MODELLI DEI DATI

Come si immagazzinano dati nel calcolatore?

Prima di tutto definiamo dei MODELLI DI DATI ASTRATTI.

Un ADT (ABSTRACT DATA TYPE) è costituito da un insieme di oggetti e da un insieme di operazioni (tipicamente AGGIUNGI, ELIMINA e CERCA) che agiscono su di esso.

L'ADT più semplice è il DIZIONARIO:

un insieme di oggetti su cui possono essere effettuate operazioni di: INSERZIONE, CANCELLAZIONE e RICERCA.

ESEMPIO: - registro di classe
- iscritti a un corso o a un esame
- lista della spesa

Un ADT può essere REALIZZATO con diverse REALIZZAZIONI ASTRATTE cioè dei modelli che contengono una serie di operazioni logiche e ben definite che permettono appunto di realizzare l'ADT. A loro volta le REALIZZAZIONI ASTRATTE vengono realizzate con STRUTTURE DATI, SUBROUTINE e FUNCTIONS di un determinato linguaggio di programmazione.

LISTA: è una sequenza finita, anche vuota, di elementi di uno stesso tipo.

Sono sempre indicate come: $(a_1, a_2, a_3, \dots, a_n)$.

La lunghezza di una lista è il numero dei suoi elementi.

Se la lunghezza è zero la lista è vuota.

Se una lista non è vuota essa è costituita:

- da un primo elemento, detto TESTA (HEAD)
- dal resto della lista, detto CODA (TAIL)

SOTTOLISTA: è una lista che si ottiene prendendo dalla posizione i fino alla posizione j

SOTTOSEQUENZA: è una lista che si ottiene dalla lista di partenza rimuovendo 1 o più elementi.

PREFISSO: Qualsiasi sottolista che si ottiene a partire dalla testa

SUFFISSO: Qualsiasi sottolista che termina alla fine della lista

---

ESEMPIO:

lista per nobili:

(elio, neon, argon, krypton, xenon, radon)

TESTA → elio

CODA → (neon, argon, krypton, xenon, radon)

SOTTOLISTA → (argon, krypton, xenon)

SOTTOSEQUENZA → (neon, krypton, radon)

SUFFISSO → (xenon, radon)

OCCORRENZE: in genere uno stesso elemento può comparire più volte in una lista. Il numero di volte che un elemento compare è detto occorrenza.

A ciascun elemento di una lista viene assegnata una posizione, cioè un intero i tale che $1 \leq i \leq n$ con n lunghezza della lista.

OPERAZIONI SU LISTE

1) INSERIMENTO: si inserisce un elemento x nella lista. x può in teoria essere inserito ovunque. Un elemento può essere inserito anche se compare già nella lista.
Un caso particolare, molto frequente, è quello in cui un elemento viene inserito alla testa della lista $\rightarrow$ PUSH

2) CANCELLAZIONE: eliminiamo un elemento x dalla lista. Se x vi è più di una occorrenza: o cancelliamo la prima occorrenza, cancelliamo la seconda, le cancelliamo tutte, ecc.
Se x non compare nella lista, la cancellazione non ha effetto.
Un caso particolare è la cancellazione della testa della lista detta POP.

3) RICERCA: cerco un elemento x nella lista, se lo trovo restituisco .TRUE., se non lo trovo .FALSE.

4) CONCATENAZIONE: si considerano due liste L e M e si ottiene una lista LM che comincia con gli elementi di L e finisce con quelli di M.

$$L = (a_1, a_2, \dots, a_n) \qquad M = (b_1, b_2, \dots, b_k)$$

$$\Rightarrow LM = (a_1, a_2, \dots a_n, b_1, b_2, \dots b_k)$$

5) PRIMO(L) restituisce il primo elemento della lista, ULTIMO(L) restituisce l'ultimo elemento.

6) ACCEDI(i,L) restituisce l'elemento in posizione i della lista L. Si ha errore se i è maggiore della lunghezza.

