**STRUTTURE DI DATI**

Per struttura di dati si intende un' aggregazione di dati.

Le strutture dati possono essere:

- Astratte

- Concrete

Una **struttura dati astratte** può essere definita come un insieme di dati tra i quali viene stabilita una relazione logica.

Esempi.

1. Per memorizzare i voti di 24 studenti di una classe è necessario utilizzare un vettore monodimensionale, che permetta di individuare in modo veloce il voto di uno studente conoscendo la sua posizione all’interno del vettore .

2. Per memorizzare i dati relativi al fatturato di ogni mese in 10 filiali di un'azienda è necessario utilizzare una struttura a "doppia entrata" (matrice) .

3. Per memorizzare i dati relativi a 100 prodotti commercializzati da un'azienda servirà una struttura tabellare.

4. per memorizzare gli utenti in coda ad uno sportello cosa utilizzaremo? E per rappresentare un albero genealogico?

Anche questi sono dati che sono collezionati allo scopo di utilizzarli ma hanno una gestione diversa rispetto, per es, ad un vettore (una coda di persone scorre e chi viene servito deve essere eliminato) Questi ultimi casi devono essere gestiti con una organizzazione logica particolare che non ha una variabile corrispondente nel linguaggio (qualche volta si ma nel 90% dei casi no). Per permettere la memorizzazione delle strutture di dati astratte, la memoria viene gestita utilizzando delle apposite

organizzazioni dette **strutture concrete di dati**.

Le strutture concrete di dati riguardano quindi la rappresentazione delle strutture astratte in memoria centrale, ossia

riguardano l'**allocazione fisica di memoria**.

Per ogni struttura astratta scelta per gestire una particolare situazione, dovrà poi essere utilizzata, in fase di programmazione, una opportuna struttura concreta.

**STRUTTURE ASTRATTE DI DATI**

**- Liste non sequenziali**

- **Pila**

- **Coda**

- **Grafo**

- **Albero**

**Una lista è un insieme di elementi per i quali può essere data una “legge d’ordine” che fa si che (tranne che per il primo e l’ultimo) per ogni elemento è possibile individuare un predecessore e un successore. (la lista della spesa, le parole di una frase, …)**

L’insieme di elementi che costituiscono una lista, quando devono essere memorizzati (informaticamente) utilizzano le

LISTE CONCATENATE.

Nelle **liste non sequenziali** (**concatenate**) gli elementi che la compongono non vengono memorizzati in memoria centrale uno di seguito all'altro così che l’organizzazione logica non sia coincidente con quella fisica (come succede per gli elementi dei vettori che sono sia fisicamente che logicamente contigue ovvero vicine), ovvero vengono collocati in memoria in posizioni non necessariamente contigue. L'ordinamento logico viene garantito da un **puntatore**, che indica la posizione di memoria centrale nella quale si trova l'elemento che logicamente dovrebbe essere il successivo.

Ogni elemento della lista è quindi un **record** con almeno 2 campi:



Il campo DATO contiene l’informazione che interessa mentre il campo PUNTATORE contiene l’indirizzo dell’elemento da considerare come successivo.

Il puntatore dell’ultimo elemento della lista contiene il valore *nil (o null)*, che rappresenta il puntatore nullo ovvero a “nessun elemento”.

Una lista gestisce la memoria in **modo dinamico**. Quando serve un nuovo elemento, viene fatta una richiesta (in genere l’istruzione è detta NEW) di uno spazio di memoria centrale necessario a contenere l’oggetto, il sistema operativo riserva (allocare) una porzione di memoria (prelevandola da una zona particolare detta HEAP) in modo che altri utenti non possano accedervi.

Quando un oggetto non serve più, viene fatta una deallocazione (istruzione DISPOSE O FREE) della memoria che conteneva l’oggetto, quindi viene liberata la memoria e restituita al sistema operativo in modo che altri utenti la possano utilizzare.

La **gestione dinamica** della memoria si contrappone alla **gestione statica**, nella quale la memoria viene allocata all’inizio del programma e viene deallocata alla fine del programma (esempio l’uso della istruzione DIM che in caso di allocazione dinamica non viene proprio utilizzata).

