Trees

Concetto di albero

Def. Albero: *ADT utilizzato per memorizzare un insieme di elementi fra i quali è possibile stabilire una* ***relazione gerarchica****.*

Se ho dei dati da archiviare e su questo insieme devo fare delle operazioni, l'albero è un modo per rappresentare a livello astratto questi dati se c'è una relazione gerarchica.

Esempi di alberi sono gli alberi genealogici, l'indice di un libro (sezioni, capitoli e sotto-capitoli).

Def. 1 formale*:un albero è una coppia ordinata (V,E) di insiemi dove:*

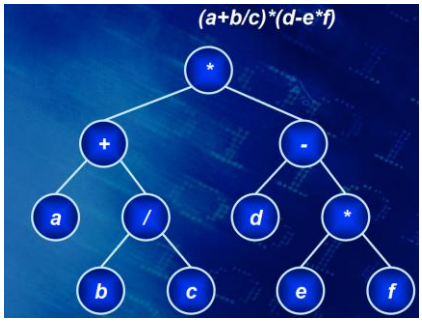
- V è un insieme finito e non vuoto di oggetti, denominati nodi(o vertici) tra il quale esiste una forma di relazione. Un nodo viene distinto dagli altri e viene chiamato radice.

- E : è l'insieme degli archi.Ciascun arco unisce due nodi (no archi flottanti o connessione a tre).

Nei nodi vengono memorizzati sia l'elemento che associo a quel nodo sia le informazioni necessarie a ricostruire la struttura (ovvero a chi è collegato ciascun nodo).

Def. 2 formale: *è un insieme di nodi, c'è un nodo di partenza chiamato root, ogni nodo ha un nodo padre (a eccezione del root); i nodi possono avere un numero qualunque di figli.*

Def. assiomatica*: il singolo nodo è di per sé un albero, supponendo che n sia un nodo e che* T_1, .. T_k *siano alberi con radici* $n_1 \ldots n_k$ *è possibile costruire un nuovo albero facendo in modo che n sia il padre di* $n_1 \ldots n_k$.



I nodi degli alberi che non hanno sotto-alberi (o figli) sono le **foglie.**

Problema: voglio rappresentare espressioni aritmetiche usando un albero. Posso fare in modo che le foglie siano gli operandi, i nodi intermedi le operazioni da effettuare e la struttura dell'albero mi dà la struttura dell'espressione.

Def. Se esiste un arco che collega il nodo a e b e il nodo a "sta sopra" allora a è il padre di b e b è un figlio di a.

Def. **Path**: dato un nodo p, il cammino che porta a quel nodo è la sequenza di nodi che partendo da root mi fa arrivare a p. in modo che n_1 sia figlio di root, n_2 siafiglio di n_1 etc. (devo sempre scendere, non posso mai salire o tornare indietro).

Generalizzando, il cammino da un nodo n_1 a un nodo n_ke' una sequenza di nodi $n_1,n_2…n_k$ tale che n_{j+1}e' un figlio di n_j, qualunque j>=1 e <k.

Se esiste un cammino che collega a->b allora a è ascendente (ancestor) di b, e b è il discendente (descendant) di a.

Teorema: se esiste un cammino fra a e b, allora il cammino è unico.

Profondità di un nodo:è la lunghezza dell'unico cammino che porta dalla radice al nodo scelto (c'è né sempre uno perché l'albero non è sconnesso). Inoltre la profondità di root è pari a 0.
Livello: vengono utilizzatedue diverse convenzioni, o livello=profondità (->root ha level 0) o livello=profondità +1 (->root ha livello 1).

Altezza di un nodo: è lunghezza del massimo cammino dal nodo a una foglia (lunghezza del cammino tra il nodo è la foglia più lontana sua discendente).
Altezza dell'albero: altezza della sua radice.

Nel caso in cui tutte le foglie abbiano la stessa profondità e l'altezza dell'albero sia H, per ogni nodo i si ha che:

$p(i)+h(i)=H$ con p(i): profondità del nodo i; h(i): altezza del nodo i.

Grado di un nodo: numero di discendenti diretti di quel nodo, il numero di archi che escono da quel nodo. Il grado dell'albero è il massimo grado dei suoi nodi. **Un albero di grado 2 è detto albero binario** (da ogni nodo escono o nessuno, 1 o 2 archi). L'albero binario può utilizzato per memorizzare un'espressione booleana.

Lunghezza del cammino interno: la somma delle profondità dei nodi di un albero.

Ordine: solitamente si assume un ordine implicito, i figli implicitamente vengono ordinati da sinistra a destra. In un albero genealogico per esempio scelgo di ordinare i figli da sinistra a destra in base all'età.

Operazioni sugli alberi

Sia n un generico nodo e T l'albero sul quale stiamo operando. Allora:

- ***Parent(n,T):*** torna il padre del nodo n nell'albero T, un nodo nullo se il nodo è la radice.
- ***Left_most_child(n,T)***: torna il figlio di n più a sinistra, se n è una foglia torna un nodo nullo.

- ***Right_sibling(n,T)***: torna il fratello a destra del nodo n nell'albero T, nodo nullo se non esiste.
- ***Create_node(v, T1,T2…Tk)***. Crea un nodo r con la label v (il dato che memorizzo) e k figli, radici dei sotto-alberi T1,T2..Tk. Ritorna l'albero avente radice r.
- ***Root(T)***: torna la radice, NULL se l'albero è nullo.
- ***Makenull(T)***: trasforma l'albero T in un albero nullo.
- ***Deleting a node(B)***: prima di cancellare B, devo fare in modo che i figli di B diventino figli del padre di B e poi cancello B e i suoi archi (entranti e uscenti). Così facendo mantengo in qualche modo un ordine anche se alcuni nodi salgono di livello.
- ***Deleting the root***: prendo uno dei suoi figli che diventerà la nuova radice, e gli altri figli di A diventeranno figli della nuova radice.

Esempio: come vengono gestite le directory e le sotto-directory da un SO? C'è una struttura gerarchica e naturalmente questa viene gestita con un albero. Anche la ricerca avanzata nei SO viene realizzata gestendo bene l'albero. Quindi tutte queste operazioni, sebbene sembrino noiose, in realtà sono importanti, sono i nostri mattoncini per poi fare qualcosa d'interessante sugli alberi.

Rappresentazione degli alberi

Come rappresenterei un albero? A parte le funzioni che operano sull'albero, come lo implemento? Se devo sempre partire dalla radice e cercare le foglie, forse non m'importa sapere chi è il padre del nodo, se invece devo andare su e giù potrebbe essere comodo trovare facilmente il padre. Se invece davo partire dalle foglie e devo andare al padre mi basta sapere chi è il padre (come nel torneo).

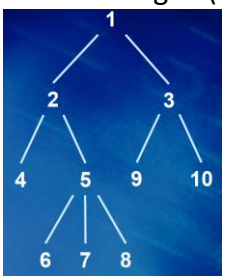
Posso farlo tramite:

- Vettore
- Liste dei figli
- Liste multiple

Non esiste un'unica soluzione, dipende da cosa ci devo fare.

Rappresentazione tramite vettore

Se il numero di nodi è noto a priori conviene usare un vettore in cui metto dentro quello che voglio (magari anche il puntatore al padre).



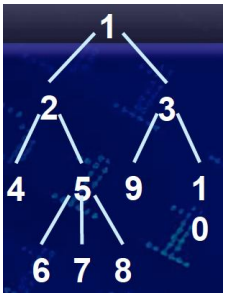
Fonte: testgroup.polito.it

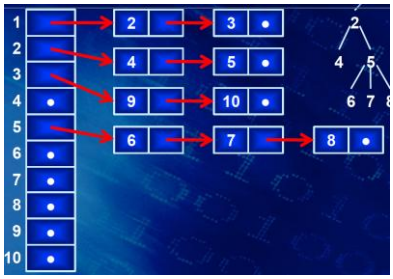
1	2	3	4	5	6	7	8	9	10
0	1	1	2	2	5	5	5	3	3

In questo caso, che complessità ha l'algoritmo per trovare il padre? 1 perché se devo trovare il padre, il padre si troverà in vett[j]. Devo solo fare un accesso. Dato un nodo j trovare i figli; che complessità ha? Devo leggere tutto l'albero, è lineare O(n) (se so che l'albero è binario posso ottimizzare qualcosa). Quindi l'array va bene per trovare il padre (costo costante), per trovare fratelli e figli il costo è lineare(O(n)).

In generale, data una struttura, le operazioni avranno costi diversi.

Rappresentazione tramite liste dei figli





Non rappresento il puntatore al padre ma alla lista dei figli. Dato un nodo, trovare i figli costa 1+G con G=grado dell'albero (ovvero il numero massimo di figli). Trovare il padre invece costa O(nG) perché devo scandire tutti i figli di tutte le liste finché non trovo il puntatore al nodo richiesto. Se invece devo trovare la radice (l'unico elemento senza padre) dovrò scandire tutte le liste dei figli, l'unico elemento che non è presente corrisponde il padre.