7) LUNGHEZZA(L) restituisce la lunghezza.

8) VUOTA(L) restituisce TRUE se L è vuota.

DAL MODELLO ASTRATTO DI LISTA DOBBIAMO PASSARE ALLA SUA IMPLEMENTAZIONE IN UN LINGUAGGIO DI PROGRAMMAZIONE:

- USO DI LISTE CONCATENATE
- USO DI ARRAY

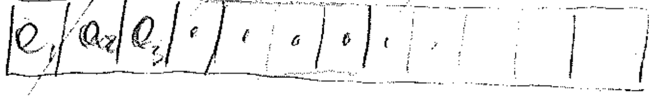
Un modo per realizzare le liste è mediante la STRUTTURA DATI "LISTA CONCATENATA"

La lista concatenata viene realizzata mediante una sequenza di CELLE o NODI ciascuna delle quali contiene:

- 1 OGGETTO (ovvero l'informazione che voglio memorizzare, per es. INTEGER, REAL(KIND=8), etc.
- 1 PUNTATORE alla prossima cella

Mi muovo attraverso le celle usando i puntatori e cancello o aggiungo celle "muovendo" i nuovi puntatori.

L'altro modo di implementare le liste è attraverso array



Dedico un certo numero di elementi dello stesso tipo che mi interessa memorizzare alla memorizzazione della lista.

Ho bisogno di una struttura dati di questo tipo:

```
TYPE LISTA_ARRAY
     INTEGER :: LUNG_MAX
     INTEGER :: LUNG
     CHARACTER(LEN=10), ALLOCATABLE :: NOME
END TYPE
```

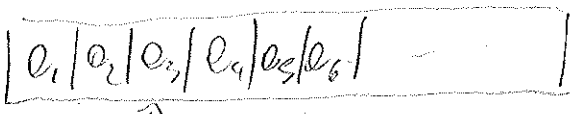
Quando creo la lista devo specificare la lunghezza massima e inizializzo la lunghezza attuale a 0

```
FUNCTION CREA_LISTA(LMAX) RESULT(NUOVA_LISTA)
INTEGER, INTENT(IN) :: LMAX
TYPE(LISTA_ARRAY) :: NUOVA_LISTA
ALLOCATE (NUOVA_LISTA%NOME(LMAX))
NUOVA_LISTA%LUNG_MAX = LMAX
NUOVA_LISTA%LUNG = 0
END FUNCTION CREA_LISTA
```

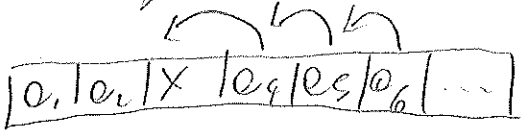
La ricerca può avvenire nel modo seguente

```
FUNCTION CERCA(NOME_IN, L) RESULT (TROVATO)
INTEGER, INTENT(IN
```

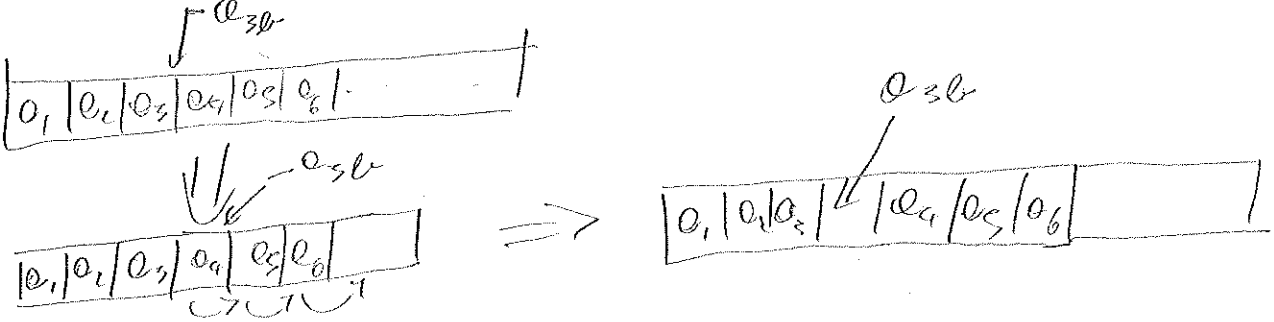
Il problema principale con le liste realizzate con gli array è l'inserimento e la cancellazione di un dato elemento:



