FSM Manual Synthetis (part 2)

Il passo successivo è passare dal diagramma a stati al circuito vero e proprio. Siamo arrivati alla STT (tabella a stati) già minimizzata, adesso devo passare alla netlist.

State encoding

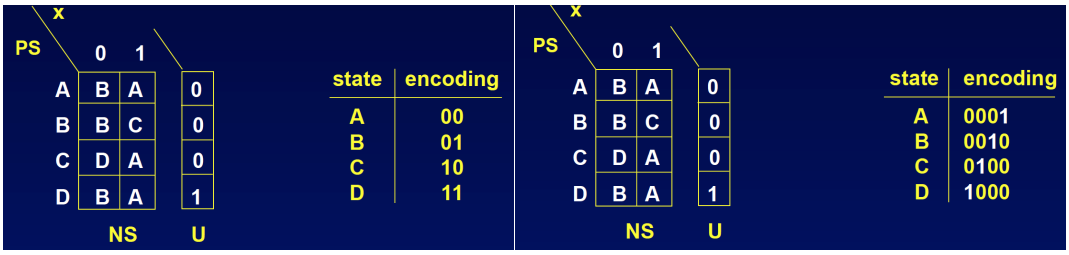
Devo assegnare a ogni stato una codifica in termini di zeri e uno.

Dato il numero degli stati, qual è il numero minimo di variabili di stato (flip-flop, elementi di reazione) che permettono di implementare tutti quegli stati? Se k sono gli stati, allora con N=numero di variabili di stati ho che:

N=ceiling($\log_2 k$) ovvero $2^{n-1}<k<2^n$

Affermazione: non c'è nessun metodo per l'assegnazione che a priori si può definire ottimale. Di fatto uso uno dei tre seguenti metodi:

- **Random**: assegno gli stati in modo random con una piccola limitazione sul reset.
- **One-hot**: ho un flip-flop per ogni variabile (non cerco di minimizzare), andamento lineare e non logaritmico. Per ogni stato c'è una solo una variabile di stato settata a 1 e le altre sono tutte a 0.
- **Heuristic-based**: basato sul random ma cerca di ottimizzare qualcosa (i tool di sintesi usano questo).



Soluzione random (a sx) Vs soluzione one hot (a dx). Fonte: testgroup.it

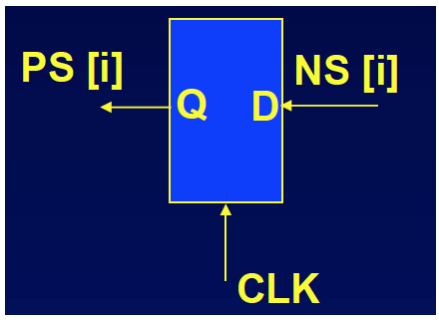
Problema del reset in sintesi manuale: abbiamo identificato uno stato come stato di reset e di li sono partita per fare il progetto. Ho supposto che il sistema entrasse in quello stato non appena il segnale di reset diventasse attivo. Adesso che devo veramente implementarlo devo garantire questa condizione fondamentale. **L'unico vincolo all'assegnamento casuale è quello di mettere tutti 0 allo stato di reset**. Inoltre dovrò collegare tutti i reset dei singoli flip-flop dello state register al segnale di reset asincrono, così azzererò tutti i flip-flop e quindi il mio circuito si troverà proprio nello stato di reset.

A questo punto basterà riscrivere la STT mettendo la codifica scelta fra 0-1 e stati , possibilmente scrivendola già come mappa di Karnaugh e coprire la tabella delle uscite e delle variabili future. Dopo di che ho ottenuto una macchina di Moore (no connessione fra PI's e la rete combinatoria delle uscite), e devo solamente implementarlo fisicamente.

LIBRARY BINDING

Prima di tutto devo scegliere il tipo di flip-flop da utilizzare, ma poiché le regole impongo uso di **flip-flop di tipo D** allora userò sempre quelli (la sintesi manuale con il tipo D è la più semplice). Poi devo fare in modo che il circuito sia tale che la variabile di stato futuro diventi la variabile presente al colpo di clock successivo. Ovvero,

NS=f(PS, PI) e PO=g(PS) allora NS_t -> PS_{t+1}



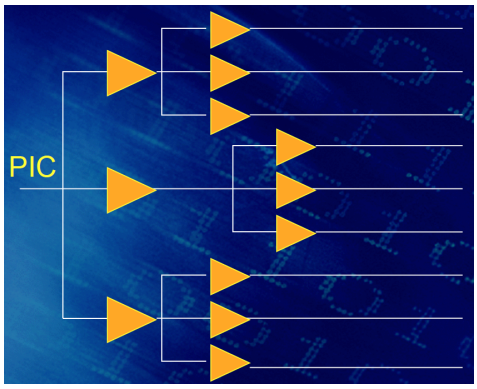
Di fatto lo state register è un flip flop, dove D sono le next-state e Q è il present-state. Utilizzando il tipo D ottengo proprio quello che mi serve, se uso gli ingressi come next-state e uscite come present-state ho risolto il mio problema. Quindi NS->D e PS->Q per cui mettendole insieme ottengo il risultato desiderato.