Rappresentazione tramite liste multiple

Versione extra-lusso, va bene devo conoscere velocemente sia figli, fratelli e padre.
Costa di più in termini di area occupata, però tutte le ricerche hanno lo stesso costo.

```
Nodo {
padre
  dati nodo
  fratelli
figli
}
```

Warning: di volta in volta, se voglio ottimizzare, devo trovare la struttura dati che ottimizza le operazioni che richiamerò più frequentemente.

Alberi binari

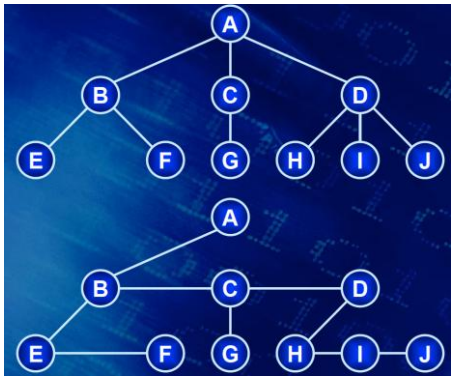
Alberi di grado 2, ciascun nodo può avere:
- nessun figlio
- figlio sinistro
- 1 figlio destro
- 1 figlio destro e 1 sinistro.

La distinzione tra sinistra e destra per i figli di un nodo la posso applicare anche su nodi che non sono legati da una relazione ancestor-descendant.

Regola: se a e b sono fratelli e a è alla sinistra di b, allora tutti i discendenti di a sono alla sinistra di tutti i discendenti di b. Per esempio se alla sinistra ho i nodi contenenti un valore minore e alla destra quelli con un valore maggiore del padre, per fare ricerche ordinate può essere comodo avere questa informazione. Nei BST (binary search tree) infatti si sfruttano proprio queste proprietà.

Regola pratica: dato un nodo n, per trovare i nodi alla sua sinistra e alla sua destra basta percorrere il cammino che va da n alla radice; tutti i nodi che si dipartono dalla sinistra (e i loro eventuali discendenti) si trovano alla sinistra di n. Analogamente per la destra.

Trasformazione binario-normale



Dato un albero normale, lo posso trasformare a un albero binario collegando i figli di quel nodo in cascata e cancellando tutti i rami dal nodo in esami ai suoi figli, eccetto quello con il figlio più a sinistra.

Così facendo cambia la semantica dell'albero, mi perdo delle relazioni gerarchiche. Può comunque essere utile perché se non m'intessa l'ordine gerarchico ma il concetto di sinistra e destra (ad esempio per fare ricerche veloci), allora ha senso perdere in parte la struttura.

Se devo passare da binario a normale-> allora i figli sinistri rappresentano i figli e i figli destri rappresentano i fratelli. (In questo modo riesco anche a non perdermi la struttura di un albero normale trasformato in binario).

Il massimo numero di nodi aventi profondità i è 2^i e quindi per raggiungere la foglia desiderata ho una relazione logaritmica e non lineare. Un albero binario che ha tutti i nodi è detto completo.

Un albero binario è **bilanciato** se per ogni nodo, le **altezze** del sottoalbero di sinistra e del sottoalbero di destra non differiscono per più di un'unità (non è detto che siano perfettamente uguali, un livello in più o meno da una parte non influisce troppo le cose).

Un albero binario si dice **perfettamente bilanciato** se per il sottoalbero di sinistra e per quello di destra, il **numero dei nodi** non differisce per più di un'unità.

Possibili rappresentazioni di alberi binari

Sono casi particolari di quelli visti prima, però adesso conosco a priori il numero di figli.

- **Rappresentazione tramite vettore**: se ho degli alberi binari completi e conosco a priori il numero di elementi, allora userò un vettore di N celle memorizzando

l'i-esimo nodo in posizione i. Se la ricerca del padre risulta frequente, conviene aggiungere a ciascun record un campo aggiuntivo contenete l'indirizzo del padre di quel nodo.
Proprietà:

- La radice ha indice 1.
- se un nodo ha indice *k*, il figlio di sinistra ha indice *2k*e quello di destra *2k + 1.*
- *I*l nodo *k*-esimo si trova al livello log2*k*.
- il primo nodo al livello *i* è quello di indice *2i.*
- il nodo *k*-esimo è il (*k –2*log2k)-esimo sul suo livello.

Operazioni:
•parent(i,T):

–se i≠1 ritorna i/2.
–altrimenti iè la radice;
•left_child(i,T):
–se 2i≤ N ritorna 2i.
–altrimenti inon ha figli sinistri;
•right_child(i,T):
–se 2i + 1≤ Nritorna 2i + 1
–altrimenti i non ha figli destri.

- **Liste multiple**.

Visita di un albero binario

Ci sono dei criteri/modi per visitare l'albero.

Def. Visita *(o spanning) o attraversamento (traversal) di un albero binario, intendiamo l'esame di tutti i nodi dell'albero, in un certo ordine.*

A seconda delle specifiche esigenze, sono definiti 3 tipi di visita (da sapere):

- Visita in **ordine anticipato**(preordertraversal): prima i nodi poi i figli. Prima il padre e poi i discendenti.

```
Preorder-Tree-Walk(x)
If x!=NIL
      then
            printkey[x]
            Preorder-Tree-Walk(left[x])
            Preorder-Tree-Walk(right[x])
```

In modo pratico, prima disegno una curva che tocchi tutti i nodi in senso antiorario. Visito un nodo quando lo tocco con la curva per la prima volta. In questo caso, se ho un albero che rappresenta un'espressione aritmetica e ne faccio una visita inpre-order, allora ottengo un' espressione prefissa (prima le operazioni e poi i dati).

- **Visita inorder (o misto)**: prima il figlio di sinistra, poi il padre e poi il figlio di sinistra. Con la regola pratica prendo il nodo quando lo becco la seconda volta. In questo caso ottengo la notazione infissa.
-
- **Visita postorder**: prima il figlio di sinistra, poi il figlio di destra e poi il padre. Regoletta: segno il nodo l'ultima volta che lo visito. Così facendo, applicando questo tipo di visita a un'espressione aritmetica ottengo la RPN (notazione polacca inversa), notazione postfissa (è una macchina a stack).

Priority Queues

La coda prioritaria come ADT

Def. CODA PRIORITARIA: un tipo di dato astratto utilizzato per gestire un insieme di elementi dove a ciascun elemento è associata una chiave (priorità).

Nella coda normale non c'è un ordine all'interno, solo chi prima arriva prima viene servito; qui invece ci sono elementi con un peso maggiore e che quindi andranno serviti prima. Il problema come sempre è capire che un determinato problema si può modellizzare con una coda prioritaria.

Ha senso utilizzarla quando faccio queste operazioni:

- ***Inserimento di un elemento.***
- ***Ricerca ed eventuale eliminazione dell'elemento con priorità massima/minima.***

In pratica quindi viene realizzata una struttura del tipo:

- HPFO: Highest Priority First Out.
- LPFO: Lowest Priority Fisrt Out.

Operazioni sulle priority queues

- insert(x,P) : inserisco l'elemento x nella coda P.
- find_min(P): torna l'elemento nella coda P avente priorità minima.
- Delete_min(P): cancella l'elemento nella coda P avente priorità minima.

Implementazione

In genere sono implementate tramite liste o vettori (va bene se ho pochi elementi da trattare, qualche decina <30). Se le dimensioni crescono possiamo utilizzare gli heap (particolari alberi), le code unificabili o i tornei.

Binary Search Tree (BST)

Introduzione

Gli alberi binari di ricerca sono alberi in cui in ciascun nodo è contenuto un elemento dell'insieme. *In particolare, per ogni nodo, tutti gli elementi che memorizzo nel sottoalbero di sinistra hanno chiave minore della chiave dal nodo in cui sono partita, mentre tutti quelli di destra hanno chiave maggiore.*

Di conseguenza con una visita in order ottengo tutti gli elementi in modo crescente (perché prima visito il figlio più sulla destra, poi il padre e poi quello sulla destra. Facendo ricorsione e sapendo che l'albero è binario ho ottenuto quello che volevo).

Dato un insieme di elementi, esistono più di un possibile BST (Binaryserachtree), **non c'è una soluzione unica.**

Ricerca

Dato un insieme di elementi e un elemento a parte, come faccio a capire se questo elemento è contenuto nell'insieme?

Prima lo confronto con la radice, se è uguale ho finito; altrimenti in modo ricorsivo scelgo i sottoalberi di sinistra o di destra. Al massimo faccio tanti confronti quanto l'altezza dell'albero (nel caso in cui l'elemento da ricercare si trovi foglia più profonda). Ma poiché non c'è un modo unico per fare l'albero, se ho un albero bilanciato l'altezza è in genere più bassa. Per cui cercherò di lavorare con alberi il più possibile bilanciati.

Inserimento

Supponiamo di non voler aggiungere dei duplicati, faccio una ricerca e mi sposto a sinistra o a destra in base ai valori della chiave che trovo sul cammino (ogni nodo lo confronto con la chiave, se c'è già mi fermo, altrimenti mi sposto a sinistra o a destra). La posizione in cui devo inserire il nuovo elemento è l'ultima che ho visitato.