Se voglio cancellare e_3, poi mi rimane un buco e quindi devo compattare la lista!!



Se invece voglio inserire un elemento in una posizione diverse dalla coda devo creare spazio:



Inoltre rispetto alla lista concatenata le liste con gli array OCCUPANO più memoria!!

La lista concatenata è più DINAMICA, occupa memoria solo quando serve, la lista con array occupa subito la memoria che PENSO mi possa servire.

Inoltre la lista con array si può RIEMPIRE e per continuare devo allocare un'altra lista e portare il suo contenuto.

REALIZZAZIONE DEI DIZIONARI ATTRAVERSO LISTE

DIZIONARI: collezione di oggetti in cui posso INSERIRE, CANCELLARE e CERCARE.

Se lista in generale ammette duplicati e questa è una differenza fondamentale con i dizionari. Se voglio realizzare un ADT dizionario con una lista devo decidere come comportarmi con i DUPLICATI.

Per esempio l'implementazione standard della lista non tiene conto dei duplicati e quindi il programmatore deve effettuare delle scelte:

- CASO 1) Sono sicuro che non avrò mai dei duplicati, per esempio se memorizziamo il numero di matricola (unico) degli studenti di una classe ⇒ Posso usare l'implementazione standard
- CASO 2) Potrebbero esserci dei duplicati, devo decidere come comportarmi:
  - 1) Posso memorizzare i duplicati
  - 2) Posso non memorizzare i duplicati.

  Per entrambi i casi devo modificare l'implementazione standard

Supponiamo di NON MEMORIZZARE i duplicati

- Devo modificare l'implementazione standard in modo da fare un controllo prima di inserire un nuovo elemento!! Se c'è già non inserisco.

Se memorizzo i DUPLICATI, l'inserimento di nuovo velocissimo, perché posso inserire subito un elemento in qualsiasi posizione.
Purtroppo la cancellazione è più lenta perché devo controllare tutta la lista per essere sicuro di averne eliminato tutti.
Anche la ricerca rallenta, perché in generale n ≠ il numero di elementi che vogliamo trovare nel dizionario, la lista con duplicati avrà lunghezza m.

Un altro modo per realizzare le liste, mediante LISTE ORDINATE SENZA DUPLICATI. Creo una lista senza duplicati in cui ho preso cura di mantenere in ordine gli elementi. In questo modo se trovo un elemento più grande di quello che sto cercando posso interrompere la ricerca.

2, 8, 9, 15, 24 ← Se cerco 7, mi fermo quando trovo 8

È più complicato da implementare ma più conveniente.

CONFRONTO TRA I METODI

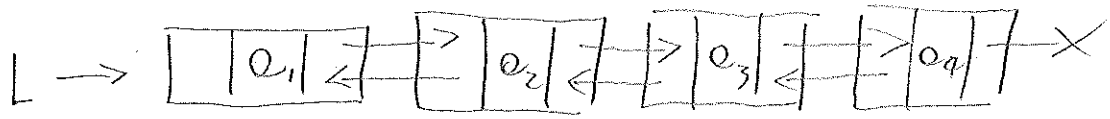
TABELLA DI COMPLESSITÀ

	INSERIMENTO	CANCELLAZIONE	RICERCA
SENZA DUP.	$n/2 \to n$	$n/2 \to n$	$n/2 \to n$
CON DUP.	0	m	$n/2 \to m$
ORDINATA	$n/2$	$n/2$	$n/2$

$a \to b$ indica che la complessità è in caso di successo, è in caso di insuccesso.