RT-level sequential synthesis – design rules

Regole da eseguire nella sintesi **manuale** a livello RT.

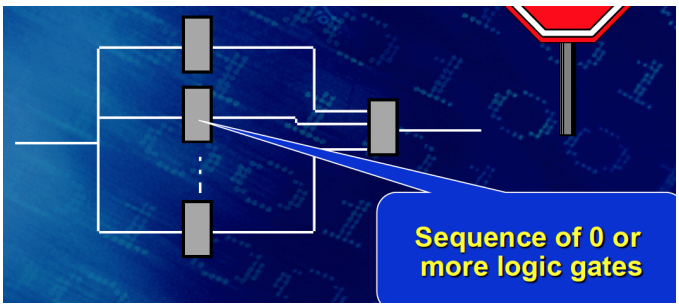
Perché si usano delle regole? Da un lato sono mirate a garantire la **portabilità**, ovvero riuscire ad avere circuiti che funzionino indipendentemente dal singolo componente fisico utilizzato, inoltre servono ad aumentare l'**affidabilità**. Tutte le regole che vedremo non sono esaustive, sono riferite solo alla sintesi manuale (i tool automatici le usano già), inoltre se io miei scopi sono un po' diversi, ad esempio se voglio minimizzare la potenza dissipata o altre cose alcune di queste regole potrebbero essere diverse o non del tutto valide.

1. **Partizionare il circuito in blocchi manageble, ovvero approccio top-down**. Separare chiaramente le parti analogiche da quelle digitali (i due blocchi parleranno con AD e DA converter).
2. Garantire sempre e in modo sistematico una netta distinzione fra segnale di clock, data e controllo. Un segnale di clock rimane un segnale di clock e basta.
3. Qualsiasi circuito deve avere **almeno un segnale di reset asincrono**.
4. Se nel circuito ho 1000 flip-flop e a questi mando lo stesso segnale di reset, con un segnale solo non potrò alimentare 1000 porte (*il numero massimo di segnali che posso far discendere da uno solo è indicato con fan-out, in genere 4 o 5*). Dovrò costruire un albero con degli amplificatori. Durante cammino che il segnale deve percorrere per arrivare alle porte, gli amplificatori introdurranno un ritardo e

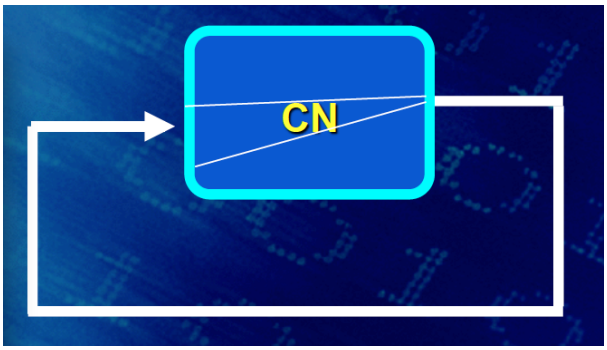


questo può essere un problema. Analogamente per il clock che dovrà pilotare tutti i dispositivi. Inoltre devo fare in modo che l'albero sia perfettamente bilanciato, ovvero tutti i cammini siano lunghi uguali, altrimenti rischierei che i flip-flop più lontani siano resettati mentre gli altri stanno già funzionando. Manualmente lo risolvo implementando delle macchine ad hoc per il timing.

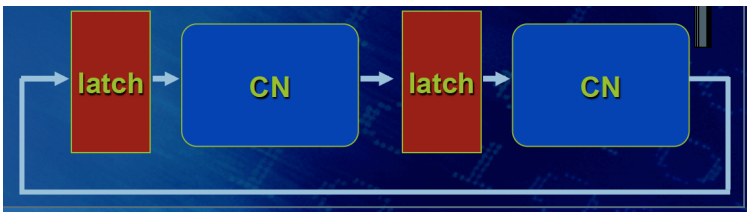
5. **Assoluta sincronicità**:
   - **i.** Evitare strutture di questo tipo:



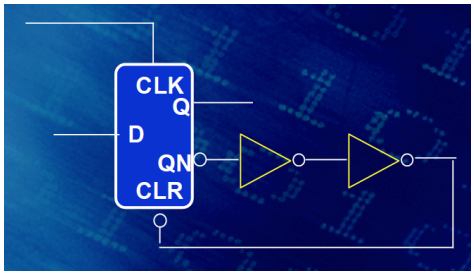
   - **ii.** Anche nel caso dell'Hamlet Circuit, in prima battuta la prima domanda che ci poniamo è a cosa può servire, visto che avrei sempre 1 in uscita. In realtà non ho considerato il ritardo fra le porte. Quindi questo è un modo furbo per passare da fronte a impulso, peccato che non sia asincrono.
   - **iii.** Loop asincroni non sono permessi:



   - **iv.** Anche reti di questo tipo non sono permessi perché il latch è una macchina di Mealy:



   - **v.** No ai self-resetting flip-flop (per lo stesso motivo del circuito di Amleto):



