algoritmi e strutture di dati

tipi astratti di dato

(pile, code, liste implementate tramite array)

m.patrignani

nota di copyright

sommario

- tipi astratti di dato
- strutture di dati elementari
  - le pile
  - le code
  - le liste
- implementazione con array di queste strutture

tipo astratto di dato

- un *tipo astratto di dato* (o ADT, abstract data type) è una descrizione di un tipo di dato indipendente dalla sua realizzazione in un linguaggio di programmazione
- un tipo astratto di dato è costituito da:
  - i domini interessati
    - tra cui il dominio di interesse ed eventualmente altri domini di supporto
  - un insieme di costanti
  - una collezione di operazioni

esempio di tipo astratto di dato

- insieme di interi
  - domini
    - il dominio di interesse è l'insieme I degli insiemi di interi
    - dominio di supporto: gli interi Z = {0, 1, -1, 2, -1, …}
    - dominio di supporto: i booleani B = {true, false}
  - costanti
    - l'insieme vuoto {} è una costante del dominio I
  - operazioni
    - verifica se un insieme è vuoto
      $$\text{IS-EMPTY: } I \rightarrow B$$
    - aggiunta di un elemento ad un insieme
      $$\text{ADD: } I \times Z \rightarrow I$$
    - verifica se un elemento appartiene all'insieme
      $$\text{SEARCH: } I \times Z \rightarrow B$$
    - ecc.

il tipo astratto di dato pila

- le pile (o stack) realizzano una strategia LIFO (last-in first-out)
- tipo astratto: pila di interi
  - domini
    - il dominio di interesse è l'insieme delle pile P di interi
    - dominio di supporto: gli interi Z = {0, 1, -1, 2, -1, …}
    - dominio di supporto: i booleani B = {true, false}
  - costanti
    - la pila vuota

il tipo astratto di dato pila

- tipo astratto: pila di interi
  - operazioni
    - verifica se una pila è vuota

      IS-EMPTY: $P \rightarrow B$
    - inserimento di un elemento nella pila

      PUSH: $P \times Z \rightarrow P$
    - rimozione e restituzione dell'elemento affiorante della pila

      POP: $P \rightarrow P \times Z$
  - operazioni aggiuntive
    - lettura dell'elemento affiorante della pila

      TOP: $P \rightarrow Z$
    - svuotamento della pila

      EMPTY: $P \rightarrow P$
    - numero degli elementi nella pila

      SIZE: $P \rightarrow Z$

realizzazione di un tipo astratto di dato

- un tipo astratto di dato può essere *realizzato* (o *implementato*) in uno specifico linguaggio di programmazione tramite la definizione di
  - tipi concreti o strutture dati nello specifico linguaggio di programmazione corrispondenti ai *domini* necessari
  - costrutti che consentono di codificare le *costanti*
  - funzioni che realizzano le *operazioni* previste
- ovviamente uno stesso tipo astratto di dato può avere diverse realizzazioni con diverse proprità

realizzazione di un ADT: osservazioni

- la definizione dei tipi concreti o delle strutture che corrispondono ai domini può comportare delle limitazioni
  - esempi di limitazioni
    - il dominio P delle pile viene ristretto alle pile di dimensione massima `maxsize`, nota a priori
    - il dominio Z degli interi viene realizzato tramite il tipo concreto `int` che ha un valore minimo `MININT` e massimo `MAXINT`
- le costanti vengono spesso codificate tramite delle funzioni che ritornano la struttura corrispondente
  - esempio di costante:
    - la pila vuota viene realizzata tramite

      `CREATE-STACK(maxsize)`

      che ritorna il riferimento ad una pila vuota che potrà contenere al massimo `maxsize` elementi

funzioni che operano su pile

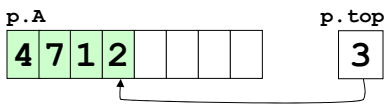
- `CREATE-STACK(maxsize)`
  - ritorna il riferimento ad una pila vuota che potrà contenere al massimo `maxsize` elementi
- `IS-EMPTY(p)`
  - ritorna `true` se la pila è vuota, `false` altrimenti
- `PUSH(p,x)`
  - inserimento di un elemento nella pila
    - può dare un errore di "overflow" se l'implementazione prevede un numero massimo di elementi nella pila
- `POP(p)`
  - rimozione e restituzione dell'elemento affiorante della pila
    - dà un errore di "underflow" se la pila è vuota