Aggiungendo degli elementi potrei sbilanciare l'albero, potrei arrivare ad avere una betulla e non un cespuglio, ma io vorrei un cespuglio perché così riduco l'altezza e quindi faccio meno confronti quando devo fare delle ricerche. Se non presto attenzione, i tempi di ricerca potrebbero dipendere dall'ordine d'inserimento degli elementi. Il trucco è prima inserire e poi bilanciare (ovviamente il bilanciare costa). Se bilancio pago qualcosa ma ci guadagno in termini di ricerca, quindi bilanciamento sì o no? Dipende dall'applicazione. Sapendo quanto mi costa la ricerca, quante ricerche faccio e quanto mi costa bilanciare decido cosa scegliere.

Quindi nei due estremi ho un albero completamento sbilanciato -> è una lista lineare e la ricerca costa O(n); se invece l'albero è bilanciato ho un andamento logaritmico. Per numeri molto grandi il logaritmo è più basso (meglio albero bilanciato), se ho pochi elementi bisogna fare due conti e vedere se il gioco vale la candela.

Quanto può costare il bilanciamento di un albero?

Posso dover spostare molti elementi, e quindi costa. In genere conviene bilanciare se il numero di inserimenti è molto minore della frequenza di ricerca, se invece faccio molti inserimenti e cancellazioni conviene non bilanciarlo ogni volta.

Si può comunque dimostrare che grandi valori di n, il costo medio di ricerca in un albero in cui non viene effettuato un ri-bilanciamento dopo ogni aggiunta di un nuovo elemento è un $O(1.4\log_2 n)$.

Cancellazione

- Il nodo non ha figli: lo cancello e basta, no problem.
- Ha solo un figlio: sostituisco l'elemento da cancellare con il figlio. Ho ancora un BST.
- Ho due figli: devo sostituirlo con qualcosa che mantenga la gerarchia. Lo sostituisco con i numeri a lui più vicini, il più grosso dei più piccoli di lui o il più piccolo dei grandi di lui.

Anche la cancellazione può creare problemi di sbilanciamento.

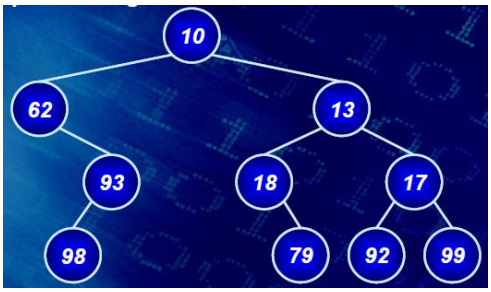
Altri tipi di alberi

Cancellare o inserire un elemento può richiedere lo spostamento di molti elementi, può essere un casino. Se guardiamo alle applicazioni, (per esempio la gestione di un file systems), non è così ovvio che il numero di inserimenti/cancellazioni siano molto inferiori alle letture sul file system, non posso dirlo a priori. Un normale BST dove ogni volta bisogna bilanciarlo può richiedere un costo elevato, per ovviare al problem ci siamo inventati altri tipi di alberi di ricerca dove il bilanciamento costa poco.

- **M-way searchtree**: alberi di ricerca non più binari (ogni nodo può avere 3-4 figli).
- **B-tree**: come m-way, viene usato dai SO, faccio le modifiche solo sul cammino dalla radice al nodo da modificare non a tutto l'albero (modifiche solo locali). Mi costa un po' di più l'infrastruttura ma ottengo dei vantaggi nel costo bilanciamento.

Heap

Def. Heap*(mucchio): è un albero binario in cui ogni nodo contiene una chiave di valore minore o uguale a quelle dei figli.*



Quindi non più vero che le cose che ho a sinistra sono più piccole.

La radice ha la chiave di valore minimo, del resto l'heap l'avevamo visto come un modo per implementare le code prioritarie. Inoltre, quando cambio il valore di un nodo, posso ripristinare la struttura a heap facendo al massimo un numero di scambi pari all'altezza dell'albero.

Inserisco un elemento e poi controllo che il padre sia minore del figlio. E vado avanti finché questa condizione non è verificata (quindi parto dalla foglia e torno verso la radice, il worst case è che ripercorro tutta l'altezza dell'albero).Se invece voglio inserire un nodo, lo aggiungo come foglia e poi lo faccio muovere fino alla posizione corretta (faccio solo confronti con il padre, vedo se è > o <).

Ricordando che uso gli heap per la gestione delle code prioritarie, la cancellazione (ovvero il servire un elemento), vuol dire fornire quello con chiave minore (la radice quindi) e poi sistemare la struttura in modo da riavere un heap.

Anche nel caso dell'heap, se è bilanciato le complessità delle operazioni sono logaritmiche.

Finché facciamo giocattoli va quasi bene tutto, sul piccolo posso anche avere soluzioni esponenziali e lineari; ma se devo cercare soluzione che in un secondo tempo possa scalare in complessità devo pensarci prima di mettermi a scrivermi.

Implementazione: spesso viene fatto utilizzando il vettore (se so apriori quanti elementi ho), numero l'albero e poi inserisco l'elemento i-esimo in posizione i-esima.

Graphs:introduction & definitions

Negli alberi il cammino che porta dalla radice a un nodo qualsiasi è univoco. Se vogliamo è una limitazione, io vorrei poter avere più padri o dei figli condivisi. Così facendo ottengo un grafo.

Introduzione

Storicamente, la teoria dei grafi deriva dalla teoria dei giochi e dalla matematica ricreativa. Il grafo rappresenta relazioni fra oggetti, gli oggetti sono i nodi e le relazioni sono rappresentate tramite gli archi. Una volta che abbiamo mappato il nostro problema come un grafo, ci estraiamo dalla problema originario e lo trattiamo come grafo, il risultato che ottengo la rimappo e torno al problema originario.

***Def. formale**: si definisce grafo una copia ordinata di insiemi (V,E) dove V è un insieme finito e non vuoto di oggetti, denominati nodi o vertici, E è un insieme di archi o spigoli.*

Arco: coppia di nodi che risultano collegati dall'arco stesso.
|V| = cardinalità degli elementi (numero elementi); |E|= numero di archi.
Nodi-> cerchi; archi-> segmenti. Il segmento può essere o meno orientato (saranno casistiche diverse). In progettazione abbiamo visto tutti grafi orientati. Non è necessario associare a ogni arco un valore, se invece metto un'etichetta allora il grafo diventa "pesato".
Quali possono essere i problemi su di un grafo? Il grafo è connesso o no (ogni elementi sono raggiunti da almeno un arco)? Ci sono dei punti critici (se tolgo un arco va tutto a pallini)?

Definizioni varie

Quando tutti gli archi non sono orientati allora il grafo è detto **non orientato** (o non diretto). Ogni arco 'e' in un grafo non orientato può essere espresso come una coppia non orientata di vertici $\{v_1, v_2\}$. Quindi il verso di percorrenza è lo stesso {v1,v2} = {v2,v1}. In questo caso v1 e v2 sono adiacenti e l'arco è incidente sui vertici v1 e v2. Se i due vertici sono coincidenti allora ho un self-loop.

Grafi orientati: gli archi hanno un verso, il grafo è detto orientato o diretto. In questo caso {v1,v2}!={v2,v1}.V1=coda; v2=testa.
v1 è predecessore di v2; v2è un successore di v1.
Il grafo non può contenere archi multipli (diventerebbe un multi grafo), in un grafo orientato va bene avere due archi fra due nodi per indicare sensi di percorrenza opposti, non va bene averne 3.

Un grafo è **semplice** se non contiene cappi (no self-loop).

Se il mio grafo ha tutti i possibili archi che posso disegnare, allora il grafo è detto **completo**. Nel caso di un grafo completo la notazione è $K_{\#}$ con #=numero nodi. Questo serve perché un problema tipico è: dato un grafo, trovare tutte i suoi sottografi completi. Per esempio nei nostri diagrammi a stati (STG), quando dovevo trovare degli stati equivalenti, quello che posso pensare è di fare un secondo grafo in cui ricopio gli stati e provo a ipotizzare delle equivalenze fra gli stati, allora se avessi un algoritmo che trova i sottografi completi, allora quei sottografi li posso fondere in un unico stato e così ottimizzo qualcosa.

Se $K_{|v|}$ è un grafo completo con v vertici, allora avrò:

- |v|(|v|-1) spigoli se il grafo è orientato.
- |v|(|v|-1)/2 spigoli se il grafo è non-orientato.

In un grafo qualunque, il **grado o valenza** di un nodo v il numero ρ(v) di archi incidenti il nodo stesso. Nel grafo orientato in particolare si distingue fra grado in ingresso e grado in uscita (negli STG dovevamo stare attenti che l'out-degree fosse uguale al 2^nro. ingressi).

Teorema: La somma dei gradi dei vertici di un grafo, orientato o no, è un numero pari, a condizione che si assuma che un cappio dia un contributo pari a 2 nella valutazione del grado del vertice cui è incidente. Tale somma vale 2|E|.
Corollario: in un qualsiasi grafo, esiste un numero pari di vertici aventi ordine dispari.