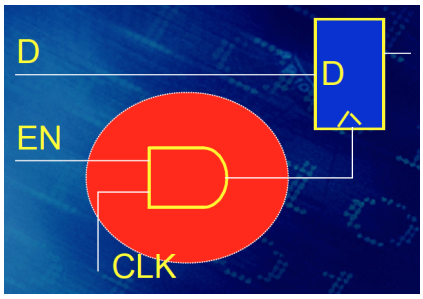
   - **vi.** No circuiti Monostabili.

vii. I segnali di reset asincroni li posso utilizzare **solo** durante l'inizializzazione mentre durante il normale funzionamento dovrò utilizzare solo reset sincroni.

viii. Se faccio un circuito e lo metto in libreria, chi lo usa non sa in quanti rami viene mandato un segnale in ingresso e quindi non sa se può riutilizzare quel segnale per mandarlo anche da altri parti. Invece dicendo di mettere sempre un flip-flop all'ingresso e uno all'uscita, non ho problemi fi fan-out e posso riusare quel segnale agevolmente.

ix. Solo flip-flop di tipo D.

6. No ai clock signal generator.
7. Clock gating: da non usare mai. Se voglio disabilitare il flip-flop non posso mettere una porta prima in cui metto insieme il clock e l'Enable, questo circuito non è più perfettamente asincrono perché la porta introduce un ritardo e quindi il flip-flop commuta sempre con un ritardo. (oggi i ritardi più lunghi sono quelli dovuti alle connessioni e non alle porte, ma in questo corso non facciamo così i sofisticati). Nella realtà sia l'enable con ingresso separato sia con il clock gating vengono utilizzati, il primo è perfettamente sincrono ma il secondo consuma di meno. Negli ultimi processori per consumare di meno si toglie proprio l'alimentazione alle parti non utilizzate, inoltre a seconda della modalità di consumo regolo la tensione di alimentazione e quindi la frequenza del circuito. Inoltre per evitare che il circuito smetti funzionare perché invecchiando i ritardi aumentano, si mettono dei sensori che riducono la frequenza di clock quando il processore invecchia.



8. Il clock tree deve essere perfettamente bilanciato.
9. Non usare mai ingressi flottanti (perché altrimenti si becca tutti i disturbi del mondo).
10. Se ho un bus, allora devo sempre avere un valore sul bus per problemi di affidabilità.

Fra tutte queste la più importante è la perfetta sincronicità.

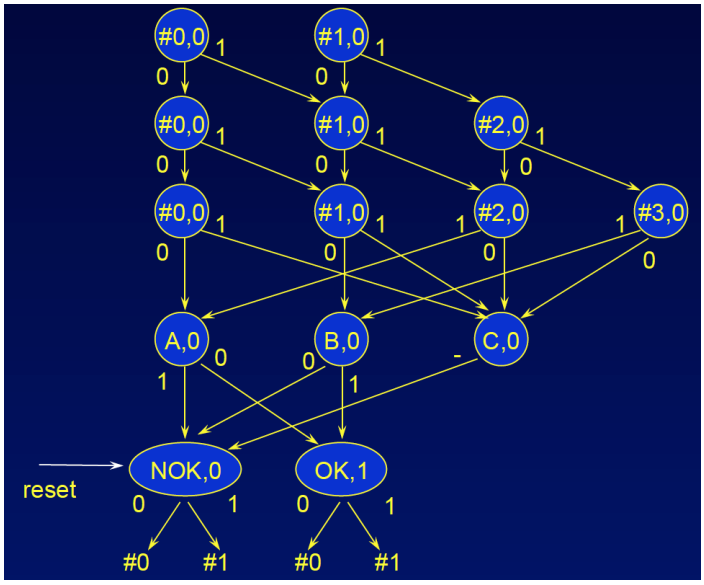
STG Design Examples

Esercizio 7.37

"On a serial transmission line X, bits are transmitted synchronously w.r.t. a clock signal CLK, one bit per clock cycle. The line is used to transmit groups of 5 bits. In each group, the first 3 bits are a data and the remaining 2 bits are a code associated to the data to detect transmission errors. In particular, for each group of bits, the code encodes the number of bits equal to '1' in the data of the same group. Codes are transmitted Most Significant Bit (MSB) first. A circuit to be connected to the serial line is to be designed, such that its output OK is asserted for one clock cycle iff, at the completion of a group, no transmission error has been detected."

Il problema di questo circuito è che il codice di controllo non è molto restricted, perché se mando 101 e ricevo 110 allora ho sempre due 1 ma il data è diverso.

Prima conto il numero di uni e poi lo confronto con il codice, dovrei implementare questo ragionamento in hardware. Alla fine del terzo bit di data ho 4 situazione possibili (0, 1, 2 o 3 bits). Parto dal reset, al terzo bit sono o in #0, #1, #2 o #3, in questo modo ho contato il numero di uno (non ho un contatore, devo farlo attraverso gli stati). Poi anziché nella IV riga avere No, forse 1, forse 2, forse 3 basta avere un sicuramente no e altri due stati in cui a seconda del V bit vado in ok o ko. Inoltre NOT_OK e reset sono equivalenti.