altre funzioni su pile

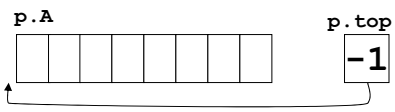
- `TOP(p)`
  - ritorna l'elemento affiorante senza rimuoverlo
    - può dare errore se la pila è vuota
- `EMPTY(p)`
  - svuota la pila
- `SIZE(p)`
  - ritorna il numero degli elementi in pila

realizzazione del dominio pila

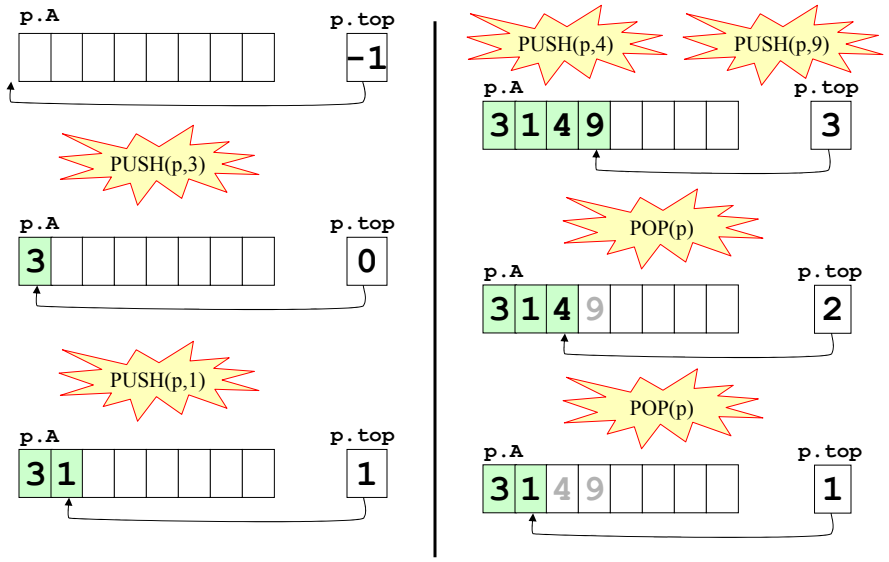
- una pila `p` può essere implementata con un oggetto contenente un array `A` e un attributo `P.top` che contiene l'indice dell'elemento affiorante



- quando la pila è vuota `p.top` vale -1



sequenza di operazioni su una pila



operazione CREATE-STACK

- implementazione in pseudocodice della funzione di creazione

```
CREATE-STACK()
1. ▷ creo un oggetto p con:   p.A array di 100 interi
2. ▷                          p.top intero
3. p.top = -1
4. return p
```


operazioni PUSH e POP

- implementazione in pseudocodice delle funzioni di modifica

```
PUSH(p,x)
1.  if p.top == p.A.length-1
2.      then   error("overflow")
3.      else   p.top = p.top + 1
4.             p.A[p.top] = x
```

```
POP(p)
1.  if p.top == -1
2.      then   error("underflow")
3.      else   p.top = p.top - 1
4.             return p.A[p.top + 1]
```


altre operazioni sulle pile

```
IS-EMPTY(p)
1.  return p.top == -1          ▷ true se la pila è vuota
```

```
EMPTY(p)
1.  p.top = -1                  ▷ vuoto la pila
```

```
TOP(p)
1.  return p.A[p.top]           ▷ l'elemento affiorante
```

```
SIZE(p)
1.  return p.top + 1            ▷ il numero di elementi
```

- il tempo di esecuzione di ogni operazione è $\Theta(1)$

uso delle realizzazioni di un ADT

- dopo aver realizzato un ADT possiamo mettere l'implementazione a disposizione dei programmatori specificando
  - le eventuali limitazioni della realizzazione
  - l'elenco delle operazioni supportate e la loro complessità asintotica
- esempio
  - si dispone di una realizzazione di una pila con le seguenti funzioni e complessità nel caso peggiore
    - `CREATE-STACK(maxsize)` $\Theta(1)$
    - `IS-EMPTY(p)` $\Theta(1)$
    - `PUSH(p,x)` $\Theta(1)$
    - `POP(p)` $\Theta(1)$
    - `EMPTY(p)` $\Theta(1)$

il tipo astratto di dato coda

- le code (o queue) realizzano una strategia FIFO (first-in first-out)
- tipo astratto: coda di interi
  - domini
    - il dominio di interesse è l'insieme delle code Q di interi
    - dominio di supporto: gli interi Z = {0, 1, -1, 2, -1, …}
    - dominio di supporto: i booleani B = {true, false}
  - costanti
    - la coda vuota

il tipo astratto di dato coda

- tipo astratto: coda di interi
  - operazioni
    - verifica se una coda è vuota

