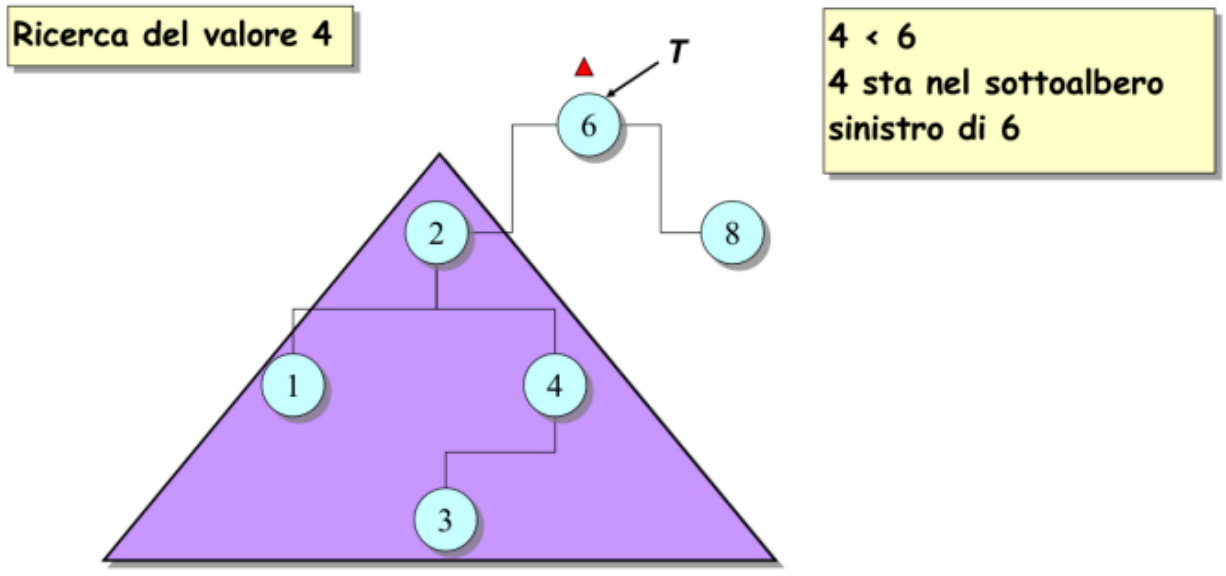
Specifiche dell'ABR

Le operazioni dell'ABR possono essere suddivise in tre categorie logiche:

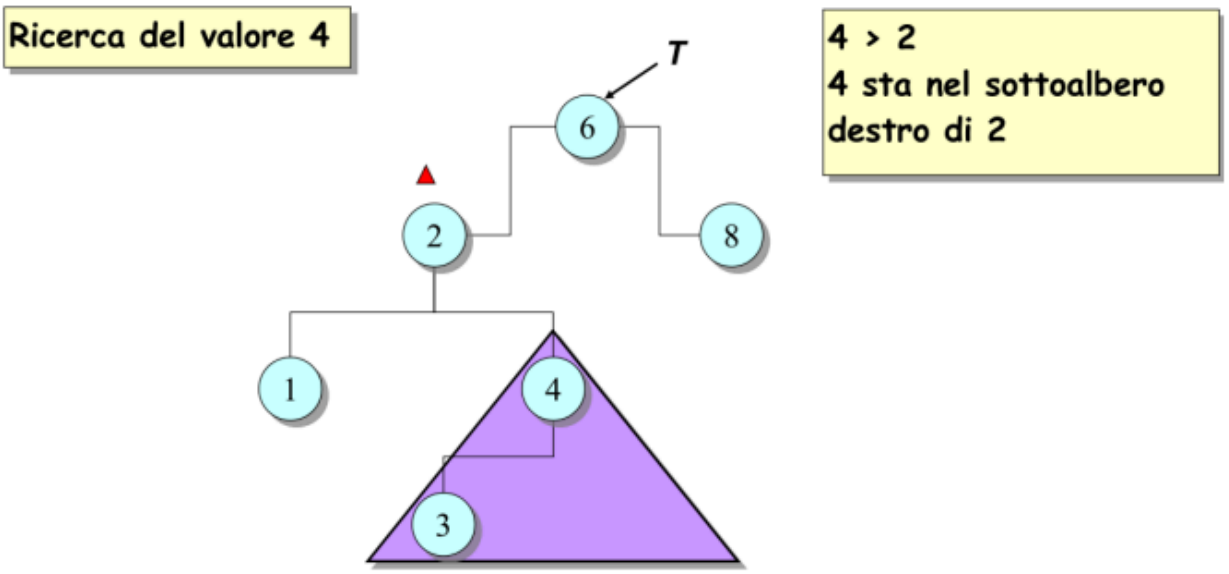
- **Operazioni di accesso e navigazione**:
  Esse permettono di leggere lo stato del nodo corrente, tramite le funzioni `key()` e `value()`, le quali restituiscono rispettivamente la chiave ed il valore di quel nodo o di spostarsi nei nodi adiacenti, tramite le funzioni `left/right()` e `parent()` ove si ottiene il puntatore al rispettivo figlio sx/dx e/o padre.
- **Operazione core**:
  Esse sono le funzioni principali per la gestione dei dati, tramite `lookupNode(Item k)` si cerca un nodo avente la chiave `k`, tramite `insert(Item k,Item d)` si inserisce, continuando con l'ordinamento presente, una nuova coppia di chiave-valore ed infine grazie a `delete()`, rimuoviamo un nodo dall'albero.
- **Operazioni d'ordine e topologiche**:
  Attraverso la struttura ordinata dell'ABR, queste funzioni cercando un determinato nodo, specialmente con la funzione `min()/max()` si restituisce il nodo con la chiave minima (che a logica sarà posto a sinistra) o viceversa per il massimo. Con la funzione `successorNode(Item T)` e `predecessorNode(Item T)` invece si restituisce la chiave del nodo immediatamente successivo/precedente a quello dell'ordinamento totale.

Esempio Ricerca

Per comprendere la **ricerca**, si analizzi la foto qui presente:

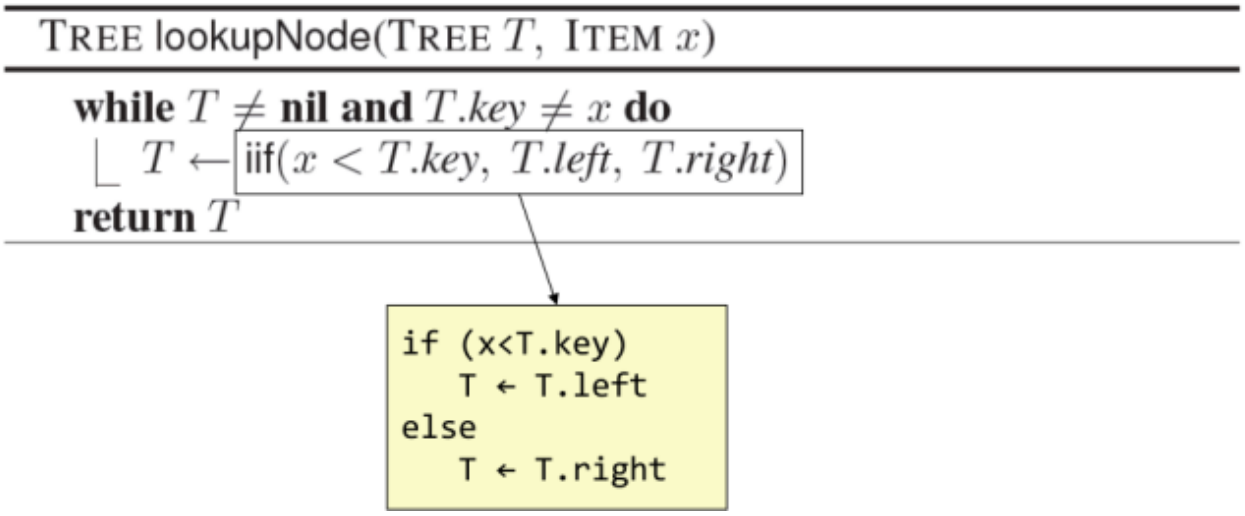


L'algoritmo per trovare il valore 4 inizierà sempre dal puntatore alla radice dell'albero.
Il nodo sotto esame T è la radice, avente valore 6, dopo di che si confronta questo valore con quello da trovare e ovviamente $4 < 6$, quindi si esclude a priori il sottoalbero destro.