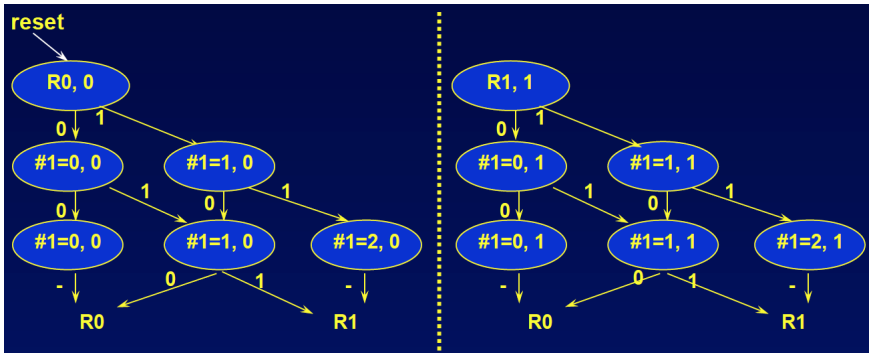
Hint: *attenzione se un circuito deve fare più operazioni insieme.*

Esercizio 2_2.009

"On a serial transmission line, data are transmitted according to a "2 out of 3" protocol:

• The transmitter encodes each data bit as 3 consecutive identical bits (e.g., '1' as '111'). No padding bit is inserted between two consecutive data.

• The receiver analyses each group of 3 bitsand decodes it:
- as '0' when at least two bits in the group are equal to '0',
- as '1' otherwise.
A circuit is to be design to be connected to the transmission line to regenerate the received data:
• it receives data from an input line,
• it interprets them according to the protocol,
• it transmits them on an output line, with a minimum delay, according to the same protocol (e.g., when it receives '010' it transmits '000', when it receives '011' it transmits '111', when it receives '111' it transmits '111' and so on)."



2 out of 3 protocol. Il circuito deve fare due cose diverse, ho 1 ingresso e 1 uscita, deve essere sia ricevitore sia trasmettitore. Mentre ricevo il gruppo n+1 io sto ancora trasmettendo il gruppo n. Se non dovessi trasmettere non avrei problemi, qui basta mettere le due cose insieme. Inoltre non posso sull'uscita mettere 000 perché posso solo far uscire un bit per volta. Nei primi 3 bits dal reset manderò in uscita sporcizia. È dall'altra parte che interpreterò la prima sequenza come "don't care".

Hint: in alcuni casi potrebbe essere utile partire subito dalla STT.

Esercizio 2_2-021

Up-down counter in modulo 8.

Programmazione: Liste

Molto usate quando non conosco a priori il numero di elementi su cui andrò a lavorare e/o quando la cardinalità può cambiare durante il programma. Per esempio con l'allocazione dinamica io non posso cambiare a runtime la dimensione dell'area di memoria richiesta (una volta che ho chiesto 100 byte attraverso la funzione new, non posso chiedere di aggiungerne altri 50 byte ai 100 già allocati ma devo prima dismettere i vecchi 10 e poi richiedere 150). Invece attraverso le liste prendo solo la memoria man mano che mi serve. Le liste sono strutture dati che permettono di aggiungere elementi, di concatenare un anello a un anello già preesistente, e di dismettere gli anelli non usati senza dover deallocare tutto quanto.

Liste

Le liste sono un tipo di dati astratto, contengono i dati su cui andrò a lavorare e delle funzioni per operare su questi. Permettono l'inserimento, la ricerca e la cancellazione di un qualunque elemento.

La lista è una sequenza di elementi tutto dello **stesso** tipo, n è il numero di elementi che la compongono, allora se n=1 allora la lista è vuota (empty list), a_1 è la testa (head) dell'elemento, a_n è a coda (tail) della lista. A seconda dell'implementazione che adotteremo, il concetto di posizione avrà un significato diverso (puntatore, indice,). Per velocizzare le operazioni sulla lista, può essere utile avere un elemento in posizione successiva all'ultimo elemento valido della lista. Quest'ultimo elemento è il parametro di ritorno della funzione eol() (End Of List).

Operazioni sulle liste

Tutte queste funzioni sono in pseudo codice.

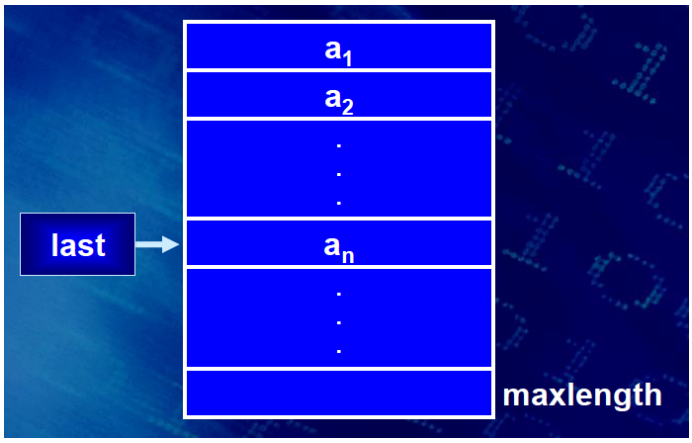
- Insert (x,p,L) , inserisce l'elemento x in posizione p nella lista L. Si può anche aggiungere nel centro della lista, torna ok se avvenuto con successo, error altrimenti.