      IS-EMPTY: $Q \rightarrow B$
    - inserimento di un elemento nella coda

      ENQUEUE: $Q \times Z \rightarrow Q$
    - rimozione e restituzione dell'elemento più vecchio della coda

      DEQUEUE: $Q \rightarrow Q \times Z$
  - operazioni aggiuntive
    - lettura dell'elemento più vecchio della coda

      FRONT: $Q \rightarrow Z$
    - svuotamento della coda

      EMPTY: $Q \rightarrow Q$
    - numero degli elementi nella coda

      SIZE: $Q \rightarrow Z$

realizzazione dell'ADT coda

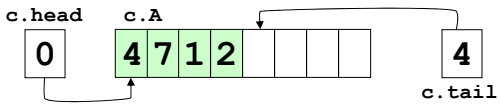
- `CREATE-QUEUE(maxsize)`
  - ritorna un riferimento ad una coda vuota che può contenere al massimo `maxsize` interi
- `IS-EMPTY(c)`
  - ritorna `true` se la coda è vuota, altrimenti ritorna `false`
- `ENQUEUE(c,x)`
  - inserisce un elemento `x` nella coda
    - può dare un errore di "overflow" se l'implementazione prevede un numero massimo di elementi
- `DEQUEUE(c)`
  - rimuove l'elemento più vecchio della coda e lo restituisce
    - dà un errore di "underflow" se la coda è vuota

altre operazioni su code

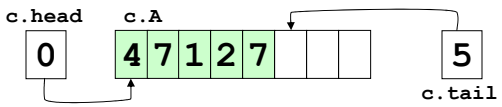
- `FRONT(c)`
  - ritorna l'elemento più vecchio senza rimuoverlo
    - può dare errore se la coda è vuota
- `EMPTY(c)`
  - svuota la coda
- `SIZE(c)`
  - ritorna il numero degli elementi in coda

implementazione di una coda

- una coda `c` può essere realizzata con un array `c.A` arricchito da due attributi `c.head` e `c.tail` che contengono gli indici dell'elemento più vecchio e della prima posizione utile

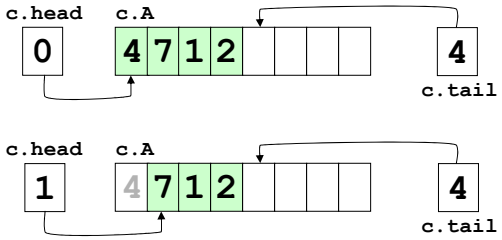


- un nuovo elemento viene aggiunto da `ENQUEUE` nella posizione `c.tail` (che viene incrementato)

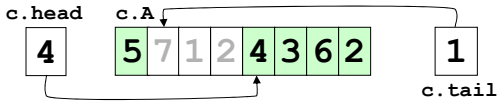


implementazione di una coda

- l'elemento ritornato da `DEQUEUE` è quello in posizione `c.head` (che viene incrementato)

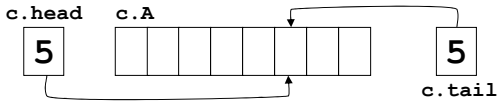


- l'array è gestito come una lista circolare

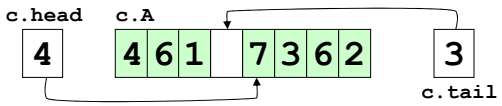


implementazione di una coda

- la coda è vuota quando `c.head` e `c.tail` puntano alla stessa casella



- la coda non può avere più di n-1 elementi
  - se avesse n elementi, che valore potremmo dare a `c.tail` senza creare equivoco con la coda vuota?