LISTE BIDIREZIONALI

Puntatore con le liste standard posso muovermi solo in una direzione. Potrebbe essere utile muoversi anche nell'altra ⇒ liste bidirezionali o doppio concatenate



STRUTTURA DATI IN FORTRAN

```
TYPE NODO_BID
    CHARACTER(LEN=10) :: NOME
    TYPE(NODO_BID), POINTER :: PRECEDENTE, PROSSIMO
END TYPE
TYPE LISTA_BID
    TYPE(NODO_BID), POINTER :: NODO_0
END TYPE
```

PILE o STACK

Le PILA è un altro ADT basato sul modello delle liste, in cui tutte le operazioni vengono effettuate su un solo estremo della lista, detto CIMA.
Sinonimo di PILA è "LISTA LIFO"

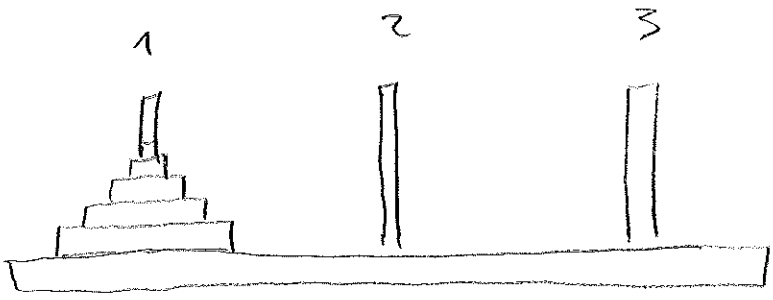
LIFO = LAST IN FIRST OUT

Le operazioni tipiche nelle PILE sono le seguenti:

- PUSH: inserimento di un elemento nella PILA (in CIMA)
- POP: estrazione dell'elemento dalla PILA (dalla CIMA)

ESEMPI:

- P.lo di resti:
- Torre di HANOI:



- CALCOLATRICI IN NOTAZIONE POLACCA INVERSA (RPN)

le espressioni matematiche possono essere scritte in notazione POSTFISSA

$(3+4)\cdot 2 \iff 3\,4\,+\,2\,\cdot$

le espressioni si leggono da SX a DX e appena si trova un operatore si esegue l'operazione richiesta sugli operandi immediatamente a sinistra. È molto comodo nelle calcolatrici perché si possono scrivere espressioni molto complesse senza le parentesi

Es: $\dfrac{20\cdot(7-3)}{2} + \dfrac{83^5 - [(2+1)^3 - 7]\cdot(8/3)}{15(7-3)(8+2)-3}$

20, 7, 3, −, ·, 2, /, 83, 5, ^, 2, 1, +, 3, ^, 7, −, 8, ·, 3, /, −,

15, 7, 3, −, ·, 8, 2, +, ·, 3, −, /, +.

le espressioni in notazione post-fissa sono calcolate usando il modello della PILA:

- S: inserisco i valori da SX a DX
- Appena si trova un operatore, si fa il POP degli ultimi due elementi:

inserite, e si esegue l'operazione e si restituisce il risultato nella cima.

Esempio $(3+4)*2 \Leftrightarrow 3, 4, +, 2, *$

SIMBOLO	PILA	AZIONI
INIZIO	$\emptyset$	/
3	3	PUSH 3
4	3, 4	PUSH 4
+	$\emptyset$	POP 4, POP 3
		CALCOLA $7 = 3 + 4$
	7	PUSH 7
2	7, 2	PUSH 2
×	$\emptyset$	POP 2, POP 7
		CALCOLA $14 = 7 \times 2$
	14	PUSH 14

Mentre la pila permette di fare una elaborazione occupando poco spazio. Nel caso in cui la pila è lunga al massimo 2.