- Delete (p, L), cancella elemento in posizione p e ricompatta gli elementi (tutti gli elementi da posizone p+1 vengono shiftati di 1).
- Locate(x,p,L) -> cerco il valore x a partire dalla posizione p, torna posizione della prima occorrenza, se non trovo nessuna occorrenza torna eol().
- Retrieve (p, L) -> torna elemento in posizione p, torna error se non esiste.
- Next (p, L), previous(p,L) -> torna le posizioni prima/dopo p, torna error se chiedo next(eol(L), L),.
- Makenull(L) -> svuoto completamente la lista e torno eol(). Se è implementata in modo statico metto a un certo valore tutte le celle, se in modo dinamica faccio il delete delle celle.
- First(L)-> torna l aprima posizione nella lista (quella di testa), torna eol() se la lista è vuota.

Possibili implementazioni

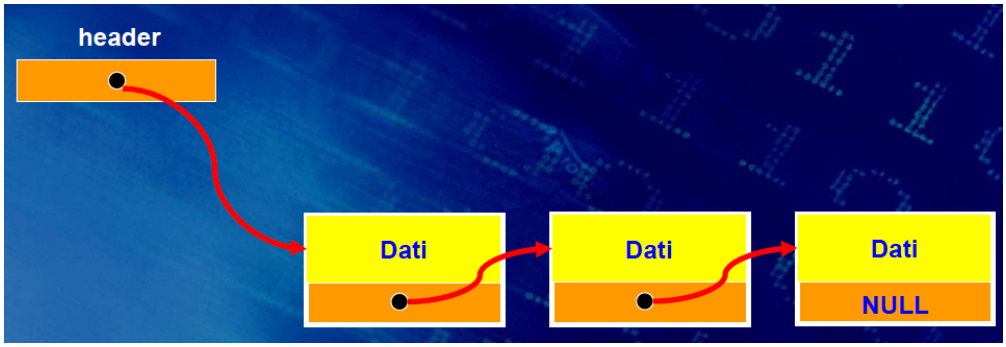
Ci sono diverse implementazioni, la distinzione principale è nella posizione dell'elemento successivo. L'ordinamento non è richiesto. Se è un array è l'elemento successivo, se uso i puntatori potrebbe non essere nella cella successiva ma da qualche altra parte. Se in una lista so che opero sempre in testa o in coda non mi conviene usare i puntatori, se so che andrò a lavorare in qualunque parte della lista conviene usare i puntatori (molto più efficiente). Con una lista di array, se devo mettere un elemento a metà, devo spostare tutti gli elementi avanti. Con i puntatori basta aggiornarli. L'indice viene utilizzato se lavoro con array bidimensionali. Gli elementi della lista sono contigui (se lavoro con gli array no problem, se invece lavoro con puntatori allora questi saranno tutti vicini ma possono andare a puntare da qualunque parte).

Vettori



Vantaggi e svantaggi: il vettore va dimensionato in base al numero massimo di elementi che lo comporranno, a un certo punto del programma qualcuno mi deve dire la dimensione massima (no #define). L'inserimento e la cancellazione richiedono lo spostamento di tutti gli elementi rimanenti. È vero che con l'allocazione dinamica di un array posso ottenere gli stessi risultati (alloco e dealloco sempre tutto), ma questa soluzione non è proprio molto efficiente. L'implementazione con i vettori è vantaggiosa se il numero di inserimenti/cancellazioni in testa o in coda sono predominanti rispetto agli altri.

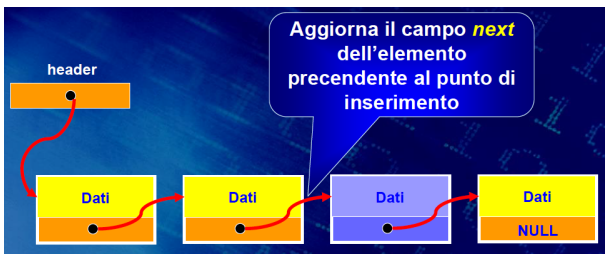
Puntatori



Ogni cella del mio array diventa un oggetto(record) (una struct se sono in C, un oggetto se sono in C++). Tutti gli oggetti sono allocati dinamicamente. La finezza è che ogni oggetto non solo contiene il valore del mio dato ma contiene anche un puntatore all'elemento successivo. Ho quindi una sorta di catena che tiene connessi tutti gli elementi, quindi basta aggiornare i puntatori e linkare il nuovo oggetto alla lista preesistente). In questo caso non ho limitazioni di dimensioni. In genere si usa un puntatore chiamato header che punta al primo elemento della lista. L'header è un elemento della lista ma non contiene nessun dato utile. #todo disegnino.
Tutti gli oggetti sono dello stesso tipo (stessa classe o stessa struct), l'ultimo elemento ha come puntatore dell'elemento successivo a NULL.
A questo punto l'inserimento consiste nell'istanziare il nuovo oggetto e linkarlo agli oggetti già esistenti (non devo deallocare tutta la lista e poi allocarla con dimensione n+1). Semplicemente mi preparo l'oggetto n+1 e poi lo link.