Scendendo a sinistra il puntatore T ora porrà sotto esame il valore 2, ed esegue la stessa logica precedente, ossia, dato che $2 < 4$ allora il valore da trovare sarà nel sottoalbero destro.
Così l'algoritmo andrà verso destra e il puntatore T dovrà confrontarsi col il valore 4, dato che quest'ultimo è uguale al valore da cercare, l'algoritmo si ferma con successo restituendo il valore presente nel nodo.
In soli 3 passaggi (**confronti**), l'algoritmo ha trovato il valore in un albero che contiene molti più nodi, dimostrando l'efficienza rispetto a una ricerca sequenziale che avrebbe dovuto controllarli tutti.

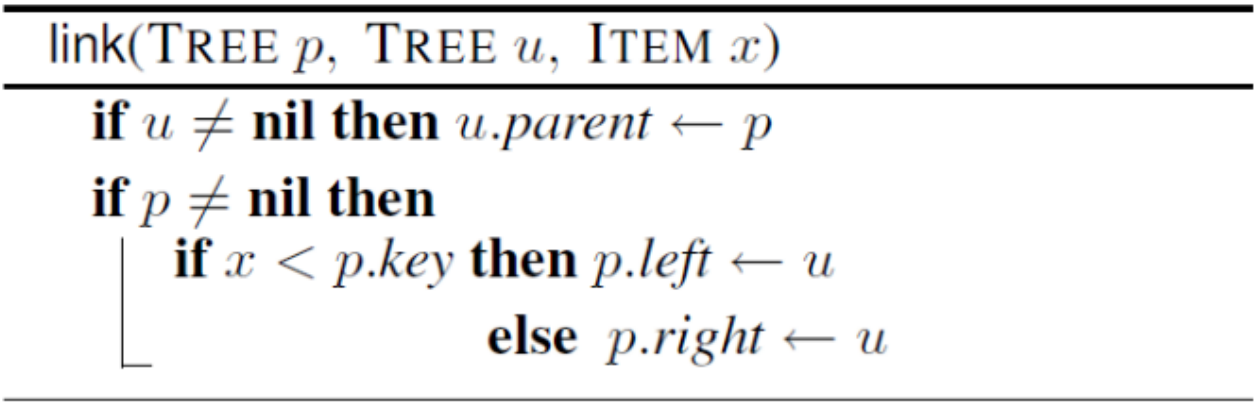
Implementazione:



L'implementazione della funzione di ricerca (`lookupNode`) utilizza un approccio **iterativo**.
L'algoritmo ha lo scopo di individuare un nodo contenente la chiave x all'interno dell'albero T . Se il nodo esiste, viene restituito il puntatore ad esso, terminando il ciclo; in caso contrario, viene restituito un valore nullo (`nil`).
L'algoritmo utilizza un ciclo while per scendere lungo l'albero; il ciclo continua fintanto che sono vere due condizioni contemporanee:

- $T \neq nil$: non si è ancora usciti dall'albero
- $T.key \neq x$: il nodo attuale non contiene il valore da trovare
  All'interno del ciclo, dobbiamo decidere se spostarci a sinistra o a destra, per comprenderlo, il codice utilizza una notazione compatta: `T ← if(x < T.key, T.left, T.right)` , ovvero poni in T la scelta di andare a destra o a sinistra in base al valore di x rispetto a quello corrente.

Inserimento



La funzione `link` serve a gestire la creazione del doppio collegamento con puntatori tra un padre e un figlio, rispettando la struttura dell'albero.
In questa funzione oltre al nodo corrente u e alla chiave x è presente un nuovo elemento p, esso rappresenta il nodo che è destinato a diventare un nuovo padre ed in questo caso il nodo u sarà quello ad esser connesso al padre.

L'algoritmo esegue due operazioni distinte per stabilire la relazione di parentela bidimensionale:

- `if u != nil then u.parent <- p` : se u esiste il suo puntatore al padre viene aggiornato per puntare a p, stabilendo così al figlio la possibilità di connessione con il padre.
  Il controllo $u \neq nil$ è necessario perché questa funzione potrebbe essere usata anche per staccare un nodo (passando nil come u).
- Verifichiamo che il padre esisti, se p esiste si attua la proprietà degli ABR per decidere se u andrà a sinistra o a destra. Se invece p non esiste vuol dire che u sta diventando radice del sottoalbero senza bisogno di alcun collegamento:

```
if p != nil then
      if x<p.key then p.left<-u
      else p.right<-u
```

Inserimento iterativo

TREE insertNode(TREE T, ITEM x, ITEM v)

```
TREE p ← nil                                          % Padre
TREE u ← T
while u ≠ nil and u.key ≠ x do                        % Cerca posizione inserimento
    p ← u
    u ← iif(x < u.key, u.left, u.right)
if u ≠ nil and u.key = x then
    u.value ← v                                       % Chiave già presente
else
    TREE n ← Tree(x, v)
    link(p, n, x)
    if p = nil then return n                          % Primo nodo ad essere inserito
return T
```