operazioni ENQUEUE e DEQUEUE

```
ENQUEUE(c,x)
1.  if c.head==c.tail+1 or (c.tail==c.A.length-1 and c.head==0)
2.      error("overflow")
3.  else   c.A[c.tail] = x
4.         if c.tail == c.A.length-1
5.             c.tail = 0
6.         else    c.tail = c.tail + 1
```

```
DEQUEUE(c)
1.   if c.head == c.tail
2.      error("underflow")
3.  else   x = c.A[c.head]
4.         if c.head == c.A.length-1
5.             c.head = 0
6.         else    c.head = c.head + 1
7.         return x
```

altre operazioni sulle code

```
IS-EMPTY(c)
1.  return c.head == c.tail
```

```
EMPTY(c)
1.  c.head = c.tail = 0
```

```
FRONT(c)
1.  if c.head == c.tail
2.      then   error("empty queue")
3.      else   return c.A[c.head]
```

- il tempo di esecuzione di ogni operazione è $\Theta(1)$

esercizi su pile e code

1. scrivi lo pseudocodice della procedura `CREATE-QUEUE()` che restituisce il riferimento ad una coda vuota
2. scrivi lo pseudocodice della procedura `SIZE(c)` che restituisce il numero di elementi in una coda
3. scrivi lo pseudocodice che implementa una struttura dati in cui si possa inserire/rimuovere elementi sia in testa che in coda
   - deve avere contemporaneamente `PUSH`, `POP`, `DEQUEUE`, `ENQUEUE` (dove evidentemente `ENQUEUE` è uguale a `PUSH`)

esercizi su pile e code

5. descrivi come sia possibile implementare un coda disponendo solo della realizzazione di una pila
   - ti occorreranno due pile `p1` e `p2` da usare simultaneamente
   - le operazioni `ENQUEUE` e `DEQUEUE` si tradurranno in opportune operazioni di `PUSH` e `POP` sulle due pile `p1` e `p2`
   - discuti la complessità delle operazioni `ENQUEUE` e `DEQUEUE`
6. descrivi come sia possibile implementare una pila disponendo solo della realizzazione di una coda
   - discuti la complessità delle operazioni `PUSH` e `POP`
7. scrivi lo pseudocodice della procedura `CONTAINS(p,x)` che ritorna `true` se l'elemento `x` è contenuto nella pila `p` e ritorna `false` se `x` non è contenuto (lasciando la pila invariata)

liste

- le liste sono strutture di dati in cui gli oggetti sono disposti in una sequenza lineare
  - occorre poter scorrere la lista con un iteratore, cioè un indice di posizione
- tipo astratto: lista di interi
  - domini
    - il dominio di interesse è l'insieme delle liste L di interi
    - dominio di supporto: gli iteratori I che identificano le posizioni
    - dominio di supporto: gli interi Z = {0, 1, -1, 2, -1, …}
    - dominio di supporto: i booleani B = {true, false}
  - costanti
    - la lista vuota

operazioni dell'ADT lista

- operazioni sulle liste di interi
  - ritorna l'iteratore del 1° elemento: HEAD: $L \rightarrow I$
  - ritorna l'iteratore dell'ultimo: LAST: $L \rightarrow I$
  - ritorna l'iteratore dell'elemento seguente: NEXT: $L \times I \rightarrow I$
  - ritorna l'iteratore dell' elemento precedente: PREV: $L \times I \rightarrow I$
  - ritorna l'intero nella posizione specificata: INFO: $L \times I \rightarrow Z$
  - verifica se una lista è vuota: IS-EMPTY: $L \rightarrow B$
  - inserisce in testa alla lista: INSERT: $L \times Z \rightarrow L$
  - inserisce prima di una posizione: INSERT-BEFORE: $L \times I \times Z \rightarrow L$
  - inserisce in coda: ADD: $L \times Z \rightarrow L$
  - inserisce dopo una posizione: ADD-AFTER: $L \times I \times Z \rightarrow L$
  - cerca la posizione di un elemento: SEARCH: $L \times Z \rightarrow I$
  - elimina un elemento: DELETE: $L \times I \rightarrow L$
  - cerca ed elimina un elemento: DELETE: $L \times Z \rightarrow L$
  - svuota la lista: EMPTY: $L \rightarrow L$
  - numero degli elementi: SIZE: $L \rightarrow Z$

realizzazione di una lista

- creazione

  `CREATE-LIST(maxsize)`
  - ritorna il riferimento ad una lista vuota che può contenere al massimo `maxsize` elementi
- operazioni di aggiornamento