Inserimento in testa: creo dinamicamente un nuovo elemento, lo riempio con i dati, copio l'header nel campo next del mio nuovo oggetto (se l'header puntava a 12, ora il next del mio nuovo oggetto deve essere 12) e poi aggiorno l'header con l'indirizzo dell'oggetto che ho creato.

Inserimento in centro: creo un nuovo elemento in modo dinamico, lo riempio, devo identificare il punto d'inserimento, aggiorno il campo next del nuovo elemento all'elemento successivo al nuovo e aggiorno il campo next dell'elemento precedente al nuovo al nuovo oggetto. È come se la lista fosse collegata da fili e io devo staccare i fili e attaccarli opportunamente. Giocando con i puntatori, fisicamente i dati posso essere ovunque ma io tengo tutto insieme con i puntatori.

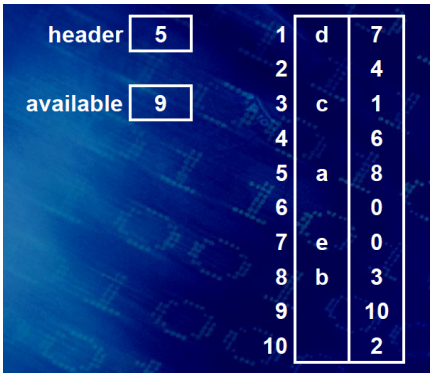


Inserimento in coda: creo elemento, lo riempo, il campo next dell'elemento che ho creato viene messo a NULL e il vecchio elemento di coda avrà come puntatore next all'oggetto che ho creato.

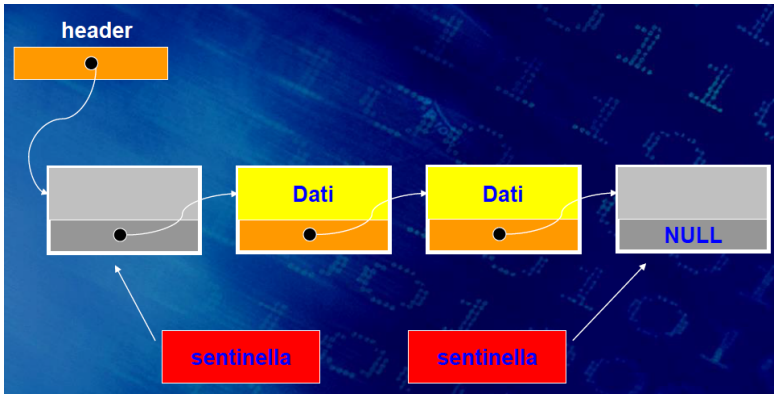
Invece per la cancellazione ho che l'elemento precedente a quello da cancellare avrà come next il next dell'elemento da cancellare e prima di deallocare l'elemento da cancellare per precauzione setto il suo next a NULL. Lo spazio occupato è determinato solamente dal numero di elementi che effettivamente utilizzo . L'altro vantaggio che l'inserimento e la cancellazione non richiedono lo shiftamento di tutta la lista, basta aggiornare due puntatori. Ovviamente i puntatori sono delle variabili (occupano spazio), occupo un po' più di memoria (perché ho dato+puntatore) rispetto all'implementazione con array. In situazioni molto dinamiche questa è la soluzione migliore.

Indici

Tramite indici: vengono utilizzati in quei linguaggi dove non posso utilizzare i puntatori, emulo l'implementazione con i puntatori attraverso un array bidimensionale. Su una colonna ho le informazioni utili e dall'altra parte ho indici per scorrere la lista. Non è una gestione proprio dinamica come con i puntatori ma ho più flessibilità rispetto alla soluzione con array.



Le sentinelle

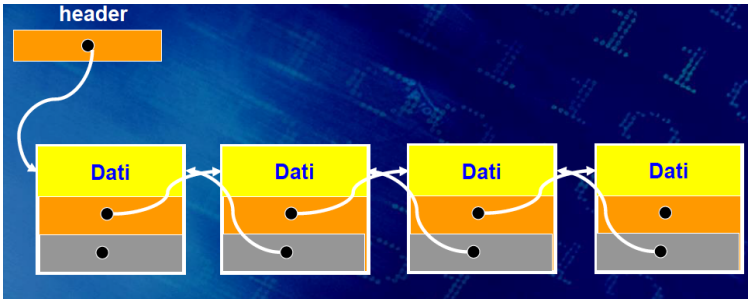


Sono elementi fittizi (oggetti che non contengono dati utili) che vengo aggiunti alla testa e alla coda per velocizzare alcune operazioni.

Ho sempre l'header, ho un elemento dummy (sentinella) subito dopo l'header e uno alla fine. In genere vengono utilizzati nelle liste ordinate, in questo caso le sentinelle contengono il valore minimo e quello massimo memorizzabili, così è più veloce perché posso eliminare il test di fine lista.