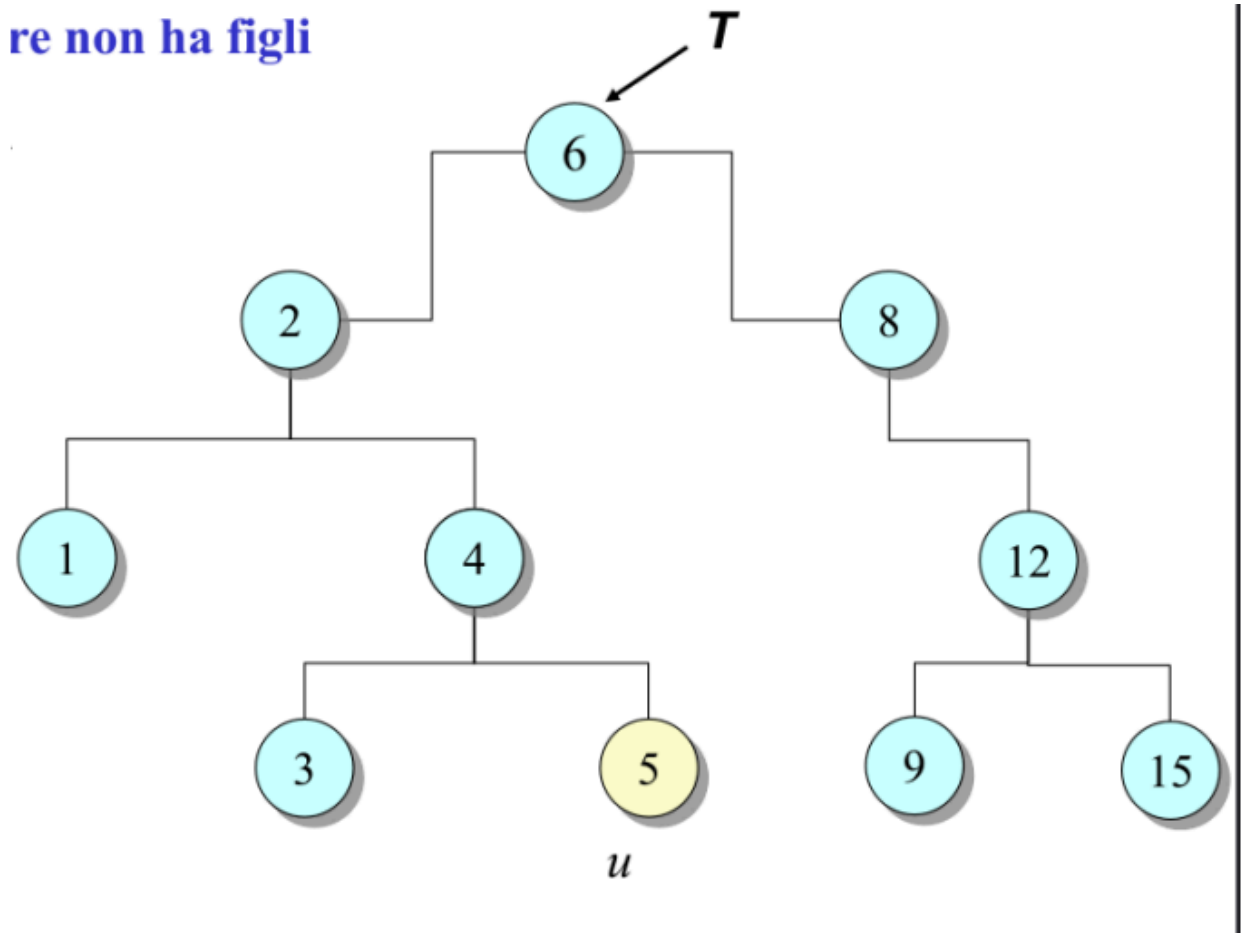
Cancellazione

A differenza della ricerca o dell'inserimento, la cancellazione deve essere utilizzata con con cautela perché potrebbe spezzare l'albero o violare l'ordinamento delle chiavi. Per gestire questa complessità, l'operazione viene suddivisa in **tre casi distinti**, in base al numero di figli del nodo da eliminare:

Caso 1:

Si entra nel **caso 1** solamente quando il nodo u è una **foglia** ed essendo un terminale, la sua rimozione non porta alcun tipo di problemi alla struttura né tanto meno creerebbe orfani. Si procede quindi ad eliminare il nodo e modificare il puntatore del padre di u dall'indirizzo di

u a nil, recidendo il legame.

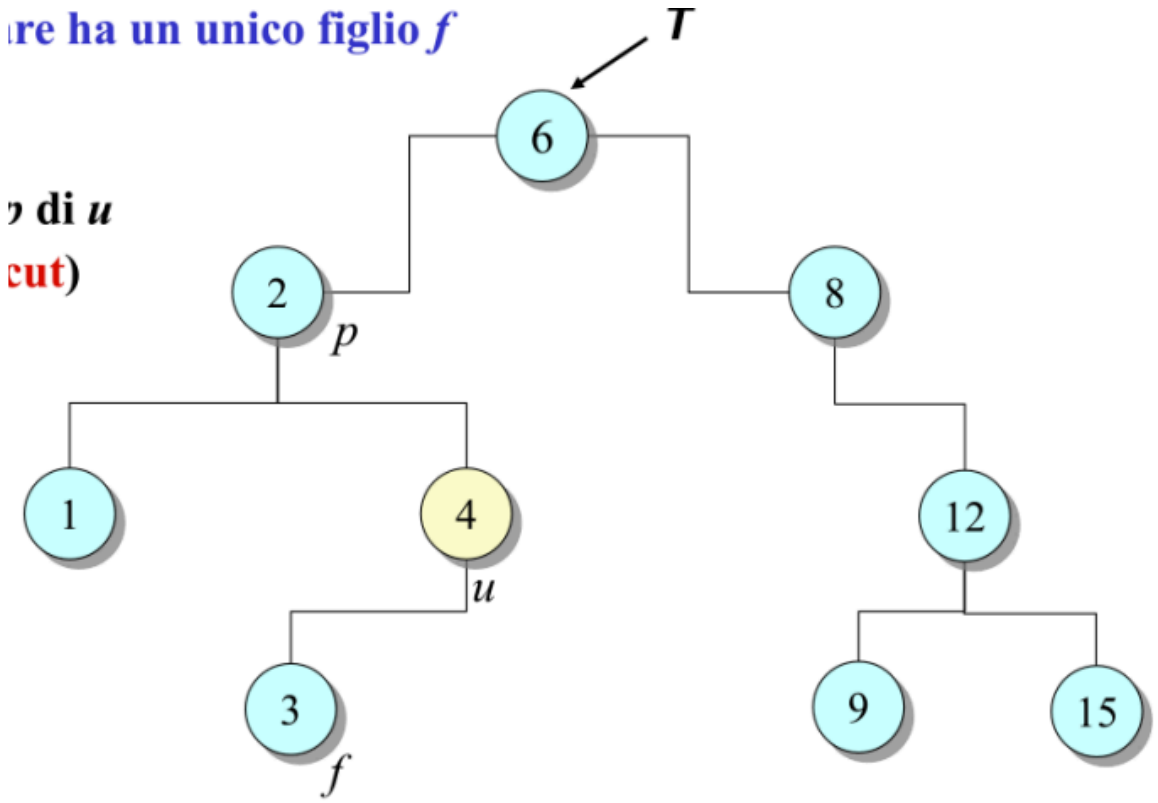


Caso 2:

Il **caso 2** si verifica quando il nodo u, che si intende rimuovere, possiede esattamente un **figlio**. A differenza del caso 1, qui possiamo limitarci a rimuovere il nodo, altrimenti il figlio f e tutto il suo eventuale sottoalbero rimarrebbero scollegati dalla struttura principale.
L'operazione necessaria è una "scorciatoia o **Short-cut**". L'obbiettivo è collegare il nonno direttamente al nipote, saltando il nodo padre che viene eliminato.
Per la risoluzione di questo problema abbiamo 3 passi:

1. Individuare f di u.
2. Attuare la **Short-cut**, il puntatore del padre di u punterà ad f e non più ad u e se u era figlio sinistro di p, f diventa il nuovo figlio sinistro di p e viceversa, il problema della posizione non sussiste avendo un solo figlio, rimpiazza semplicemente il padre.

3. Eliminare l'ormai zombie u.



Dubbio: e se al posto di 3 f fosse stato 0?, avrei avuto a destra un valore minore di quello che p possiede a sinistra, cosa che non può accadere.

Soluzione: c'è una **regola fondamentale** degli Alberi Binari di Ricerca (ABR) che impedisce che questa situazione si verifichi in partenza.
In un ABR, la regola "minore a sinistra, maggiore a destra" non vale solo per i figli diretti, ma è una **proprietà globale ereditaria**.
Se il nodo **4** (u) si trova nel sottoalbero **destro** del nodo **2** (p), significa che **4 è maggiore di 2**,ma significa anche che **TUTTI i discendenti di 4** (figli, nipoti, pronipoti) devono essere **maggiori di 2**.