  `INSERT(l,x)`
  - inserisce l'elemento `x` in testa alla lista `l`

  `INSERT-BEFORE(l,x,i)`
  - inserisce l'elemento `x` prima della posizione i

  `ADD(l,x)`
  - aggiunge `x` in coda alla lista `l`

  `ADD-AFTER(l,x,i)`
  - aggiunge l'elemento `x` dopo la posizione i

  `DELETE(l,i)`
  - rimuove l'elemento in posizione i dalla lista `l`

  `EMPTY(l)`
  - vuota la lista

operazioni possibili sulle liste

- operazioni di consultazione

  `NEXT(l,i)`
  - ritorna l'iteratore dell'elemento che segue `i` nella lista
  - dà errore (o ritorna un iteratore invalido) se `i` è l'ultimo elemento

  `PREV(l,i)`
  - ritorna l'interatore dell'elemento che precede `i` nella lista
  - dà errore (o ritorna un iteratore invalido) se `i` è il primo elemento

  `HEAD(l)`
  - ritorna l'iteratore del primo elemento della lista

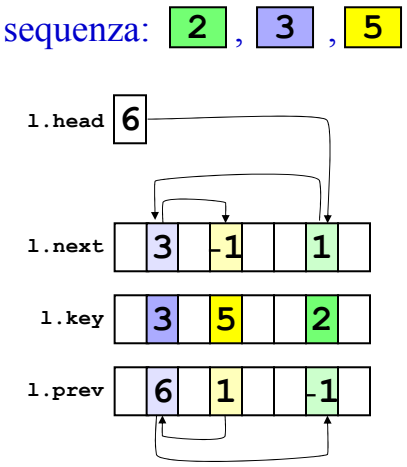
  `LAST(l)`
  - ritorna l'iteratore dell'ultimo elemento della lista

  `SEARCH(l,k)`
  - ritorna l'iteratore dell'elemento con chiave `k` nella lista `l`

  `IS-EMPTY(l)`
  - ritorna `true` se la lista è vuota, altrimenti ritorna `false`

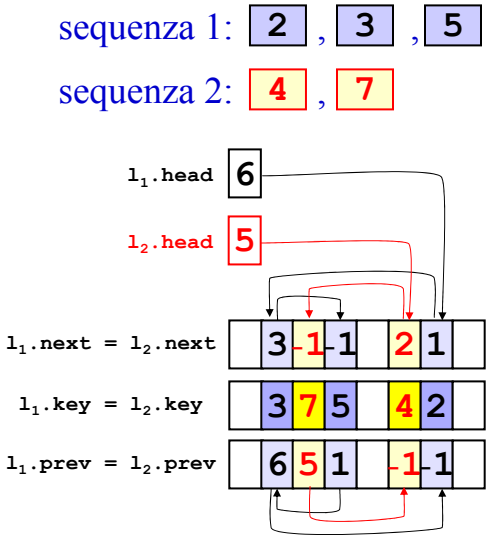
realizzazione di una lista con un array

- un elemento x_i della lista corrisponde ad un indice i dei tre array
- `l.next`[i] contiene l'indice dell'elemento che segue x_i
- `l.prev`[i] contiene l'indice dell'elemento che precede x_i
- se `l.next`[i] o `l.prev`[i] è -1, x_i è l'ultimo o il primo elemento della lista



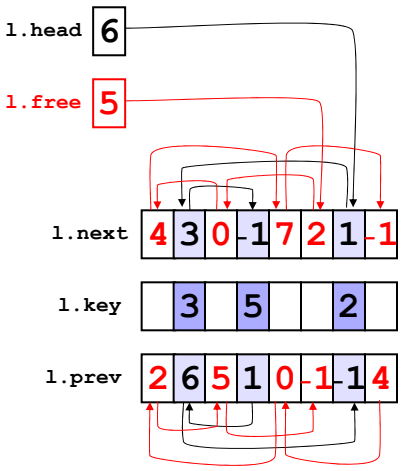
più liste con gli stessi array

- gli array `l.next`, `l.key` ed `l.prev` possono essere condivisi da due o più liste
  - le liste non interferiscono, perché utilizzano posizioni diverse degli array