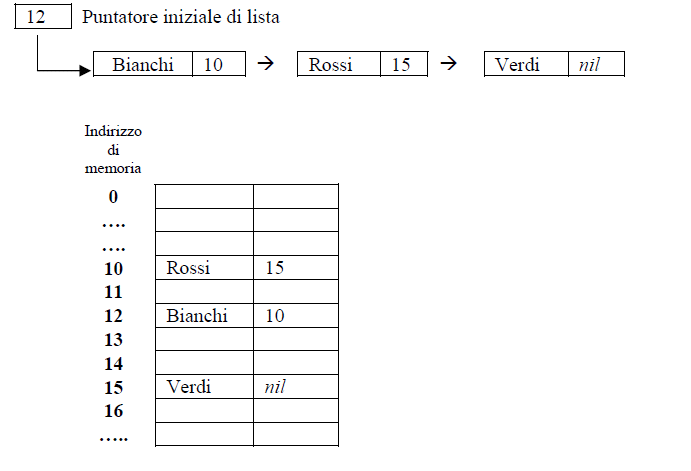
Una gestione dinamica della memoria è preferibile in quanto permette in ogni momento di allocare esattamente la memoria che serve, senza *prenotarla* (DIM)all’inizio.

Esempio:

La seguente lista di nomi:

* Rossi che si trova all’indirizzo di memoria 10
* Bianchi che si trova all’indirizzo di memoria 12
* Verdi che si trova all’indirizzo di memoria 15

sarà essere memorizzata nel seguente modo in memoria centrale se vogliamo garantire l’*ordinamento alfabetico* dei nomi (ovvero avere questa **Legge D’Ordine**



Operazioni di **inserimento** di elementi in una lista.

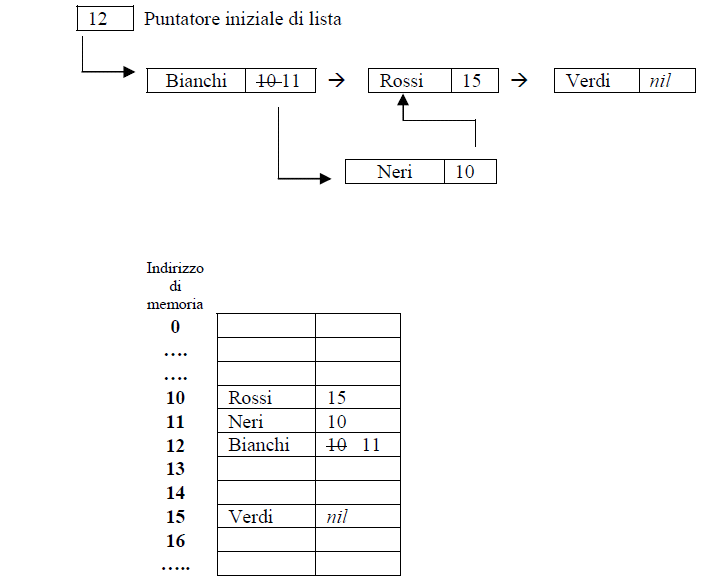
L’operazione di inserimento deve permettere di caricare in memoria un elemento in una posizione libera e deve poi garantire l’ordinamento logico degli elementi *aggiornando* i puntatori.

Esempio.

Supponiamo di voler aggiungere alla lista precedente il nome

* Neri che viene collocato all’indirizzo di memoria 11

La lista dovrà essere aggiornata nel seguente modo



Operazioni di **cancellazione** di elementi da una lista.