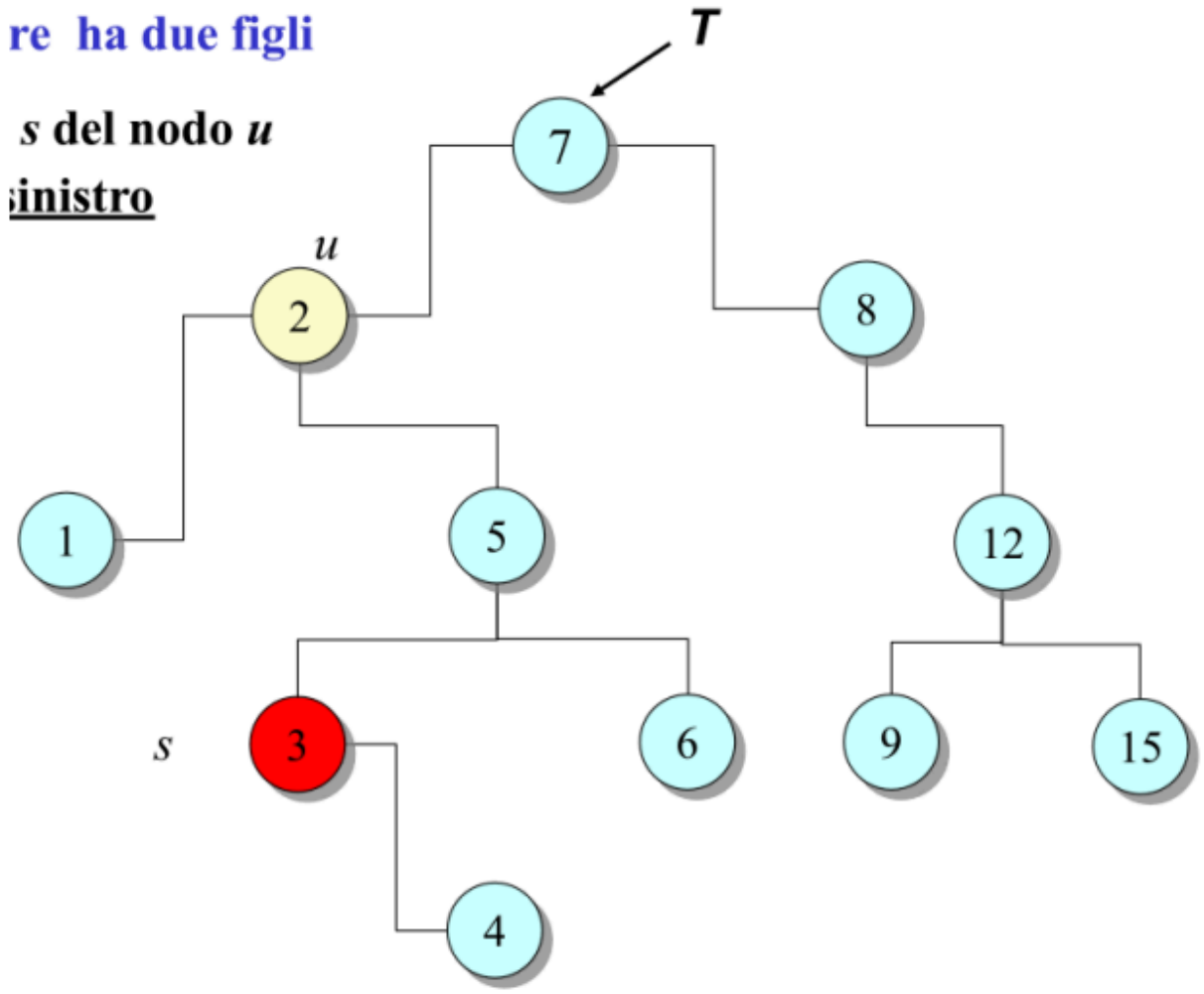
Caso 3:

Il **caso 3** si verifica quando il nodo u che si intende eliminare possiede entrambi i figli, in questo scenario non possiamo applicare la tecnica dello **short-cut** perché il padre di u possiede un solo puntatore disponibile per quella direzione, e non saprebbe come farsi carico di due sottoalberi distinti e indipendenti.

La strategia risolutiva evita di rimuovere fisicamente il nodo u della sua posizione, poiché esso funge da crocevia strutturale importante. L'algoritmo procede invece a sostituire il contenuto informativo di u con quello di un altro nodo che sia **compatibile con l'ordinamento** e **facilmente rimovibile**.
Per essere compatibile con l'ordinamento deve in primis esistere una situazione possibile di

questo tipo, mentre per essere facilmente rimovibile basta che il nodo trovato rispecchi il caso 1 o 2; queste operazioni si racchiudono in **tre passaggi sequenziali** molto rigorosi:

1. **Individuare il successore** s: bisogna in primis trovare il nodo che abbia la chiave immediatamente successiva ad u in tutta la struttura, ovvero s, il quale sarà sempre il nodo con la chiave **minima nel sottoalbero destro**, questo perché essendo s il minimo locale non avrà figli, altrimenti non sarebbe il minimo appunto, eliminando quindi il minimo locale si potrà rientrare nel caso 1 o 2.
2. **Isolamento del successore**: prima di cancellare u, bisogna prima rimuovere il successore dalla sua attuale posizione per non cadere in **duplicati**, per questo si utilizza la stessa short-cut, usata nel caso 2, sul nodo s. Poiché s non ha figlio sinistro, se possiede un eventuale figlio destro, questo viene **adottato** dal padre di s, prendendo il posto di s stesso.
3. **Conclusione di** u: prima che u venga eliminata definitivamente ora si copiano i dati del nodo s in u, in modo tale che la struttura dell'albero rimanga la medesima ma con le informazioni di s sovrascritte in u, cancellando di fatto ciò che prima era in u attraverso le info del suo successore ormai zombie.



Applicando le logiche precedenti allo schema si nota come per trovare il minimo locale dal nodo di nostro interesse (ovvero 2) si vada nel sottoalbero maggiore (destra) per trovare il nodo minimo (il valore più piccolo a sinistra), ove nel nostro caso corrisponde a 3. Trovato il successore ora bisogna capire se utilizzare il metodo 1 o 2; essendo per definizione 3 il minimo locale, non può avere figli sinistri ma nel nostro caso ha 4 che è

figlio destro, ergo si attua il **caso 2**.
Quindi il nodo 5 diventerà il padre del nodo 4, poiché il padre di 4 è stato scollegato, prendendo il posto del nodo sinistro del padre, esattamente come visto nel caso 2.
Per concludere ora basta semplicemente eliminare il nodo 2 tramite sovrascrittura dei dati presenti nel nodo zombie 3, lasciando il posto invariato ma con nuovi dati ed eliminando definitivamente il nodo s, rendendo però la struttura matematicamente ancora valida, poiché $1 < 3 < 5 > 4$.

Implementazione pseudocodice della cancellazione:

```
TREE removeNode(TREE T, Item x){
    TREE u<-lookupNode(T,x)
    if(u!=nil) then
        if(u.left!=nil and u.right!=nil) then{ //caso 3
            TREE s<-u.right
            while s.left!=nil do s<-s.left
            u.key<-s.key
            u.value<-s.value
            u<-s
        }
        TREE t
        if(u.left !=nil and u.right=nil) then t<-u.left //caso 2 solo
figlio sx
        else t<-u.right //caso 1 o caso 2 con solo figlio dx
        link(u.parent,t,x)
        if u.parent=nil then T=t
        delete u
    }
    return T
}
```

Si esegue subito la funzione `lookupNode` per capire se nell'albero T è effettivamente presente il nodo dal valore di x da voler eliminare, se trovato si continua altrimenti restituisce l'albero invariato.