La ricerca all'interno di una lista non ordinata devo scandagliare tutta la lista e controllare che l'ultimo elemento abbia puntatore null (quindi devo fare un confronto sul valore e sul puntatore).

Liste doppie e multiple



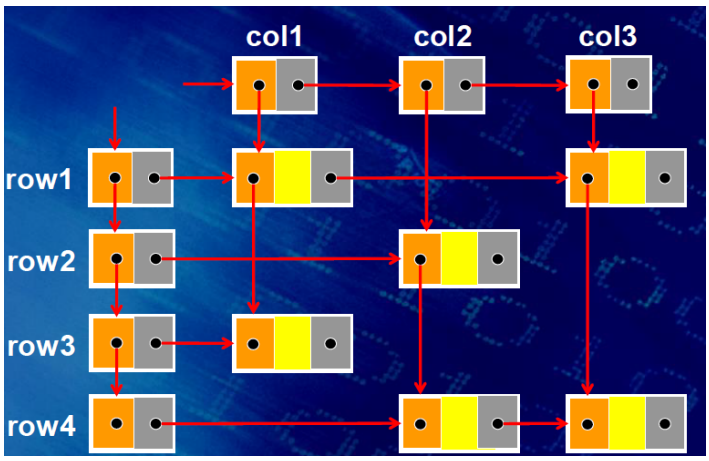
Sono ancora più flessibili perché posso navigare sia da testa->coda sia da coda->testa (i fili non hanno solo più un verso, se voglio tornare all'indietro di una posizione non devo ripercorrere tutta la lista). A questo punto mi serve una lista doppiamente linkata (in ogni oggetto ho un puntatore al successivo e uno al precedente). Quindi in ogni oggetto ho due puntatori, quindi per fare l'inserimento/cancellazione non devo modificare più 2 puntatori ma 4. Ho più flessibilità ma occupo il doppio di spazio per i puntatori (2*size(int)).

Liste circolari

Sono liste dove l'ultimo oggetto punta al primo. Devo solo fare check in più sugli indici.

Liste multiple

Sono liste dove ogni elemento ha più di un puntatore, ho una specie di matrice dinamica di puntatori. Un tipico esempio d'uso è nella rappresentazione di matrici sparse (risparmio memoria), va bene se ho un numero esiguo di elementi.



Stacks & Queues

Sono liste specializzate, hanno regole ben precise.

STACK

Def. *È una lista dove tutte le operazioni di inserimento e cancellazione avvengono solo da un'estremità (dalla testa), chiamata top.*

È molto simile a una pila di piatti. Posso solo aggiungere in cima e togliere quelli in cima. Realizza una struttura di tipo LIFO (Last in First Out). Tutte le architetture dei calcolatori prima degli anni '80 erano basati su memorie di tipo stack, non potevo accedere alla memoria dove volevo e trattarla in modo dinamico, potevo solo accedere all'ultimo registro.

Operazioni sugli stack

- Makenull
- Top(S): ottengo l'elemento in testa.
- Push(x, S): inserisco elemento in testa, spostando gli altri verso il basso.
- Pop(S): torna l'elemento cancellato e sposta gli altri verso l'alto. (prendo l'ultimo piatto).
- Empty(S): controlla se la lista è vuota.
- Full(S):

Esempio d'utilizzo

Un parentesi checker, controlla se a tutte le parentesi aperte vengono poi anche chiuse (e anche nell'ordine corretto). Posso farlo attraverso vettori o liste linkate, è sempre richiesto un elemento che punti al top dello stack.

- Attraverso array: ho bisogno di un elemento di N array. La criticità è che la dimensione è sempre fissata (se voglio aumentarla devo deallocare e allocarla più grande).
- Linked list implementation: mentre prima avevo l'header abbastanza generalizzato, adesso invece tutte le operazioni di push e pop passano attraverso il top. Qui le operazioni sono più semplici rispetto alla lista generica perché opero sempre sul primo elemento.
- 

Code

Particolare tipo di lista dove tutte le operazioni di inserimento avvengono in coda e quelle di cancellazione avvengono in testa (come alle poste). Ho un grado in più di libertà rispetto allo stack. Vanno molto bene quando devo fare lo scheduling di qualcosa (FIFO, first in first out); chi prima richiede prima ottiene, ho una serie di richieste e tratto queste richieste attraverso la FIFO).

Operazioni

- Makenull
- Front
- Enqueue(x,Q)-> aggiungo elemento in coda.
- Dequeue(x,Q)

Implementazioni

Ho bisogno di un elemento top che punti alla testa e di una variabile che sia alla fine.

- Array: array allocato dinamicamente, una variabile head che punti all'elemento più vecchio, e una variabile che punti al primo elemento libero in coda (il puntatore all'ultimo elemento non m'interessa perché non posso prelevarlo, in coda m'interessa scrivere). Un modo efficiente per trattarla è utilizzare un array circolare, i puntatori circolano all'interno dell'array, arrivati a N la posizione successiva è 0. Sono io programmatore che all'array applico questa gestione circolare. Il problema è capire se la coda è piena o vuota; per ovviare a questo problema ci sono diverse possibili soluzioni.