L’operazione di cancellazione deve permettere di eliminare dalla lista l’elemento stabilito continuando a garantire

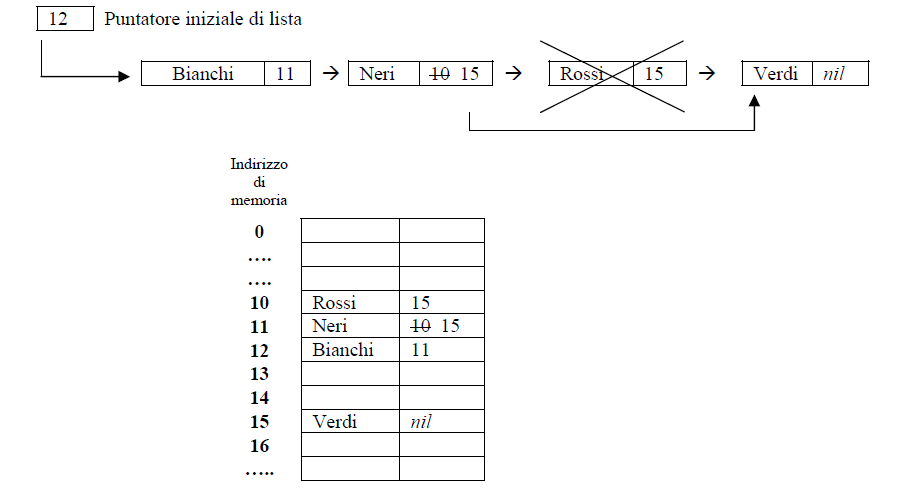
l’ordinamento logico degli elementi *aggiornando* i puntatori.

Esempio.

Supponiamo di voler eliminare dalla lista precedente aggiornata il nome

* Rossi che si trova all’indirizzo di memoria 10

La lista dovrà essere aggiornata nel seguente modo



Ovviamente cancellazioni ripetute provocano un dispendio di spazio (se re-inserisco Rossi, questo verrà inserito, ad es. nella posizione 16 e non nella 10) per cui ad un certo punto ci sipotrebbe ritrovare nella situazione che lo spazio risulta tutto occupato (in teoria). Alle normali routine di gestione della lista si aggiunge allora quella per la GARBAGE COLLECTION ovvero quella che si preoccupa di tener traccia delle posizioni che sono inutilizzate per poterle riutilizzare al posto di nuove posizioni.

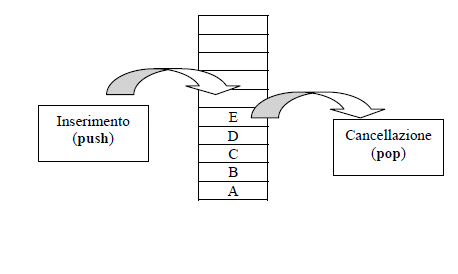
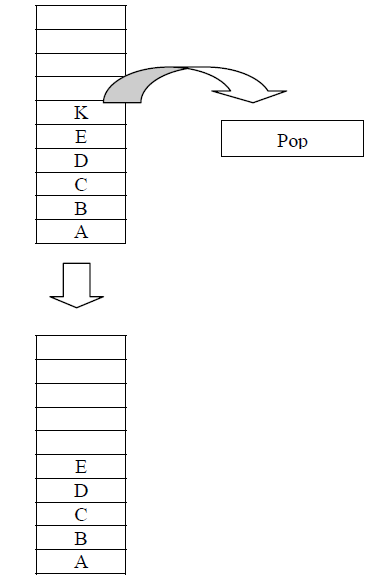
**Pila**

La struttura di dati **pila** (**stack** in inglese) rappresenta un insieme di elementi nel quale le operazioni di inserimento e

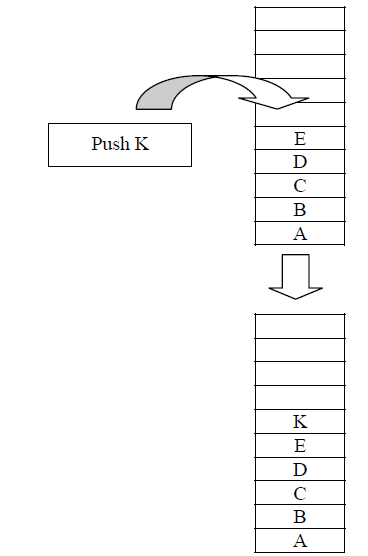
cancellazione seguono una politica **LIFO** (**Last In First Out**). Ciò significa che le operazioni di inserimento di nuovi

elementi (detto **push**) e l’operazione di cancellazione di elementi (detta **pop**) avvengono ad un solo estremo della pila e quindi che l’oggetto cancellato è quello che è stato inserito per ultimo.