Se il nodo è presente si controlla il caso più complesso, ovvero il caso 3; verificando che il nodo u ha sia figlio destro che sinistro, se il caso è il seguente si va nel sottoalbero destro e si scende sempre più a sinistra fin quando non si trova il minimo `while s.left!=nil do s<-s.left`.
Dopo di che si copia il valore presente in s in u, sovrascrivendo i dati, dopo di che si esegue il passaggio finale, dove i puntatori vengono **scambiati**, quindi la variabile u ora punta ad s, andando a modificare la richiesta, non sarà più u da cancellare ma bensì s.

Ora si crea una variabile `t` di tipo `TREE` che sarà il nuovo albero rimasto che andrà ad incollarsi all'albero originale. In `t` si vedrà se utilizzare il caso 2, in caso ha un solo figlio sinistro, trasformando in questo caso `t` in `u.left` o se usare il caso 1, quindi portando in `t` il valore del nodo destro di u.
Chiamando la funzione `link`, già vista precedentemente, andiamo a *ricucire* l'albero appena modificato. Questa funzione collega il padre del nodo da eliminare direttamente al nodo t, scavalcando di fatto u. Se il nodo eliminato non aveva padre (`u.parent = nil`), significa che abbiamo rimosso la radice e quindi si necessita di una nuova, il puntatore all'albero principale T viene aggiornato per puntare al nuovo albero t.
Alla fine di tutto il controllo, il nodo u viene rimosso fisicamente dalla memoria e l'algoritmo restituisce il puntatore all'albero aggiornato T.

Analisi della complessità per i costi di modifica

La complessità delle operazioni eseguibili su un ABR non dipende dal numero totale dei nodi n, bensì dalla **forma/topologia** dell'albero, definita come la sua altezza h.

A differenza delle strutture lineare, negli ABR avviene una scansione selettiva, dove per ogni **cammino** che non soddisfa la condizione viene eliminato tutto ciò che si trova sotto quel cammino stesso. Di conseguenza la C.C. di tali operazioni è data dall'altezza percorribile dei vari cammini, portando ad una complessità asintotica pari a:

$$T(n) = O(h)$$

L'efficienza reale dipende quindi da come i nodi sono distribuiti:

- **Albero bilanciato o piatto**: si avrà un valore di h relativamente basso, favorendo un alta velocità delle operazioni.
- **Albero sbilanciato o degenere**: si avrà un albero quasi simile ad una lista ed il valore di h tenderà al numero di nodi n, peggiorando così le prestazioni.

Il tipo astratto grafo

Specifica sintattica

Tipo	Definizione
Grafo	il tipo principale che stiamo definendo
boolean	valore di verità usato per i controlli
nodo	l'elemento atomico che costituisce i vertici del grafo
lista	una sequenza di nodi
tipoelem	il tipo di informazione associata al nodo, utile se il grafo (o nodo) è etichettato con nome, numero, oggetto

Categoria	Operatore	Sintassi (Input → Output)	Descrizione
Ciclo di Vita / Stato	`crea`	`()` → `grafo`	Crea un nuovo grafo vuoto (senza nodi né archi). Non richiede parametri.

Categoria	Operatore	Sintassi (Input → Output)	Descrizione
	`vuoto`	`(grafo)` → `boolean`	Restituisce *vero* se il grafo è vuoto (privo di nodi e archi), *falso* altrimenti.
Trasformazione	`insnodo`	`(nodo, grafo)` → `grafo`	Inserisce un nuovo nodo specifico all'interno del grafo.
	`insarco`	`(nodo, nodo, grafo)` → `grafo`	Crea un collegamento (arco) tra due nodi esistenti. Il primo è l'origine, il secondo la destinazione.
	`cancnodo`	`(nodo, grafo)` → `grafo`	Rimuove un nodo specifico e **implicitamente** tutti gli archi a esso collegati.
	`cancarco`	`(nodo, nodo, grafo)` → `grafo`	Rimuove l'arco che collega i due nodi specificati.
Navigazione / Ispezione	`esistenodo`	`(nodo, grafo)` → `boolean`	Verifica se un determinato nodo è presente nel grafo.
	`esistearco`	`(nodo, nodo, grafo)` → `boolean`	Verifica se esiste un arco che collega i due nodi specificati.
	`adiacenti`	`(nodo, grafo)` → `lista`	Dato un nodo, restituisce la lista di tutti i nodi a esso collegati (i suoi "vicini").
Gestione Dati	`legginodo`	`(nodo, grafo)` → `tipoelem`	Legge l'informazione (etichetta) associata a un nodo specifico.
	`scrivinodo`	`(tipoelem, nodo, grafo)` → `grafo`	Associa o modifica l'informazione (*tipoelem*) di un determinato nodo.

Specifica semantica

Categoria	Operatore	Specifica Semantica (Pre e Post Condizioni)
Inizializzazione / Stato	`crea`	**POST:** Restituisce un nuovo grafo $G =< N, A >$ dove l'insieme dei nodi $N = \emptyset$ e l'insieme degli archi $A = \emptyset$.
	`vuoto(G)`	**POST:** Restituisce *true* se $N = \emptyset$ e $A = \emptyset$, altrimenti *false*.
Inserimento	`insnodo(u, G)`	**PRE:** Il grafo esiste e il nodo $u \notin N$ (non appartiene già all'insieme). **POST:** Restituisce G' con $N' = N \cup \{u\}$.

Categoria	Operatore	Specifica Semantica (Pre e Post Condizioni)
	`insarco(u, v, G)`	**PRE:** $u, v \in N$ (esistono) e l'arco $< u, v > \notin A$ (non esiste già). **POST:** Restituisce G' con $A' = A \cup \{< u, v >\}$.
Cancellazione	`cancnodo(u, G)`	**PRE:** $u \in N$. **Vincolo fondamentale:** Il nodo deve essere **isolato** ($\nexists v$ tale che $< u, v > \in A$ oppure $< v, u > \in A$). **POST:** Restituisce G' con $N' = N \setminus \{u\}$.
	`cancarco(u, v, G)`	**PRE:** u, v esistono e l'arco $< u, v > \in A$. **POST:** Restituisce G' con $A' = A \setminus \{< u, v >\}$.
Navigazione	`esistenodo(u, G)`	**POST:** Restituisce *true* se $u \in N$, *false* altrimenti.
	`esistearco(u, v, G)`	**POST:** Restituisce *true* se $< u, v > \in A$, *false* altrimenti.
	`adiacenti(u, G)`	**PRE:** $u \in N$. **POST:** Restituisce una lista L contenente i nodi v tali che esiste un arco uscente da u ($L = \{v \in N \mid < u, v > \in A\}$).
Gestione Dati	`legginodo(u, G)`	**PRE:** Il nodo u deve esistere. **POST:** Restituisce il valore informativo associato a u.
	`scrivinodo(val, u, G)`	**PRE:** Il nodo u deve esistere. **POST:** Aggiorna il grafo associando il valore *val* al nodo u.
	`leggiarco(u, v, G)`	**PRE:** L'arco $< u, v >$ deve esistere. **POST:** Restituisce il valore (peso/etichetta) associato all'arco.
	`scriviarco(val, u, v, G)`	**PRE:** L'arco $< u, v >$ deve esistere. **POST:** Aggiorna il grafo associando il valore *val* all'arco $< u, v >$.

Specifiche

Le specifiche sono delle **etichette o pesi** che vengono assegnate ai nodi e/o agli archi specificando maggiori informazioni, creando i **grafi etichettati**.
Per gestire questi dati non possiamo basarci sulle funzioni di base come `insarco` / `insnodo`, è necessario introdurre nuovi operatori che permettono di **leggere** e **scrivere** le informazioni associate agli archi:

- `leggiarco(nodo,nodo,grafo)->tipoelem`: questa funzione, dato il nodo di ingresso, il nodo di arrivo e il grafo generale, restituisce l'etichetta associata a quell'arco.