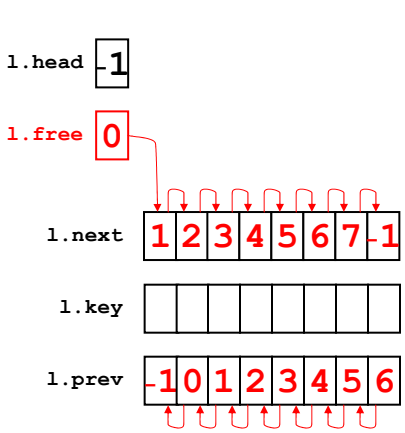
uso della lista libera

- per inserire un nuovo elemento in lista occorre sapere quali posizioni degli array sono ancora libere
- tutte le posizioni libere possono essere memorizzate in una seconda lista `l.free`
  - un inserimento di un elemento in `l` è un trasferimento di una posizione dalla lista `l.free` alla lista `l.head`
  - una cancellazione da `l` è un trasferimento di una posizione dalla lista `l.head` alla lista `l.free`

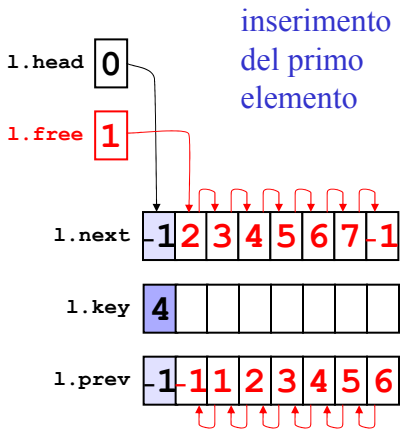


configurazione iniziale: lista vuota

- quando la lista `l` è vuota tutte le posizioni sono assegnate alla lista `l.free`



inserimento del primo elemento



creazione di una lista vuota di interi

- procedura per inizializzare una lista vuota di interi (supponendo al massimo 100 posizioni)

```
CREATE-LIST()
1.  ▷ creo un nuovo oggetto l con:
2.  ▷      l.next, l.key, l.prev array di 100 interi
3.  ▷      l.head, l.free interi
4.  for i = 0 to 99
5.      l.next[i] = i+1
6.      l.prev[i] = i-1
7.  l.next[99] = -1
8.  l.head = -1
9.  l.free = 0
10. return l
```


gestione della lista `l.free`

- procedura di servizio per ottenere una posizione libera dalla lista `l.free`

```
ALLOCATE-COLUMN(l)
1.  if l.free == -1
2.      error("overflow")
3.  else
4.      i = l.free      ▷ i è l'indice della nuova posizione
5.      l.free = l.next[l.free]
6.      if l.free != -1
7.              l.prev[l.free] = -1
8.  return i
```

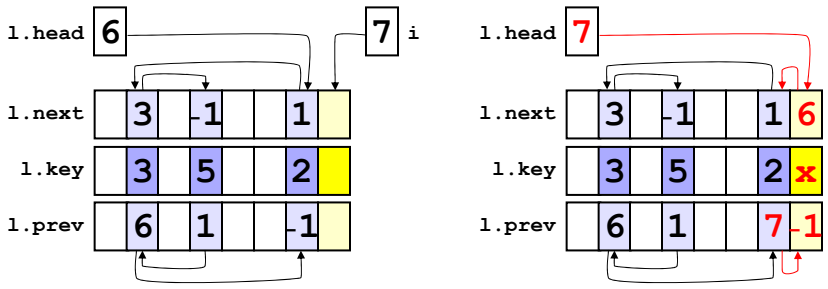

gestione della lista `l.free`

- procedura di servizio per restituire una posizione alla lista `l.free`

```
FREE-COLUMN(l,i)
1.  l.prev[i] = -1
2.  l.next[i] = l.free
3.  if l.free != -1
4.       l.prev[l.free] = i
5.  l.free = i
```


codice di `INSERT(L,x)`

```
INSERT(l,x)          ▷ x è il valore da aggiungere in lista
1.  i = ALLOCATE-COLUMN(l)      ▷ indice di una nuova colonna libera
2.  l.key[i] = x
3.  l.prev[i] = -1              ▷ i diventa primo elemento
4.  l.next[i] = l.head          ▷ il resto della lista segue i
5.  if l.head != -1
6.       l.prev[l.head] = i
7.  l.head = i
```



esercizi: liste implementate con array

1. scrivi lo pseudocodice della procedura `DELETE(l,i)` che rimuove da `l` l'elemento in posizione `i`
2. scrivi lo pseudocodice della procedura `SEARCH(l,k)` che restituisce la posizione del primo elemento di `l` con valore della chiave `k`
3. scrivi lo pseudocodice della procedura `SIZE(l)` che conta gli elementi della lista `l`