Laboratorio 5

Quando faccio accesso a un file in modo sequenziale, non è che non posso modificare l'informazione in un punto qualsiasi senza non sovrascrivermi le informazioni che seguivano. Se voglio fare l'aggiornamento di un file sequenziale devo copiarmi su un nuovo file la parte vecchia, aggiungere quello che mi serve e poi in append aggiungere la parte dopo. Poiché il file non è strutturato non posso shiftare tutto di quante posizioni che io voglio. È **uno stream di bit**, non posso aggiungere delle patch a metà. Questo è uno spreco perché se voglio scrivere un file devo andare a scriverlo sull'hard disk e non sulla RAM o cache-> penalizzazione ulteriore.

L'alternativa è fare un file ad accesso randomico (l'editor di testo è un software molto complesso che mi fa vedere solo le cose più banali, quello che fa lo sa solo lui). **Il file è un flusso di bit persistente**. La semantica associata allo stream (come lo leggo e come lo utilizzo) è lasciata al programmatore, solo lui sa come buttarlo fuori e come leggerlo. Gli standard

servono a uniformare l'interpretazione delle informazioni. È solo un flusso di bit che deve essere storato. Scrivere non è un grosso problema, il problema è nella lettura, devo sapere come interpretarlo (l'ASCII è codificato su 8 bit, l'editor di testo prende 8 bit per 8 bit e lo decodifica). I file eseguibili (*.exe) solo il supporto a runtime e il SO sa come interpretarlo, anche i programmi sono dei file ma i tipi sono diversi.

I file ad accesso casuale servono per muoversi all'interno del file sapendo che il file è una specie di array di una mia struttura che conosco. Già io li ho sovradimensionati, alloco staticamente delle struttura all'interno dei file e poi muovendomi all'interno di questo posso andare a toccare solo la singola i-esima struttura.

Gestione dinamica: per funzionare i dati devono stare nella RAM, e questo posso farlo in modo statico (quando parto con il programma il supporto a runtime sa quanto allocare), o in modo dinamico (magari non so subito quanto grande farlo, il supporto a runtime attende ad allocare, così se altri programmi hanno bisogno di memoria ne hanno a disposizione). Così evito di sovrastimare e quindi di sprecare (gestione più fair).

Per allocare dinamicamente la memoria userò senz'altro i puntatori (perché' non conosco a priori dove sarà allocata la memoria, mi serve un puntatore per sapere dove sarà il mio oggetto). Viene allocato staticamente solo il puntatore, sono 32 bit che se ne vanno sempre, ma il puntatore viene settato a runtime.

```
Persona *phoneBookPtr;

      phoneBookPtr=new(Persona[N]);
```

new è il corrispettivo della malloc() (ma più potente), devo solo passare il tipo della variabile che voglio allocare. Se avessi passato solo Persona allora avrei allocato solo un oggetto. **La memoria dinamica si usa sia quando non conosco a priori quanta memoria allocare, sia quando so quanto allocare ma so che dopo un po' non mi serve più e quindi che deallocando riesco a recuperare spazio.** Se invece passo Persona[10] o int[10] allora sto chiedendo un array di 10*tipo. Serve il tipo perché devo sapere qual è la dimensione della singola "cella" da allocare.

Esercizio 2:

Creare rubrica telefonica in cui ogni record è composto da 3 stringhe. Nella parte privata di una classe posso mettere anche oggetti di un'altra classe. Per richiamare un metodo non su un oggetto ma sul puntatore all'oggetto non si utilizza il "." ma la "->" (uso il punto se l'oggetto è allocato staticamente, la freccia se è allocato dinamicamente). Per cui;

```
Persona *phoneBookPtr;
      phoneBookPtr=new(Persona[N]);
      for(i=0; … ) {
            cout>>…
            cin>>nameVal;
            (phoneBookPtr+i)->setName(nameVal);
            …
      }
```

Per deallocare l'area di memoria occupata uso la funzione delete.

Esercizio 3

So che posso al massimo gestire 100 clienti. Ogni record deve contenere un numero univoco che identifichi il cliente, nome, cognome e quanti soldi possiede. La gestione dei file è fatta in modo randomico. Ho 100 celle di tipo cliente allocate sul file. Quindi devo sapere a priori quanto è grande la singola cella (o comunque devo usare le variabili che staranno nella cella in modo statico). Il costruttore non ha tipo di ritorno e ha lo stesso nome del nome della classe.

Per leggere file in modo randomico devo usare la funzione read;

```
read(<indirizzo di dove deve iniziare a scrivere>, <dimensione di ciò che devo scrivere>);
// io leggo e scrivo tutto l'oggetto insieme, non variabile per variabile. In realtà c'è anche una parte di header.
```

Analogamente per la write() richiamata sull'outStream.

Sia in read sia in file ho comunque bisogno di una variabile del tipo desiderato per sapere cosa scrivere e dove buttare ciò che leggo.

Per accedere al 5^ cliente uso la funzione outFile.seekp(4*sizeof(ClientData)); -> sposto il cursore al V elemento.

Prima di muovermi nel file devo scrivere sul file 100 oggetti "vuoti".