- `scriviarco(tipoelem,nodo,nodo,grafo)->grafo` : questa funzione, dato un nuovo tipo di elemento, la coppia di nodi che identifica l'arco e il grafo corrente, restituisce un grafo aggiornato in cui quell'arco presenta la nuova etichetta specificata.

Queste due nuove funzioni estendono la formula matematica di $G = (N, A)$ in modo tale che il grafo diventi una struttura in grado di poter trasportare dati, ciò è essenziale per applicare il **calcolo del cammino minimo**, dove i pesi sono fondamentali.

Rappresentazione con matrice di adiacenza

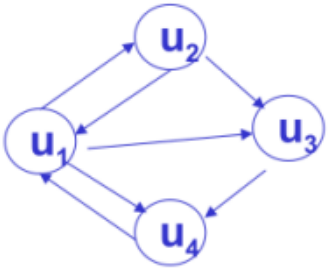
Questa tecnica è la più conosciuta per gestire un grafo in memoria, basata sull'utilizzo di una matrice quadrata E di dimensioni $N \times N$, con N numero di nodi.
Ogni cella e_{ij} della matrice rappresenta la relazione tra la riga (ovvero il nodo i) e la colonna (ovvero il nodo j).

La regola di riempimento per un **grafo semplice** è binaria:

- $e_{ij} = 1$ se esiste un arco che va da i a j con $< i, j > \in A$
- $e_{ij} = 0$ se l'arco non esiste ($< i, j > \notin A$)

La regola di riempimento per un **grafo etichettato** non è più solamente binaria ma memorizza direttamente il peso.
Sia p_{ij} il peso dell'arco $< i, j >$:

- $e_{ij} = p_{ij}$ se l'arco esiste
- $e_{ij} = +\infty$ o $-\infty$ se l'arco non esiste



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

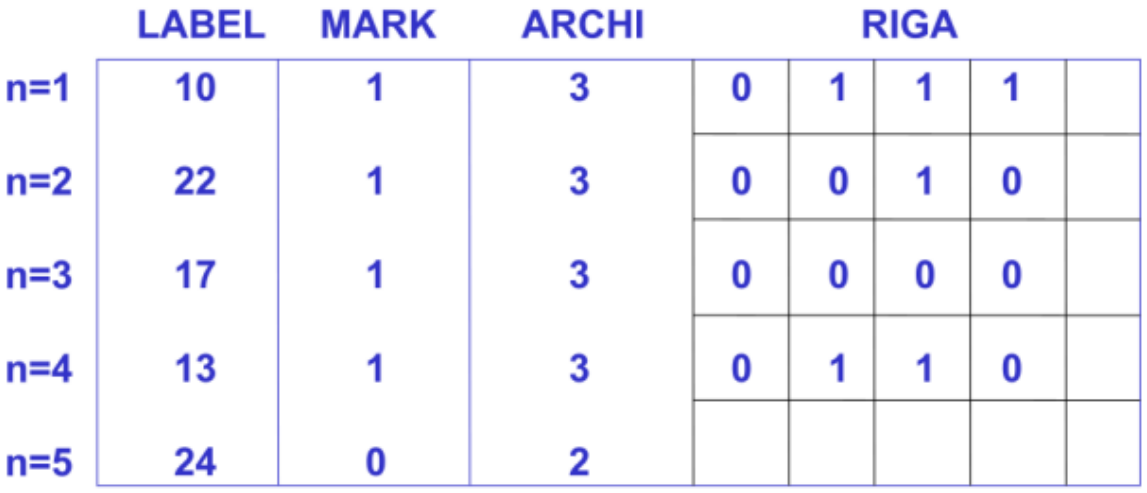
> Nel caso di grafi non orientati, la relazione è **bidirezionale**, in modo tale che la matrice risulti simmetrica rispetto alla diagonale principale. Matematicamente sarebbe $e_{ij} = e_{ji}$ per ogni coppia di nodi, in parole povere se la cella $(1, 2)$ contiene un valore x, quello stesso valore sarà il medesimo presente nella cella $(2, 1)$.

Matrice di adiacenza estesa

Nella versione estesa non si ha più una matrice singola ma quest'ultima è affiancata da **vettori ausiliari** che permettono la gestione delle informazioni del grafo in modo completo.
Ogni riga n della struttura è composta da:

- **LABEL**: Contiene l'etichetta associata al nodo.

- **MARK**: Un flag che identifica lo stato del nodo, se è impostato a zero vuol dire che il nodo non è valido.
- **ARCHI**: Un contatore che memorizza, tramite somma, quanti archi in entrata e quanti in uscita incidono con quel nodo.
- **RIGA**: La sequenza di $0/1$ che rappresenta le connessioni verso gli altri nodi.



	LABEL	MARK	ARCHI	RIGA				
n=1	10	1	3	0	1	1	1	
n=2	22	1	3	0	0	1	0	
n=3	17	1	3	0	0	0	0	
n=4	13	1	3	0	1	1	0	
n=5	24	0	2					

In questa immagine prendiamo come esempio $n = 1$:
La sua **label** è 10, è markata in modo che esista, con valore 1 e la somma degli archi è 3 , infatti dal campo riga possiamo notare che è connessa ai nodi $2,\ 3,\ 4$ lo si nota dagli uno presenti nelle celle corrispondenti al valore, la prima cella vale 1, la seconda è il nodo 2 e così via, infatti qui si ha $[0\ 1\ 1\ 1]$

Il **vantaggio** è avere una C.C. molto bassa pari a $O(1)$ per identificare l'esistenza di un arco, tuttavia è anche **svantaggioso** poiché richiede uno spazio di memoria maggiore pari a $O(N^2)$, specialmente sarebbe molto inefficace coi **grafi sparsi**, grafi in cui sono presenti molti nodi e pochi archi, avendo in questi casi una matrice con quasi tutti zeri.

Rappresentazione con matrici di incidenza

Un grafo $G =< N, A >$ viene rappresentato tramite una matrice B di dimensioni $N \times M$, dove N è il numero dei nodi e M il numero degli archi.

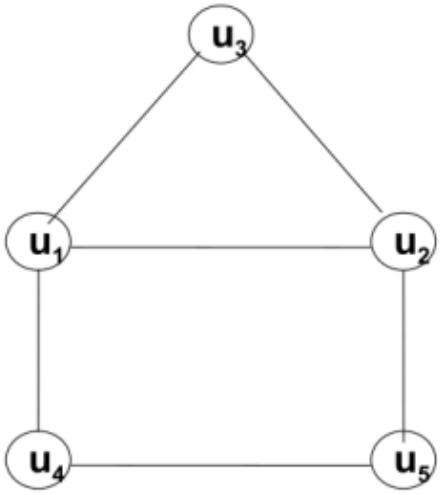
Ogni colonna indica un arco del grafo e descrive quali nodi vengono toccati da esso.

Caso grafo non orientato

In questo caso la relazione di incidenza spiega solamente se un arco tocca un nodo o meno.

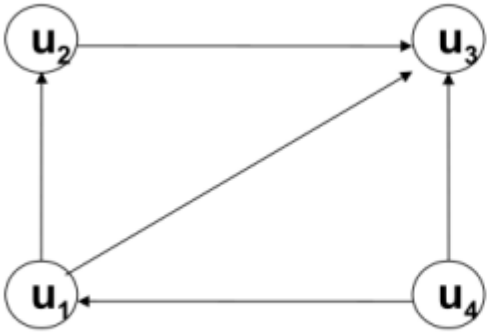
- $b_{ik} = 1$ se l'arco $k-$esimo è incidente al nodo i.
- $b_{ik} = 0$ se l'arco non è incidente.
  Si nota in questo caso una **proprietà** fondamentale, dato che ogni arco collega

esattamente solo due nodi, ogni colonna avrà due valori ad 1 e i restanti a 0.