Esempio



L’operazione **Pop** agisce nel seguente modo

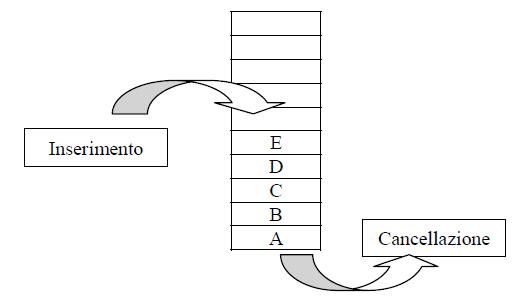
L’operazione **Push(**K) agisce nel seguente modo

**La coda**

La struttura di dati **coda** (**queue** in inglese) rappresenta un insieme di elementi nel quale le operazioni di inserimento e

cancellazione seguono una politica **FIFO** (**First In First Out**). Ciò significa che dire l’inserimento e la cancellazione di

elementi avvengono agli estremi opposti e quindi che l’oggetto cancellato è quello che è stato inserito per primo.



La gestione di questo tipo di struttura è utilizzata in ambito informatico, per esempio, dal sistema operativo nella gestione di più stampe sulla stessa stampante. Se si richiedono più stampe, esse vengono eseguite nell’ordine in cui sono state richieste:

la **prima** stampa inviata sarà la **prima** effettivamente stampata (first in, first out). Per realizzare questo meccanismo il

sistema operativo gestisce una **coda di SPOOL.**

**Nota bene:** se push necessita dell’indicazione dell’elemento da inserire (quel K fra parentesi), la pop non ne necessita in quanto si sa perfettamente che cancellare e dov’è (a differenza della lista semplice in cui inserimenti e cancellazioni possono essere effettuate ovunque)

**Per gestire una pila o una coda si può utilizzare (in modo astratto) una lista, oppure una struttura concreta come una tabella che però ha una gestione logica che prevede che gli inserimenti e le cancellazioni possano avvenire dagli estremi indicati.**

Come simulare una pila o una coda  
La più semplice implementazione si basa su un vettore e su un indice che tiene traccia dell’indice corrispondente all’ultimo elemento inserito (per la pila) e un vettore e 2 indici corrispondenti alla posizione dell’elemento in testa e in coda (per la coda)

Se per la pila la gestione è facile e l’unico problema è dato dalla dimensione dichiarata della tabella (se dichiaro di 10 allora avrò spazio solo per 10 elementi “impilati”) per la coda, con inserimenti e cancellazioni da estremi opposto il problema è differente. Infatti si dice che la coda “cammina”….

Partiamo da una coda in cui facciamo Push (A), Push(B) e Push (C)

A

B

C

Avremo che IC (inizio coda) = 0 e FC (fine coda) = 2

Facciamo successivamente Push (D), Pop, Pop, Push(E)

-

-

C

D

E

Avremo che IC=2 e FC=4 e contunuando con inserimenti e cancellazioni potremmo ritrovarci all’ultimo elemento avendo però elementi liberi all’inizio. Per questa ragione la coda, nella struttura, verrà gestita “circolarmente” ovvero dopo aver riempito l’elemento finale si cercherà spazio cominciando daccapo. In particolare c’è ancora spazio nella coda “circolare” se IC < (FC+1) mod 10 (nel nostro esempio se IC rimane sempre 3, allora posso aggiungere finchè FC non diventa 2)

**Il grafo**

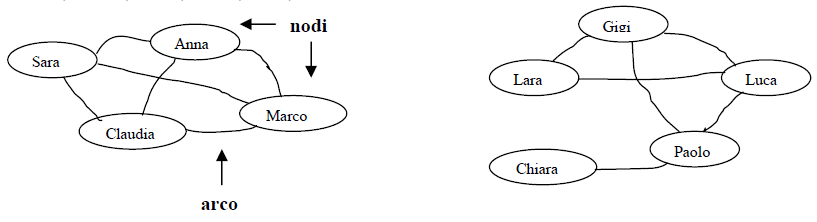
Il **grafo** è una struttura dati astratta utilizzata per rappresentare, in modo grafico, *relazioni* tra elementi.