OPERAZIONI SU PILE

1. Crea(P): crea una pila P vuota
2. Vuota(P): restituisce .TRUE. se la pila è vuota
3. Piena(P): restituisce .True. se la pila è piena
4. Pop(P, x): restituisce .False. se la pila è vuota. altrimenti x assume il valore che si trova in cima e il termine in cima viene eliminato.
5. Push(P, x): inserisce x in cima alla pila. Se la pila è piena restituisce .False. altrimenti .True.

Le pile possono essere realizzate mediante:

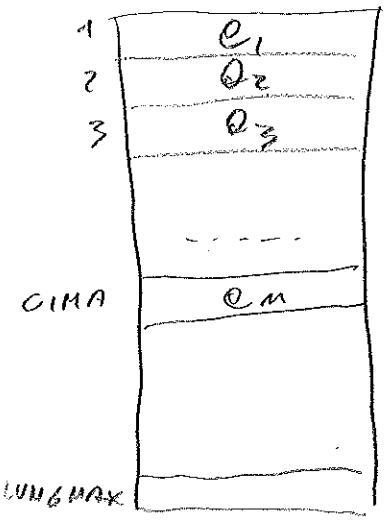
1 - ARRAY
2 - LISTE CONCATENATE

Pile con ARRAY

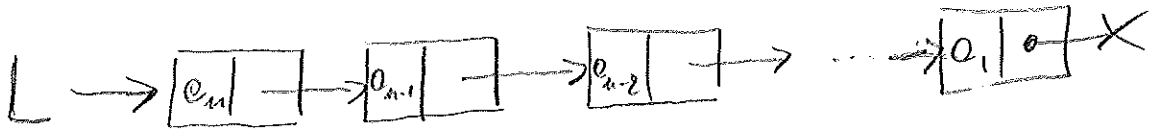
```
TYPE PILA
    INTEGER :: LUNGMAX
    INTEGER :: CIMA
    CHARACTER(LEN=10), ALLOCATABLE :: NOME(:)
END TYPE

FUNCTION CREA_PILA(LMAX) RESULT(N_PILA)
IMPLICIT NONE
INTEGER, INTENT(IN) :: LMAX
TYPE(PILA) :: N_PILA

N_PILA%LUNGMAX = LMAX
N_PILA%CIMA = 0
ALLOCATE(N_PILA%NOME(LMAX))
END FUNCTION CREA_PILA
```



PILE con LISTE CONCATENATE




```
TYPE NODO-PILA
    CHARACTER (LEN=10) :: NOME
    TYPE(NODO_PILA), POINTER :: PROSSIMO
END TYPE

TYPE PILA
    TYPE (NODO_PILA), POINTER :: PT_CIMA
END TYPE

FUNCTION CREA_PILA() RESULT (N_PILA)
IMPLICIT NONE
TYPE(PILA) :: N_PILA
ALLOCATE (N_PILA% PT_CIMA)
N_PILA% PT_CIMA% PROSSIMO => NULL()
END FUNCTION CREA_PILA
```

Poi ci sono funzioni analoghe a:

INSERISCI_IN_TESTA
ELIMINA_IN_TESTA

che sono state usate per le liste.

CODE

È un altro ADT basato sempre sul modello delle liste, in cui gli elementi vengono inseriti da un estremo ed estratti dall'altro.
Sinonimo: LISTA FIFO, FIRST IN FIRST OUT
È un modello più "ONESTO" delle PILE.
Esempio: code in autostrada.

OPERAZIONI SU CODE

1. CREA (C)
2. FUORICODA (x, C)
3. INCODA (x, C)
4. VUOTA (C)
5. PIENA (C)

Si può realizzare una CODA con una lista concatenata usando le funzioni:

INSERISCI_IN_CODA / ELIMINA_DA_TESTA

oppure

INSERISCI_IN_TESTA / ELIMINA_DA_CODA