	<u1,u2>	<u1,u3>	<u1,u4>	<u2,u3>	<u2,u5>	<u4,u5>
1	1	1	1	0	0	0
2	1	0	0	1	1	0
3	0	1	0	1	0	0
4	0	0	1	0	0	1
5	0	0	0	0	1	1

Caso grafo orientato



	<u1,u2>	<u1,u3>	<u2,u3>	<u4,u1>	<u4,u3>
1	-1	-1	0	+1	0
2	+1	0	-1	0	0
3	0	+1	+1	0	+1
4	0	0	0	-1	-1

In questo esempio si hanno quatto nodi e cinque archi.
Prendendo per esempio la riga uno con colonna $< u_1, u_2 >$ al suo interno contiene il valore -1. Infatti nel caso orientato, la matrice usa i valori -1 e $+1$ per indicare il verso della freccia; tornando sempre al nostro caso infatti il -1 indica che l'arco 1 **parte** dal nodo 1, invece se si visualizza la riga del nodo 2 si vede come il valore è $+1$ quindi l'arco 1 **arriva** al nodo 2.
In questo modo comprendiamo i collegamenti tra tutti i nodi presenti.

Lo svantaggio della matrice incidenza

A differenza della matrice con adiacenza, questo nuovo sistema non sfrutta in maniera ottimale e rapida la ricerca tra i nodi vicini $A(u)$.

Se prendiamo l'esempio del grafo ordinato e vogliamo cercare i vicini di u_4 dovremmo eseguire numerosi passaggi per comprenderlo:

1. Bisogna **scorrere tutte le celle orizzontalmente** prima di arrivare alle celle piene della riga 4.
2. Una volta trovate le posizioni in cui è presente un valore, nel nostro caso nella posizione 4 e 5, bisogna **scorre verticalmente tutte le celle** per trovare i corrispettivi segni

opposti associati. Più precisamente nel nostro caso sappiamo solo la **partenza**, scorrendo tutta la colonna troviamo l'**arrivo**.
Possiamo trovare di fatto il $+1$ nella riga 1 e 3 per i corrispettivi valori del nodo 4, per questo possiamo solo ora dire che $u_4 \to u_1$ e $u_4 \to u_3$

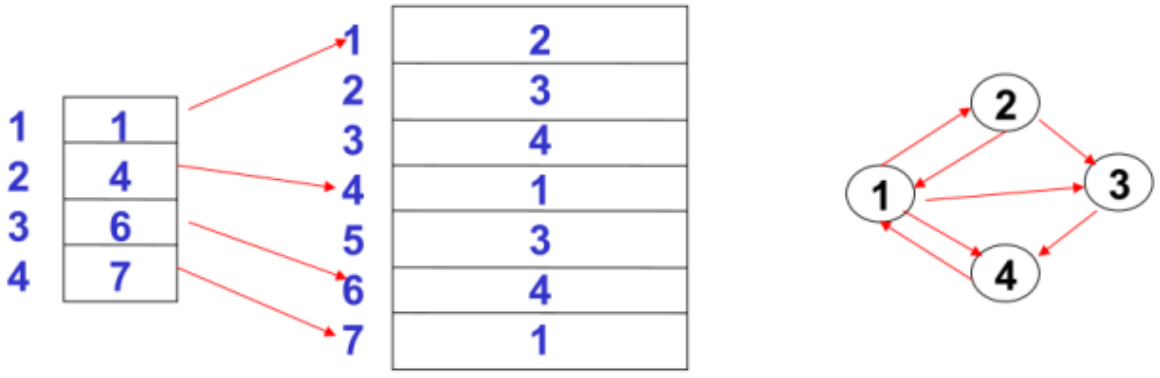
Tutti questi passaggi rendono il costo dell'operazione vertiginosamente alto con una C.C. di $O(m + d \times n)$, con:

- m: il tempo per scorrere la riga.
- $d \times n$: per ogni arco d trovato bisogna scorrere la colonna corrispondente di altezza n.

Rappresentazione con vettore adiacenza

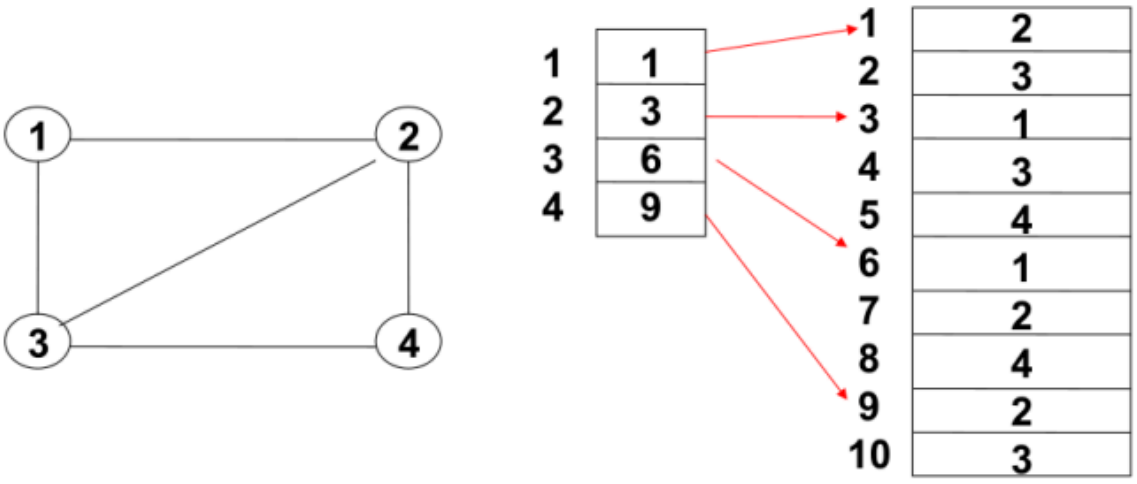
Questa tecnica utilizza due vettori per risparmiare maggiormente lo spazio.
Si ha il **vettore archi** e il **vettore nodi**:

- `ARCHI` : Contiene tutte le destinazioni scritte una dopo l'altra.
- `NODI` : E' utilizzato come vettore "cursore", ovvero ci dice *dove* iniziano e finiscono i vicini di ogni nodi nel vettore `ARCHI` .



Il metodo di lettura di questo grafo è il seguente:
Se nel vettore `NODI` si ha `NODI[1]=1` e il successivo è `NODI[2]=4` , vuol dire che nel corrispettivo vettore `ARCHI` i vicini del nodo 1 vanno dall'indice 1 al $4 - 1$, di fatto poi nel grafo è proprio così.

Come si è potuto notare questo è il caso dei **grafi orientati**, dato che sono presenti le frecce.
Nel caso dei **grafi non orientati** ogni arco deve essere scritto due volte, poiché la direzione essendo priva è biunivoca (da A a B e da B a A).