Un **grafo** può essere definito come un insieme di **nodi** collegati da **archi**.

Esempio 1.

Si vuole rappresentare la relazione *“essere amico di”* all’interno di un gruppo costituito dalle seguenti persone: Lara, Marco,

Gigi, Claudia, Chiara, Paolo, Anna, Luca, Sara.



La relazione considerata ha la caratteristica di essere **simmetrica** o **biunivoca** (se Gigi è amico di Lara, anche Lara è amica

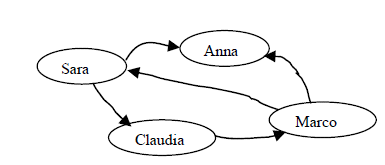
di Gigi !), questo significa che non è necessario assegnare un *verso* agli archi che collegano i nodi: in questo caso si parla di

**grafo non orientato.**

Esempio 2.

Consideriamo ora la relazione *“ritenere simpatico”* all’interno di un gruppo costituito dalle seguenti persone: Marco,

Claudia, Anna, Sara.



In questo caso la relazione **non è simmetrica** (Sara ritiene simpatica Claudia, ma Claudia non ritiene simpatica Sara!): il grafo utilizzato per rappresentarla viene detto **grafo orientato**.

All’interno di un grafo, la successione di nodi che devono essere visitati per raggiungere un nodo prefissato si chiama

**cammino** (in inglese **path**).

All’interno di un grafo si definisce **circuito** un qualsiasi cammino chiuso che permetta, partendo da un nodo, di ritornare sul medesimo nodo senza percorrere lo stesso tratto due volte.

Nell’esempio 2, il cammino Sara -> Claudia -> Marco -> Sara è un circuito.

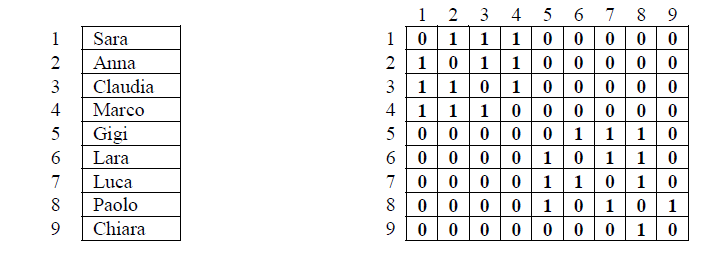
Se, fissati due nodi qualsiasi di un grafo, è sempre possibile trovare un cammino che li congiunga, si dice che il grafo è **connesso**. Se ci sono tutti gli archi che collegano i vari noti si dice che è completamente connesso

Il grafo dell’esempio 1 **non è connesso** perchè è possibile individuare 2 sottoinsiemi non collegati da alcun cammino

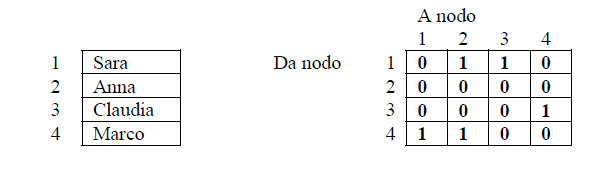
(l’insieme costituito da Sara, Anna, Claudia e Marco e l’insieme costituito da Gigi, Lara, Luca, Paolo, Chiara).

Un grafo può essere rappresentato in memoria centrale (struttura concreta) attraverso una matrice quadrata (matrice quadrata è una matrice nella quale il numero di righe e colonne è uguale), in cui il numero di righe e colonne coincide con il numero di nodi, all’interno della matrice verrà memorizzato un **1** se i due nodi sono collegati da archi, uno **0** se i due nodi non sono collegati da alcun arco: una matrice utilizzata per questo scopo viene detta **matrice di adiacenza**.