Grafi regolari: grafo in cui tutti i nodi abbiano lo stesso grado r.
Esempio: ho una mappa geografica con i confini e voglio colorare gli stati in modo che stati adiacenti abbiano colori diversi. Qual è il numero minimo di colori da utilizzare? In teoria dei grafi traduco la mappa in un grafo dove i nodi rappresentano le regioni, e collego due nodi se sono adiacenti (se hanno un pezzo di frontiera in comune). A questo punto cerco in letteratura l'algoritmo e mi cimento.

Grafi pesati: Un grafo pesato è una tripla ordinata *G = (V, E, P)* tale che:

- *G' = (V, E)* è un grafo
- *P* è una funzione che associa ad ogni arco (v_i, v_j) o $\{v_i, v_j\}$ di *G'* un peso p_i. Il modo usuale di rappresentare i grafi pesati consiste nell'indicare, in vicinanza di ogni arco, il peso corrispondente.

Sottografo: Dato un grafo *G = (V, E)* si dice sottografo di *G* un grafo *G'* tale che, se *G'= (V', E')*, si abbia:

- $V' \subseteq V$
- $E' \subseteq E$

Grafo bipartito:un grafo *G* si dice bipartito se è possibile suddividere l'insieme *V* dei suoi nodi in due sottoinsiemi disgiunti V_1 e V_2 in modo tale che ogni spigolo di *G*

congiunga un nodo di $V1$ a un nodo di $V2$.

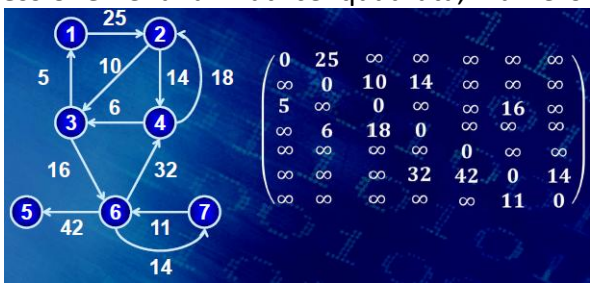
Un grafo non orientato bipartito in cui ogni nodo di $V1$ sia congiunto a ogni nodo di $V2$, viene definito **bipartito non orientato completo** ed indicato con $K_{m,n}$ essendo m ed n rispettivamente il numero di nodi di $V1$ e $V2$, $K_{m,n}$ ha $m + n$ vertici ed $m\,n$ spigoli.

Un grafo orientato bipartito in cui ogni nodo di $V1$ sia congiunto a ogni nodo di $V2$ e ogni nodo di $V2$ sia congiunto a ogni nodo di $V1$, viene definito **bipartito orientato completo** e indicato con $K_{m,n}$ essendo m ed n rispettivamente il numero di nodi di $V1$ e $V2$. ,$K_{m,n}$ ha $m + n$ vertici ed $2mn$ spigoli.

Rappresentazione di un grafo

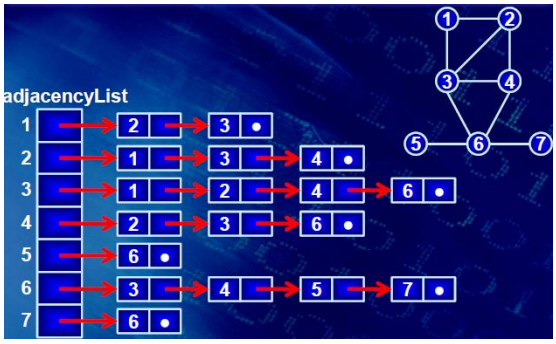
La rappresentazione dei grafi avviene tramite:

- **Matrice di adiacenza o di connessione**: è una matrice quadrata, numero righe=n.colonne=n. nodi (se vogliamo è una Mappa di Karnaugh). Ho un 1 se il nodo di quella colonna è legato al nodo indicato sulla riga. Se il grafo non è orientato la matrice è simmetrica.



  Vantaggi e svantaggi: se il grafo è completo conviene questa rappresentazione, non conviene se ho pochi archi (matrice sparsa). Se un nodo è scollegato il determinante della matrice è 0.
  Nel caso di grafo pesato riporto il valore del peso nella matrice. Ma se il peso vale 0, come faccio a distinguere fra arco con peso 0 e assenza di arco? se c'è un valore che non è assunto dai pesi sono a cavallo, altrimenti non posso far miracoli e quindi dovrò usare due strutture/matrici diverse (una per i pesi e una per gli archi).

- **Liste di adiacenza**: ho un primo vettore con size=numero di nodi, per ogni nodo la lista degli elementi a cui è adiacente. Se il grafo non è connesso, gli elementi isolati punteranno a una lista vuota. In questo caso non ho più il problema delle matrici sparse. Se il grafo è pesato aggiungo nella lista il peso. Se il grafo cambia



dinamicamente parecchio, a sua volta il vettore iniziale può essere implementato come lista.

- **Matrice d'incidenza**. Non si usano quasi mai.

An introduction to object-oriented programming (part 2)

Paradigmi fondamentali della OOP: incapsulamento, classi e istanze, ereditarietà e polimorfismo.

Inheritance

In progetti abbastanza complessi dovrei dover trattare con tante classi, e magari ci sono due o più classi che fanno/hanno tanto in comune, sono diverse perché fanno responsabilità diverse ma con property e metodi molto simili. Per non duplicare il 75% del codice forse ho dei metodi più efficienti per trattare questo tipo di situazioni.

Esempio: prendiamo 3 biciclette (una da città, una mountain bike e un tandem), sono diverse, hanno una parte in comune (ruote, manubrio, etc.) ma ciascuna introduce un particolare differente (il tandem ha 2 seggiolini, quella da città ha il cestino, la mountain ha dei cambi particolari).

Ogni volta che pensiamo alle classi, di fatto inconsciamente creiamo delle gerarchie, le tre biciclette di prima sono tutte biciclette specializzate, ognuna ha delle peculiarità in più. Ovvero possiamo identificare una classe base (bicicletta), molto comune, quella che contiene il 75% di behavior/features e delle classi derivate che ereditano le proprietà/metodi delle classe base e poi aggiungono qualche property/metodo specifico per i loro scopi. In questo modo evitiamo uno spaghetti-code. Scrivo il codice per le funzioni base una sola volta e poi mi concentro sulle loro specificità.

Non è banale e semplice capire quando posso utilizzare l'ereditarietà quando non è proprio esplicito. Per capire se una classe è derivabile posso usare la regola dell' **"is a"**? Una mountain bike "is a" bicicletta? Sì, allora la posso derivare. Il gatto "is a" mammifero? La OOP è molto simile al nostro modo di ragionare. A questo punto potrei avere anche più livelli di gerarchie (es. animale->mammifero->gatto). In questo modo cerco di riutilizzare il più possibile quello che ho già scritto.

Quindi possiamo usare l'ereditarietà se è verificata la relazione: "is a".

La OOP permette che la classe derivata erediti dalla classe base funzioni e proprietà. Oltre a ereditare tutto quello che ho da tutte le classi sopra, posso anche definire caratteristiche addizionali della mia classe derivata. Inoltre non solo eredito i metodi, ma posso anche decidere di ridefinire il metodo (**override**). Il prototipo è lo stesso

(stessi parametri in ingressi e in uscita, stesso nome) ma il corpo lo riscrivo ex-novo. Mi può servire perché magari ho il metodo nella classe base, però magari è molto generico; nella classe derivata però mi potrebbe servire un metodo più dettagliato o scritto in modo più efficiente->override.

Se faccio una classe derivata, faccio l'override di un metodo, e poi questa classe derivata diventa la classe base per una seconda classe derivata, la seconda classe derivata quale metodo eredita? In genere eredita quella del suo padre (salgo solo su di uno step) a meno che non lo specifichi espressamente (e in tal caso eredito il metodo del "nonno").

Quindi l'ereditarietà permette di organizzare informazioni, raggruppare classi simili e creare tassonomie, e modellare le similitudini della classe (eg. Override).

Con l'ereditarietà estendo ulteriormente il concetto di riusabilità. Posso prendere una gerarchia di classi già esistente e ampliarla aggiungendo qualche dettaglio per poi poterla rilasciare.

Esempio: il plug-in è un pezzo di codice che aggiunge funzionalità a un programma già esistente, questi vengono sviluppati con quest'ottica (non riscrivo tutto il programma ma aggiungo solo quello che mi serve).

Polymorphism

Nella OOP per definizione è **la caratteristica di un qualcosa che può mutare la propria forma** (in genere il tipo) **a runtime**. Più forme significano avere risposte diverse allo stesso messaggio. Cambiando forma, cambia il comportamento quando viene sollecitato. Il polimorfismo fa mutare il comportamento di una funzione o di un oggetto. Il polimorfismo non esiste in C. Esso si basa su tre cose fondamentali:

- **Ereditarietà**.
- **Metodo virtuale**: una funzione membro che posso ridefinire se considero altre classi derivate.
- **Metodo overloading**: metodo con stesso nome ma con diverso corpo, parametri in ingresso e in uscita (gioco con il prototipo, supponendo sempre che ci sia ereditarietà).