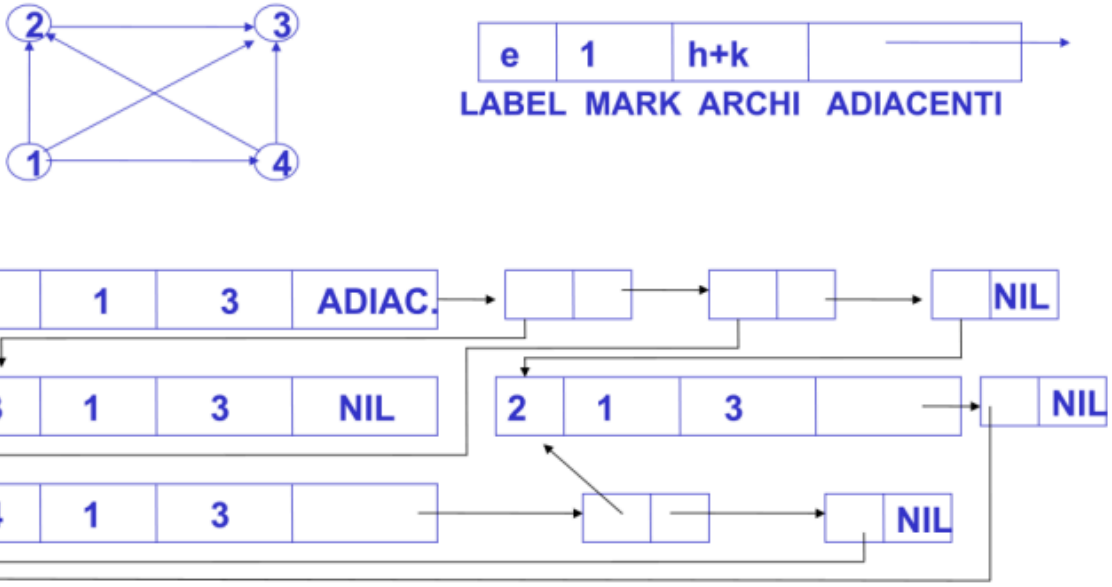


Come si può vedere è sicuramente **compatto** e questo mostra un ottimo punto a favore, tuttavia risulta svantaggiosa la sua struttura **statica**, se volessimo aggiungere un nuovo arco bisognerà shiftare tutti i dati successivi nei vettori, creando una procedura onerosa.

Rappresentazione con lista di adiacenza

La lista ad adiacenza è una delle soluzioni più bilanciate. In questa versione si ha un vettore madre in cui ogni cella contiene un **puntatore** all'inizio di una **lista concatenata**.
Ogni lista contiene solo i nodi effettivamente collegati tra loro, senza avere *zeri* inutili, è un ottima soluzione per i grafi sparsi.

Rappresentazione con struttura a puntatori



Questa versione è totalmente **dinamica** e priva di array iniziali.
Il nodo v con etichetta e, h archi entranti e k archi uscenti è rappresentato mediante un **record**.

Visita di un grafo

La visita di un grafo è un **metodo sistematico per esplorare la struttura**, assicurandosi di *visitare* ogni nodo e ogni arco almeno una volta.
Affinché la visita sia completa ed efficace, è necessario considerare le proprietà di connessione del grafo:

- **Grafo connesso**: Un grafo si dice connesso se dati due nodi u e v, esiste sempre un cammino che li collega.
- **Grafo fortemente connesso**: Un grafo si dice fortemente connesso se per ogni coppia di nodi u e v, esiste sia un cammino da u a v che un cammino da v a u.

Gli algoritmi di visita che possiamo usare per i grafi sono il **DFS** e **BFS** già incontrati durante lo studio degli alberi, infatti essi usano lo stesso meccanismo.

Per quanto riguarda l'algoritmo **DFS** la logica del LIFO rimane invariata, cambiando la gestione della memoria per evitare i cicli. Ogni grafo può contenere cicli, per questo è importante marcare ogni nodo con *visitato* appena lo si incontra, altrimenti si andrà incontro ad un loop.
L'algoritmo sfrutta anche una soluzione di **backtracking**, quando un nodo non ha più vicini da visitare allora l'algoritmo tornerà indietro, in modo tale da poter percorrere percorsi alternativi.
Pseudocode:

```
DFS(G: GRAFO;u:NODO)
    ESAMINA IL NODO u E MARCALO "VISITATO"
    for(TUTTI I v ∈ A(u)) do
```

```
        ESAMINA L'ARCO (u,v)
        if (v not "VISITATO") then
            DFS(G,v) //chiamata ricorsiva immediata
```

Se il grafo non è connesso, questa funzione va lanciata ciclicamente su ogni nodo non ancora visitato per scoprire tutte le "*isole*".

Per quanto riguarda l'algoritmo **BFS** segue sempre la struttura FIFO degli alberi ma assume un nuovo significato utilitaristico.
Nei grafi il BFS permette di trovare il **cammino minimo**, ovvero il percorso con minor numero di archi partendo dalla sorgente. La **distanza** è data dal livello di distanza percorsa, se dopo due salti ($A \to B \to C$) si trovano tutti nodi allora abbiamo trovato già la distanza migliore.
Anche in questo caso bisogna fare attenzione a non inserire nella coda nodi che sono stati già visitati o già presenti in coda da visitare, altrimenti aumenterà il livello di complessità.
Pseudocode:

```
BFS(G:GRAFO; u:NODO)
    CREACODA(Q)
    INCODA(u,Q) //Inserimento del nodo sorgente
    while not(CODAVUOTA(Q)) do
    u=LEGGICODA(Q)
    FUORICODA(Q) //Estrazione del nodo più vecchio
    ESAMINA u E MARCALO "VISITATO"

    //Espansione della frontiera
    for (TUTTI I NODI v ∈ A(u)) do
        esamina l'arco (u,v)
    //Se il nodo non è mai stato visto e non è già in coda
    if(v non è marcato "visitato" and v ∉ Q) then
        INCODA(v,Q)
```

La condizione $v \notin Q$ rappresenta un abuso di notazione, in quanto le code standard non supportano la verifica efficiente dell'appartenenza. A livello implementativo, si suggerisce di utilizzare un insieme aggiuntivo o un array di flag per tracciare i nodi già accodati.

Algoritmo generale di visita

Per sfruttare la logica del BFS e del DFS allo stesso tempo si utilizza un algoritmo che le unifica in uno schema generale. La procedura si basa sull'uso di **tre schemi logici** per gestire lo stato dell'esplorazione:

- **Insieme** R: è l'insieme contenente tutti i **nodi già visitati** definitivamente.
- **Insieme** Q: è l'insieme che contiene tutti i **nodi scoperti e utili per il proseguimento della ricerca** ma che non sono stati ancora completamente processati e visitati del tutto.
- **Insieme** B: è l'insieme contente tutti gli **archi utilizzati** per la scoperta di nuovi nodi.

Il funzionamento dello pseudocodice presente si basa su un ciclo che prosegue fino all'esaurimento del contenuto di Q:

```
Q={nodo_iniziale};
R={};B={};
repeat
    "SCEGLI UN NODO j DA Q"
    if(ESISTE UN NODO k ADIACENTE A j NON ANCORA VISITATO) then
        "VISITA k"
        "AGGIUNGI k AGLI R, Q"
        "MEMORIZZA L'ARCO (j,k) NELL'INSIEME B"
    else "ELIMINA j DA Q" //il nodo j è stato completamente esplorato
until(Q={})
```

Il punto cruciale di questo algoritmo si basa nel punto *scegli un nodo da* j *da* Q. Il criterio di scelta del nodo stabilisce la modalità di visita:

- DFS, se l'insieme Q viene gestito con la logica **LIFO**, scegliendo il nodo visitato **più recentemente**, simulando così la discesa in profondità tipica della ricorsione.
- BFS, se l'insieme Q viene gestito con la logica **FIFO**, scegliendo i nodi in **ordine di inserimento**, garantendo che i nodi vengano visitati per distanza crescente dal punto di partenza fornendo sempre il minor numero di passi.