Matrice di adiacenza associata al grafo dell’esempio 1 (grafo non orientato).



Matrice di adiacenza associata al grafo dell’esempio 2 (grafo orientato).



**L’albero**

Un struttura dati **albero** può essere definita come un **grafo connesso privo di circuiti e** valgono le seguenti proprietà:

* l’albero contiene n − 1 archi
* esiste un solo percorso semplice tra ogni coppia di nodi dell’albero
* se si rimuove un arco qualsiasi dell’albero, la struttura risultante non `e piu’ connessa, ma composta da due alberi distinti

L’albero si dice **ordinato** se in ciascun livello si considera significativo l’ordine con cui compaiono i nodi

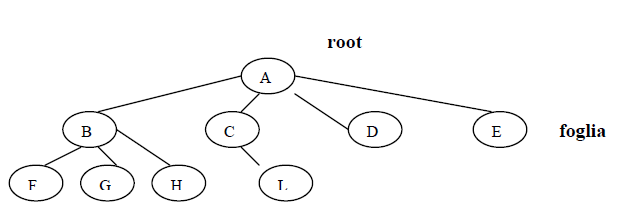
Una struttura ad albero viene utilizzata in tutte le situazioni nelle quali è possibile determinare delle gerarchie di elementi: es.

*organizzazione di un libro in capitoli, sottocapitoli, paragrafi; organigramma delle responsabilità e delle dipendenze*

*all’interno di un’azienda, l’organizzazione delle cartelle nel disco fisso, ecc...*

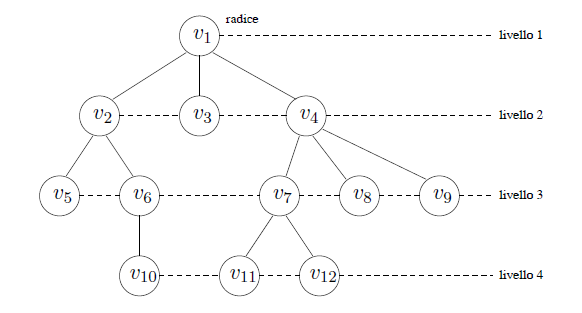
Gli elementi della relazione sono detti **nodi**, mentre con **ramo** si indica un cammino composto da più nodi connessi. Un nodo posto al termine di un ramo è chiamato **foglia**, mentre il nodo dal quale partono tutti i rami viene detto **radice** (in inglese **root**). Un’altre terminologia usa le parole **padre** e **figli**

Esempio



Assegnato un albero, scelto un nodo arbitrario come radice, si possono ordinare i suoi nodi in base al relativo livello:

* il livello della radice è 1
* il livello di ogni altro nodo è pari al numero di nodi contenuti nel percorso tra quel nodo e la radice
* nota la radice, è anche nota la suddivisione in livelli



In informatica, l’organizzazione dei file su disco, gestita dal *file system* del sistema operativo, rappresenta un esempio di albero, nel quale la radice è la directory principale (detta appunto **root** ), i nodi sono le cartelle (directory) e le foglie sono i file o le sottodirectory vuote.

Il **grado** invece è il numero di “figli” massimo che un nodo ha all’interno dello stesso albero. Per quello sopra: c’è chi ha grado 0 (es v3, v10…) chi ha grado 1 (es v6), che ha grado 2 (v2, v7) e chi, infine, grado 3 (es v1 e v4). Il grado dell’albero è dunque 3

Definizione ricorsiva

Si dice così (ricorsiva) poichè espressa in funzione di altri alberi, non fa riferimento agli archi