Overiding: come overloading ma posso solo cambiare il corpo (il prototipo deve essere uguale a quello della classe che eredito).

Supponiamo di avere una gerarchia di classi: forma dalla quale discendono cerchio e quadrato. Posso disegnare una qualsiasi forma, ma il modo in cui la disegno cambia a seconda che sia un cerchio (serve un raggio e un centro) o un quadrato (devo conoscere la lunghezza del lato). Supponiamo che il metodo `void draw()` dello shape faccia qualcosa e di fare l'overriding per il cerchio e per il quadrato (non cambio

il prototipo). Nel main definisco 3 oggetti shape, circle e square e li passo una stessa funzione esterna. Posso farlo? Sono tutte e 3 shape, e quindi posso passarli tutte e tre a una funzione che vuole in ingresso uno shape.
Quindi il polimorfismo permette di passare questi oggetti alla funzione, ma quando in questa funzione richiamo sull'oggetto la funzione membro draw(); allora grazie al polimorfismo richiamerò su shape il draw() originario; mentre su circle e square le draw() overiddate. *Polimorfismo perché la funzione draw() ha forme diverse a seconda dell'oggetto sulla quale la richiamo*.

Come fa la funzione esterna a sapere che deve prendere il draw() overridato? A runtime ogni oggetto occupa un'area di memoria dove sono contenuti sia i metodi sia le property. Quando passo cerchio, lui va all'inizio dell'area di memoria del cerchio e di li cerca la funzione draw(). Il polimorfismo è anche detto **casting dinamico**. Durante il tempo di compilazione non me ne accorgo, controllo solo che dato sia di tipo shape

Memory devices

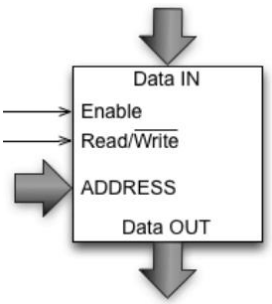
Vogliamo caratterizzare le memorie come singoli dispositivi e non all'interno di un sistema più complesso.

Memory organization

Le memorie sono composte da celle (elemento atomico della memoria), **ogni cella contiene un bit d'informazione**. In alcuni tipi di memorie le celle possono contenere più di un bit.

Le celle sono organizzate in **word** (dal punto di vista dell'utente è la più piccola porzione di memoria che si può leggere o scrivere, non si accede alla singola cella ma alla word).
Nota: In alcuni processori, quando nella loro documentazione si parla di instruction set, si ragiona in termini di word, half word, etc. ma quello è un contesto diverso e non c'entra nulla con le word delle memorie.

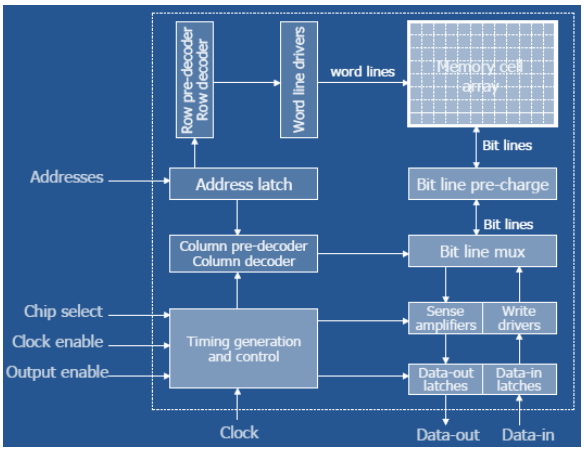
Ciascuna word è identificata in **modo univoco** attraverso un **address**. La corrispondenza fra posizione della word e il suo address è decisa in fase progettuale. Quindi quando accedo alla memoria devo fornire l'address sul quale voglio operare.



Quindi per usare una memoria avrò concettualmente:

- data in e data out.
- address.
- enable.
- segnale di controllo read/ write.

La memoria è un dispositivo sequenziale, ma qui manca sia il clock sia il reset, perché? Il clock non è indispensabile perché le memorie possono essere asincrone (no clock) o sincrone (si clock). Il reset invece non c'è mai, **le memorie sono gli unici dispositivi sequenziali che non hanno il reset**; non posso inizializzare il contenuto della memoria con un solo "filo" ma devo per forza scriverla tutta quanta, word per word.



In generale ci sono anche altri segnali di controllo.
D'ora in poi parleremo di B=byte, b=bit.

Questo è uno prima schema a blocchi di come è fatta una memoria. C'è una matrice di celle di memoria (il vero kernel), poi c'è la logica d'interfaccia verso l'esterno (amplificatori e cose varie), un registro per memorizzare l'address che utilizzerò per accedere alla matrice. In più ho una control unit che permette di gestire il tutto.

Memory characteristic parameter

- **Access time** (or delay or latency): tempo che s'impiega a compiere un'operazione di lettura o scrittura. È il tempo richiesto per leggere una word.
- **Cycle time**: intervallo di tempo minimo fra due accessi consecutivi alla memoria.
- **Capacity**: la quantità di b o B che può memorizzare.
- **Throughput**: la quantità di dati che la memoria può trasferire nell'unità di tempo (concetto usato non solo nelle memorie ma anche in network e simili).
- **Costs**: costo per Byte/bit.
- **Density**: rapporto fra capacity e dimensione fisica della memoria.
- **RBER (Raw Bit Error Rate)**: purtroppo non è 0, nella memoria ho scritto un certo dato, quando lo leggo non leggo quel dato ma quel dato con degli errori. La probabilità di questo errore può essere anche elevata, per cui quando scrivo non scrivo solo il dato ma anche un codice (ECC: Error Correcting Code) per recuperare eventuali errori, non leggo il singolo dato ma il dato + codice, qualcuno andrà a interpretare questa stringa e tirerà fuori il dato corretto. Ma allora perché non uso memorie più affidabili? Perché costerebbero troppo. RBER: qual è la percentuale/probabilità di leggere un dato sbagliato all'uscita senza applicare la ECC. Dopo l'ECC l'RBER dovrebbe andare a zero. È necessario conoscere l'RBER per dimensionare correttamente l'ECC.
- **UBER** (Uncorrectable Raw bit error date): la percentuale di errore che ho dopo l'ECC (dovrebbe essere zero). Questi meccanismi sono tutti a livello hardware, non c'entrano eventuali correzioni software.

Taxonomy

- **Packaging** (cambierà la tipologia del testing)
  - Chip
  - IP core
- **Access order**:
  - Sequential: non posso usare un ordine qualsiasi per accedere all'i-esima cella (es. nastri o FIFO).
  - Random: posso accedere in modo casuale alle word, specifico ogni volta l'indirizzo sul quale voglio operare.
  - Burst: nel caso di RAM dinamiche, indirizzo una word ma di fatto accedo all'i-esima word più quelle a lei vicine (se indirizzo la 67, leggo 67, 68, 69, 70, sono quelle che rispondono a degli indirizzi successivi). Nel caso delle memorie random avrei dovuto fare n-accessi.
- **Operations:**
  - Read only
  - Read-write
- **Data retention capability**:
  - Memorie volatile (SRAM e DRAM), perdono il loro contenuto quando manca l'alimentazione. Quando la ri-alimento il loro contenuto è casuale.
  - Non-volatili: NVM, non perdono il loro contenuto quando manca l'alimentazione.
- **Data access**:
  - By address: specifico l'indirizzo e su quello lavoro.
  - By content: CAM, non accedo per indirizzo ma per contenuto, alla memoria chiedo: "al tuo interno c'è una word che contiene 010001?": se sì mi torna l'indirizzo della word che lo contiene, altrimenti torna no. Serve a chiedere se ho già un dato.
  - By writing order: in particolare di FIFO, non parlo di LIFO perché sarebbe uno shift register.
- **Access timing**:
  - Sincrona (questa ha almeno un clock).
  - Asincrona (userà il segnale di controllo per funzionare).
- **Numero di porte di accesso:**
  - Single: un array di word e un solo utente.
  - Multiple port (ho una sola matrice di array ma ho più utenti esterni che possono accedere al contenuto). In questo caso è tutto sincrono.

External vs internal Architectures

***Architettura esterna**: il modo in cui l'utente vede il componente e lo usa.*
***Architettura interna**: il modo in cui le celle sono fisicamente organizzate all'interno del componente.*

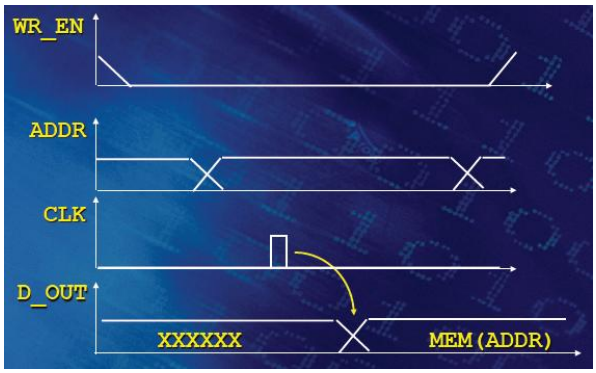
Se dall'esterno vedo la memoria come 1024 parole da 8 bit ciascuna, non è detto che l'array delle celle (quello fisico) sia un rettangolo di questo tipo.

Esempio: micron MT48LC32M16A2
(http://pdf1.alldatasheet.com/datasheet-pdf/view/226008/MICRON/MT48LC32M16A2.html)
Architettura esterna: 32 M words da 16 bits ciascuna, quindi avrò un address bus da 15 bits (32M=2^15), e un data bus di 16.
Architettura interna: è organizzata da 4 banchi, ogni banco è composto da 8k righe e 1k colonne, dove ogni colonna è formata da 16 bit.

Che impatto ha questo? La cosa più significativa è che dal punto di vista utente non m'interessa molto com'è fatta dentro, ma se mi devo porre il problema di testare la memoria (capire quali word sono distrutte) questo può essere più problematico. Uno dei guasti più tipici è che mentre accedo a una cella, allora disturbo anche le celle a lei adiacenti. Se ci fosse corrispondenza fra architettura esterna e interna no problem, ma fisicamente chi sono le celle adiacenti a quelle rotte? Se sono il costruttore lo so a priori, ma se non so nulla potrebbe essere un bit qualsiasi, se so che invece sono organizzati in banchi almeno so che i bit influenzati sono sullo stesso banco.

Volatile memory

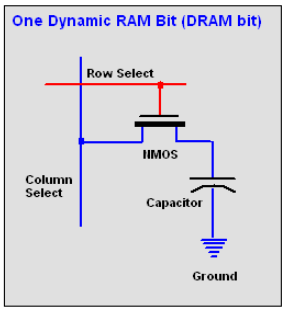
L'informazione è persa quando manca l'alimentazione, tipicamente le più usate sono quelle ad accesso casuale-> RAM. Queste possono essere dinamiche o statiche. Da parecchio le memorie sono di fatto i componenti a tecnologia più spinta, sono i front-runner dell'industria dei semiconduttori. Quando implemento SoC posso avere tante memorie (anche un centinaio). Nell'intel core 2 metà è il vero processore, metà è composta dalla cache di secondo livello -> non si può ignorare in termini di affidabilità.



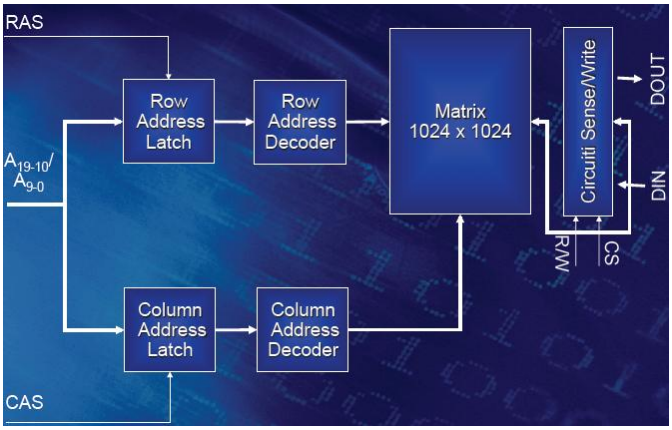
Schema memoria sincrona single, non prendiamo come oro colato questo diagramma temporale, ogni costruttore potrebbe definirsi il suo protocollo, anche se si sta andando verso una standardizzazione. Quando scrivo in uscita ho dei don't care.

In genere le DRAM sono più lente (access time di 20 ns vs 2ns) ma la DRAM è più densa (a pari numero di bit, lo spazio occupato è inferiore).Le DRAM sono più piccole e più lente; le più affidabili sono le SRAM (più grande e più veloce). Nelle SRAM la cella è implementata da un latch, nelle DRAM l'informazione è storata caricando un condensatore. Ma il condensatore si scarica perché non sarà mai collegato a una resistenza infinita , tenderà a perdere l'informazione dopo qualche millisecondo -> sono più piccole ma bisogna avere i giusti accorgimenti. In genere uso le DRAM come memoria principale e le SRAM come cache; le due sono diverse sotto vari aspetti (tecnologia, affidabilità, prestazioni, etc.).

DRAM



Ho una capacità che si carica, una word line e una bit line. Quando abilito la word line, il C è connesso alla bit line-> la bit line la posso usare per leggere o scrivere il condensatore. Ogni cella ha un transistor e un condensatore. Ma il condensatore si scarica: mi serve un meccanismo che faccia un refresh periodico per compensare la perdita di carica. Il rinfresco avviene leggendo e riscrivendo il valore prima che si scarichi. Questa operazione è fatta automaticamente a livello hardware, il software non interviene. Le operazioni di refresh devono avere la massima priorità e devo impiegare/costare poco tempo. Esiste a bordo un contatore per i refresh.



In realtà (e vale per tutte le DRAM), l'indirizzo fornito viene preso e spezzato in due parti -> metà per le colonne e metà per le righe, avrò poi due decoder per puntare alla matrice. In alcuni casi, per diminuire l'address bus size, l'indirizzo non veniva fornito in un unico colpo ma in due colpi, uno per le righe e uno per le colonne. Di qua i termini CAS (Column Address Strobe) e RAS (Row Address Strobe).

I PC users non interagiscono direttamente con le singole DRAM ma con delle board/stick contenenti più DRAM. Sulla scheda ho dei connettori per aggiungere altri slot (memorie estensibili).

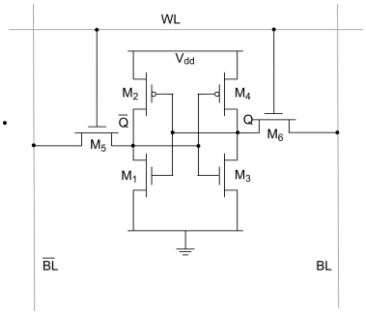
Ecco alcune tipologie di SRAM:

- **FPRAM** (Fast page mode ram): implementano il burst, posso leggere un blocco di word. Page vuol dire insieme di word consecutive.

- **EDO DRAM**: versione più veloce delle DRAM, fa più cose contemporaneamente.
- **SDRAM**: DRAM sincronizzate, uso lo stesso clock che viene mandato dal processore (alzato o abbassato in frequenza). Il controllore della DRAM è una macchina a stati finiti e usa il clock per lavorare. Con l'accesso sincrono ho prestazioni più elevate di quelle asincrone.
- **DDR SDRAM** (double data rate sincr): entrate nel mercato nel 2002, sono memorie sincrone in cui ***entrambi i fronti del clock*** contano (e non solo più uno come avveniva classicamente). Quindi riesco a fare 2 operazioni in un solo periodo di clock. DDR2&DDR3: aumentano le prestazioni. I protocolli di comunicazione per queste sono standard.
- **Hybrid memory cube**: sono DRAM organizzate a stack, ho un packaging organizzato a piani (3D packaging). Questa è una memoria, ma oggi è normale avere due piani sullo stesso chip (nel caso del chip dove ho processore più memoria, ho prestazioni più elevate perché la distanza processore-memoria è minore, impiego meno tempo, consumo di meno e occupo meno spazio).

SRAM

Ogni cella è composta da due flip flop; ho due inverter in cascata che si alimentano a vicenda, si ha sempre che sulla bitline1=!bitline2 (se bitline1=0 allora bitline2=1 e vice versa). Nel momento in cui non ho un segnale sulla word, i transistor non sono attivi, se voglio leggerla devo asserire la word line e quindi posso leggere (leggo bitline1 e bitline 2 e poi la logica mi dice qual è il valore giusto, visto che sono complementati) o scriverle (forzo bitline1 e bitline2 a due valori opposti). Quindi non ho più un rinfresco, la cella è più grossa ma più veloce (e quindi sarà destinata a parlare con la CPU).



Non-volatile memories

In genere sono organizzate in 3 famiglie:

ROM e derivati.

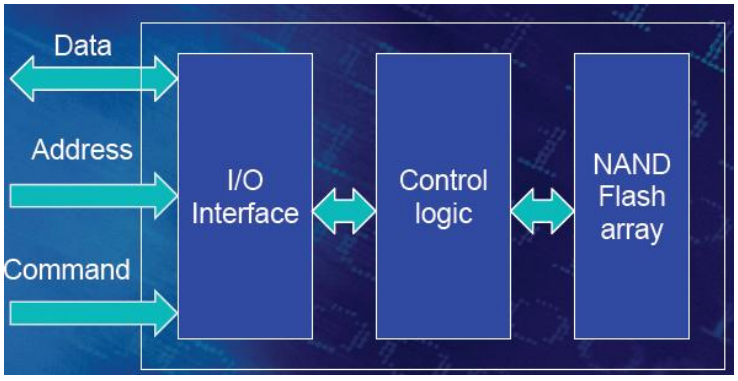
- **Read only memory (ROM):** anche la ROM ha un'accesso random (coma la RAM), ma questa è non volatile. L'informazione è contenuta in una resistenza-transistor, è in fase di costruzione che metto o no la resistenza. Solo in fase di costruzione decido qual è il contenuto da assegnare. Ma sono un po' scomode.
- **EPROM** (Erasable, programmable Read-only memory): può essere programmata sul campo, per resettarla devo esporla a luce ultravioletta per circa 20min.
- **EEPROM**: come EPROM ma viene cancellata con un opportuno campo elettrico.