**un albero è un insieme costituito da uno o più nodi tale che:**

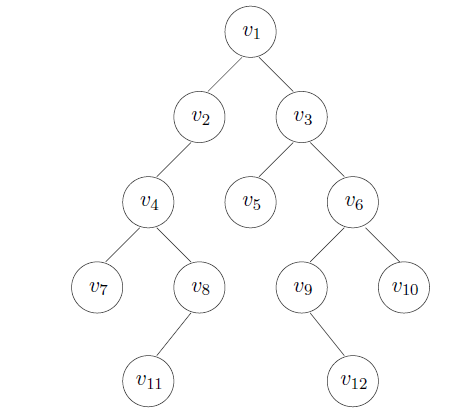
* **un particolare nodo è designato come radice**
* **i rimanenti nodi, se esistono, possono essere suddivisi in insiemi disgiunti, ciascuno dei quali à a sua volta un albero detto sottoalbero (si noti che ciascun sottoalbero ha una propria radice)**
* **il grado di un nodo è il numero i sottoalberi del nodo stesso**

Visita degli alberi

La visita di un albero consiste nell’esaminare tutti i suoi nodi uno per uno in ordine appropriato

**soluzione banale e fortemente inefficiente:** ripartire ogni volta dalla radice, dopo aver esaminato un nodo

**soluzione migliore**

sono stati proposti metodi che prevedono un **ordinamento di visita** particolarmente efficienti per alberi ordinati: questi metodi si basano su sequenze di azioni di natura ricorsiva, si distinguono per il tempo di esame di ogni nodo rispetto ai suoi sottoalberi

Per semplicità spiegheremo questo metodo più efficiente solo per **l’albero binario**

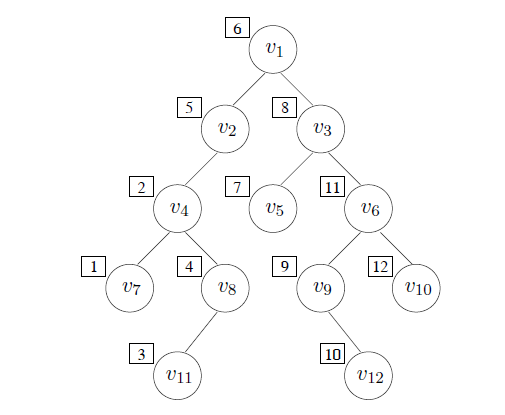
**Albero binario**

Un albero binario è un insieme di nodi, tale che:

* un particolare nodo, se il numero dei nodi e diverso da zero, è designato come radice
* i rimanenti nodi, se esistono, possono essere suddivisi in due insiemi disgiunti, ciascuno dei quali è a sua volta un albero binario (sottoalbero sinistro e destro)

L’albero binario non è un caso particolare di albero, per i seguenti due motivi:

* un albero binario può essere vuoto, mentre un albero deve contenere almeno un nodo
* ciascuno dei due sottoalberi della radice conserva la propria identità di sottoalbero destro e sinistro anche se l’altro sottoalbero è vuoto

**Visite**

La visita in **ordine simmetrico** prevede:

* visita il sottoalbero sinistro, in ordine simmetrico
* esamina la radice
* visita il sottoalbero destro, in ordine simmetrico

la scrittura equivalente alla sequenza di visita è la sequente:

v7 v4 v11 v8 v2 v1 v5 v3 v9 v12 v6 v10

E’ inoltre possibile visitarlo in ordine **anticipato** e **differito**

(sequenza di visita per l’albero dell’esempio)

ordine anticipato

La visita in **ordine anticipato** prevede:

* esamina la radice
* visita il sottoalbero sinistro, in ordine **anticipato**
* visita il sottoalbero destro, in ordine **anticipato**

v1 v2 v4 v7 v8 v11 v3 v5 v6 v9 v12 v10

ordine differito

v7 v11 v8 v4 v2 v5 v12 v9 v10 v6 v3 v1

Infine per rappresentare un albero in modo concreto possiamo usare un vettore (o una tabella) con questa regola che diamo solo per gli alberi binari: se un nodo è in posizione i, il suo figlio sinistro è in posizione (2\*i) e quello destro in (2\*i)+1

Es. valido per l’albero qui a fianco

V1

V2

V3

V4

V5

V6

V7

V8

V9

v10

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20