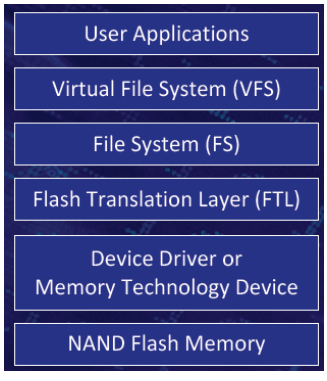
Il trend è quello di usare le EPROM (perché posso creare e togliere il ponticello attraverso luce ultravioletto/campo elettrico). Nel caso della ROM il data retection è infinito, mentre per le EPROM (10 anni) e le flash (10 anni se non le uso)

Criterion	SRAM	DRAM	ROM	EPROM	EEPROM	Flash
Relative cell size	4–6	1.5	1	1.5	3–4	1.5
Volatility	yes	yes	no	no	no	no
Data retention	∞	64 ms	∞	10 years	10 years	10 years
In-system re-programmability	yes	yes	no	no	yes	yes
In-system read speed	2.5 ns	55 ns	90 ns	90 ns	200 ns	60 ns
In-system write speed	2.5 ns	55 ns	—	—	2.5 s	6.4 μs
Number of writes	∞	∞	1	100	10^4–10^5	10^4–10^5

FLASH

A parte le chiavette USB, la grande crescita delle flash deriva dagli hard disk a stato solido (SST). Inventati nel 1984 dalla Toshiba, la tecnologia fu sfruttata solo più tardi da Intel e IBM. Indipendentemente dalla tecnologia costruttiva (NOR o AND), in entrambi i casi la lettura e la scrittura non sono operazioni simmetriche. Prima la flash deve essere azzerata (erase), poi la inizializzo. Quando la voglio scrivere, se uno 0 deve diventare un 1 no problem, se un 1 deve diventare uno 0 non posso scrivere il singolo bit ma devo scrivere tutta la pagina. Tuttavia dopo 10^4 operazioni di erase la flash è in pratica da buttare via (avrei troppi errori di lettura). Un conto è la pennetta USB che uso e probabilmente dopo 100000 letture posso comprarne un'altra più grande allo stesso prezzo, se però uso un SSD per una ditta o per un server e quindi rischierei di buttarlo via dopo qualche giorno. Ci sono una serie di applicazioni per la quale la memoria flash non è adatta (per problemi di endurance, affidabilità etc.). In genere le 10^4 operazioni sono riferite alla cella. Inoltre nei file system ci sono dei meccanismi per livellare l'invecchiamento, cerco di scrivere un po' in tutta la flash e non far morire una piccola zona a forza di scrivere sempre lì. Quando dico che dopo 10^4 operazioni

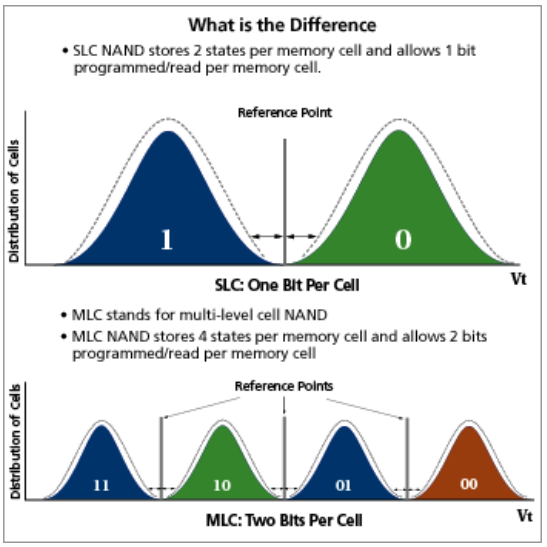




la butto via vuol dire che preso l'andamento della probabilità di guasto a vasca da bagno, allora quando sono a 10^4 scritture allora sono all'inizio della morte di invecchiamento. Per le flash ho bisogno di un file system particolare che gestisca particolari problematiche.

In un SST delle ultime generazioni c'è un controllore che in genere è un ARM 9 con un po' di RAM intorno per bufferizzare; è normale avere dentro la memoria un processore più potente di quello della macchina su cui è montata. Addirittura ci sono già i multi-core a bordo delle flash. Questo tipo di architettura e protocollo di comunicazione sono stati standardizzati dalle stesse aziende costruttrici. A differenza delle RAM, ci sono un bel po' di layer in mezzo tra cui usa la flash la flash vera e propria (sarà necessario del software specifico per poter gestire correttamente la flash).

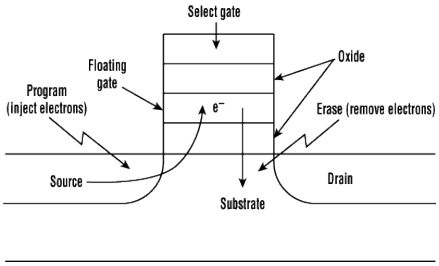
A livello tecnologico possiamo distinguere due tipologie di flash: le NOR e le NAND. A livello macroscopico, le NOR permettono un accesso random alle singole celle, la lettura è veloce ma scrittura e cancellazioni sono lente. Per cui le NOR vengono utilizzare come memorie non volatili per memorizzare il bootstrap. Il bootloader viene caricato nella flash. A differenza delle NOR, le NAND sono molto veloci in lettura e scrittura, alta densità, basso consumo e quindi vengono utilizzate per l'archiviazione di dati.



Con le memoria flash la singola cella è in grado di memorizzare non solo un singolo bit ma più bit d'informazione. Questo è possibile perché una qualsiasi tensione vicino alla tensione di riferimento è assunta come uno, mentre da 0 a un po' di rumore è uno zero logico (SLC, Single Level Cell). Con le flash non uso solo più due bande ma magari 4 -> MLC (Multi level Cell) . In genere adesso nelle flash vengono utilizzate 4 bande-> posso memorizzare 2 bit nella stessa cella. Con questo tipo di soluzione però ho meno spazio fra le bande, ho meno margine (le tensioni di alimentazioni non possono salire troppo) -> aumenta la probabilità di errore. Intrinsecamente la MLC è meno affidabile. Inoltre la MLC si guasta prima (endurance è 10^4) mentre la SLC è un po' più longeva (10^5).

L'altro problema grosso che le flash avranno è che non scaleranno sotto i 14-15 nm. Quando la tecnologia scenderà sotto quella soglia, le flash saranno superflue. Nel caso della FLex-oneNAND, chip prodotto dalla samsung, è l'utente a decidere se usarla come SLC o some MLC; in realtà la tecnologia costruttiva è la stessa, cambiano solo gli algoritmi di scrittura/lettura. Se mi serve affidabilità la uso come SLC (magari perdo di capacità), altrimenti come MLC. Questo è un primo tentativo di vedere la memoria al servizio nell'utente, nel senso che è l'utente a decidere come usarla in base alle sue esigenze.

Le flash NAND sono organizzate in celle, pagine blocchi e piani. L'informazione viene memorizzata in un floating gate dentro una specie di MOS, azzerarla vuol dire portare via tutti gli elettroni , quando voglio scriverla inietto degli elettroni. Degrada perché gli elettroni iniettati non arrivano tutti nel floating gate ma un po' rimangono intrappolati nel substrato. Normalmente accedo alla **pagina** (quando leggo, leggo tutta la pagina; è la più piccola unità che posso leggere). In genere una pagina è 4kByte più 128B di scorta (utilizzate dal SO, in genere c'è un contatore che mi dice quante volte l'ho già scritta). Il **blocco** è la minima quantità che può essere cancellato (128 pagine).



1. This is the structure of a basic Flash memory cell. If a floating gate is added between the substrate and the select gate, the cell can retain a charge that may permanently store data.

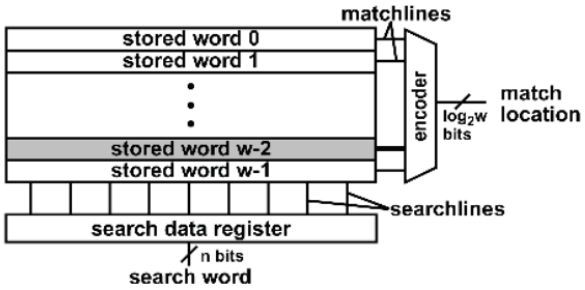
	Features			*Architecture*			*Reliability*		*Array Operations*			
	Bits /cell	**Volt-age**	**Bus width**	**Pla-nes**	**Page size**	**Pages /block**	NOP	**ECC-512B**	**Endu-rance**	**t_{READ} (max)**	**t_{PROG} (avg)**	**t_{ERASE} (avg)**
SLC	1	3.3V, 1.8V	x8, x16	1 or 2	2,112B	64+	1	1	$<10^5$	25us	200 - 300us	1.5 – 2ms
MLC	2+	3.3V	x8	2+	4,314B+	128+	4+	4+	$<10^4$	50us	600 - 900us	2ms

Universal memories

Intendiamo memorie che siano contemporaneamente veloci come le RAM e non-volatili. Facili da accedere, a basso consumo, oggi veloci quanto le DRAM. L'approccio è completamente diverso, nelle memorie statiche scrivo in un flip-flop, nelle dinamiche è un condensatore che carico o scarico; nelle memorie universali io vado sempre a leggere un valore di resistenza (alta o bassa). Ho una situazione di R_h o R_l implementando la memoria in modi diversi. Ad esempio posso avere una struttura cristallina, la scaldo molto e poi a seconda di come lo faccio raffreddare ottengo una struttura cristallina o amorfa -> cambia la resistenza opposta al passaggio della corrente; oppure ho due sostanze ferromagnetiche messa una sopra l'altra, una è fissa mentre l'altra la polarizzo in modo diverso -> se sono entrambe // avrò una certa R, se sono anti-// avrò un valore diverso (a seconda di come faccio cambiare l'orientamento della seconda ho una MRAM se uso la forza di Lorentz, STTRAM se viene fatto con particolari materiali facendo passare una corrente). Sembra quasi di essere tornati a 30 anni quando avevo i tamburi magnetici ma intanto la tecnologia è scalata paurosamente.

Caratteristiche: tutte queste tecnologie sono non-volatili (quando tolgo l'alimentazione non cambia nulla), sono estremamente veloci ma hanno ancora qualche problema di affidabilità e di endurance. Di fatto quando ho un'unica memoria che posso utilizzare come cache tanto quanto hard disk, l'architettura non è più detto che sia quella classica (all'inizio le uso come cache, poi man mano che invecchiano le posso usare come "hard disk").

Content addressable memories (CAM)



Sono memorie indirizzabili per contenuto, è una memoria molto particolare in cui rovescio il concetto classico; ho le mie parole ciascuna con un suo indirizzo, alla memoria do un dato (la parola da cercare) e in uscita mi dice se è o meno presente (se si torna anche l'indirizzo). Un uscita ho un priority-encoder perché posso avere il dato in ingresso su più words (potrei avere più match, in tal caso fornisco in uscita l'indirizzo più basso o elevato fra quelli che contengono quel dato). Applicazioni? È molto veloce, posso usarla per cercare una chiave in modo quasi istantaneo e non logaritmico all'aumentare del numero di chiavi. Supponiamo che il numero di elementi da cercare sia 1024, metto i miei 1024 elementi in una memoria e poi la scandisco in modo sequenziale. Oppure: se la mia chiave è fatta da 8 bit, se ho solo 8 bit e devo velocizzare la cosa posso usare questi 8 bit come indirizzo, se a quella chiave corrisponde uno yes allora c'è elemento (accesso O(1) contro O(n) nella ricerca di tutto il blocco). Se però la chiave è da 24 bit, la soluzione come indirizzo non è molto praticabile; la CAM può essere più ragionevole, un buon compromesso fra le due soluzioni. Se devo fare velocemente delle ricerche, per esempio nei firewall se so che il pacchetto con una particolare configurazione può essere 'pericoloso', non posso analizzarlo in software perché sarebbe troppo lento (devo prendere il pacchetto, leggerlo e analizzarlo). In realtà viene fatto attraverso una CAM, se un pacchetto è sospetto lo mando a lei. In genere la CAM viene utilizzata per velocizzare delle operazioni di ricerca.

La cella CAM è un po' più complicata del solito, in genere ha un costo maggiore di una memoria normale.

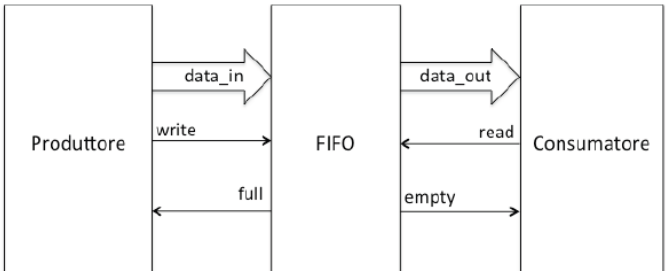
Un'altra versione è la TCAM (Ternary Cam), dove uso anche i don't care; non cerco la specifica key '1000' ma magari '10-0' (posso dargli dei don't care nella key).

Memoria associativa: non sono dispositivi singoli ma sistemi più complicati in cui posso fare ricerche per similitudini. Non è solo un don't care, le do "Christophoro Colombus" e magari lei mi trova "Cristoforo Colombo".

FIFO

First In, first Out; l'applicazione tipica è di utilizzarla come buffer di interscambio fra un produttore e un consumatore con tempi diversi. È la trasposizione in hardware della ADT, il produttore deve sapere se la FIFO è full per scriverla, il consumatore dovrà sapere se è empty (prima di leggerla controllo se è empty). Entrambi fanno un test

prima di scrivere/leggere. Ho un unico array di celle, ma ho due porte (una per la lettura e una per la scrittura) -> non ho problemi di concorrenza, consumatore e produttore non toccano mai contemporaneamente la stessa cella.



Come implementerei una FIFO in hardware usando una RAM a due porte? Una prima idea è di ricondursi all'implementare una FIFO tramite vettore in software. Il problema è che devo avere un registro per sapere qual è l'ultimo dato che ho letto e uno per sapere l'ultimo dato che ho letto -> è un buffer circolare -> ho solo il problema di capire quando è piena/vuota.

Some shared concepts

Reset: le memorie sono gli unici dispositivi sequenziali che non hanno un reset. In realtà per le memorie più complesse posso avere un reset non per l'array di memoria ma per il controllore (l'ARM o quello che è).

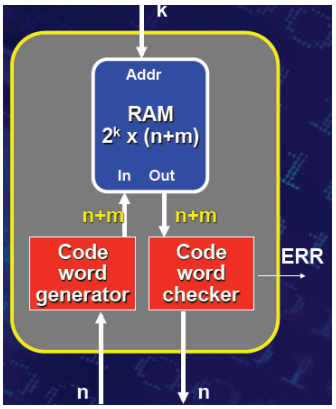
ECC: tutte le memorie (anche quelle centrali) hanno dei meccanismi di questo tipo.

*Il **bit di parità** è un codice di controllo utilizzato nei calcolatori per prevenire errori nella trasmissione o nella memorizzazione dei dati. Tale sistema prevede l'aggiunta di un bit ridondante ai dati, calcolato in modo tale che il numero di bit che valgono 1 sia sempre pari o dispari. I bit di parità sono uno dei codici di rilevazione e correzione d'errore più semplici.*

*In telecommunication, **Hamming codes** are a family of linear error-correcting codes that generalize the Hamming(7,4)-codeinvented by Richard Hamming in 1950. Hamming codes can detect up to two-bit errors or correct one-bit errors without detection of uncorrected errors. By contrast, the simple parity code cannot correct errors, and can detect only an odd number of bits in error. Hamming codes are perfect codes, that is, they achieve the highest possible rate for codes with their block length and minimum distance 3.*

Fonte: Wikipedia

Esempio: RAM da 2k x n parole, in realtà la memorie è composta da 2k x (n+m), il checker verifica n+m e se non ci sono errori butta fuori n.



I codici sono o a solo rilevazione (posso rilevare fino a k bit) o rilevazione più correzione (sempre fino a k bit). Spesso si parla di SEC_DED (Single Error correction, Double error Detection tipico per le RAM, celle da 32 bit). Nel caso delle flash invece il discorso è molto diverso, su 4k Byte magari ne devo correggere 40-50-60 errori e quindi il codice da utilizzare sarà molto diverso.

Il problema sulle flash è che correggere 40 errori con i codici BCH è abbastanza complicato, capire se c'è o no l'errore è fattibile, il problema è capire quali sono gli

errori (operazione che ruba tanto tempo). Il tempo che il codificatore impiega a capire e correggere non dipende dal numero di errori presente ma dal numero di errori che il coding può correggere -> il dimensionamento del coding va fatto sul fine vita. All'inizio vita della flash magari ho solo un bit per pagina da correggere, a fine vita ne devo correggere 50 -> già dall'inizio butto via tanto tempo. Oppure potrei farlo a geometria variabile, all'inizio il coding è da pochi errori e poi aumenta il numero di errori correggibili mentre si avvicina al fine vita.

	SRAM	DRAM	Disk	NAND Flash	NOR FLash
Cell Size	>100 F^2	6-8 F^2	(2/3) F^2	4-5 F^2	10 F^2
Read Latency	<10 ns	10-60 ns	8.5 ms	25 µs	20 ms
Write Latency	<10 ns	10-60 ns	9.5 ms	200 µs	365 µs
Energy per bit access	>1 pJ	2 pJ	100-1000 mJ	10 nJ	1 nJ (Word Read)
Static Power	Yes	Yes	Yes	No	Yes
Endurance	$>10^{15}$	$>10^{15}$	$>10^{15}$	10^4	10^5
Non-volatility	No	No	Yes	Yes	No

Le dimensioni della cella vengono fornite in tipo di F^2, ovvero data P=la minima distanza tra due pixel della litografia del chip allora F=P/2.