Algoritmi e Strutture di Dati

Capitolo 4 - Strutture di dati elementari

Sequenza

- **Struttura di dati**
  - **Dinamica e lineare**
  - **Contenente elementi generici (Item), potenzialmente anche duplicati**
  - **Ordine all'interno della sequenza è importante**
- **Interfaccia**
  - **E' possibile aggiungere / togliere elementi, specificando la posizione**
    - $s = s_1, s_2, \ldots, s_n$
    - **L'elemento s_i è in posizione pos_i**
    - **Esistono le posizioni (fittizie) pos_0, pos_{n+1}**
  - **E' possibile accedere *direttamente* ad alcuni elementi (testa / coda)**
  - **E' possibile accedere *sequenzialmente* a tutti gli altri elementi**

Insiemi (Set)

- **Struttura dati "generale": *insieme dinamico***
  - **Può crescere, contrarsi, cambiare contenuto**
  - **Operazioni base: inserimento, cancellazione, verifica contenimento**
  - **Il tipo di insieme (= struttura) dipende dalle operazioni**
- **Elementi**
  - **Elemento: oggetto "puntato" da un riferimento/puntatore**
  - **Composto da:**
    - **campo chiave di identificazione**
    - **dati satellite**
    - **campi che fanno riferimento ad altri elementi dell'insieme**

Dizionari (Dictionary o Map)

- **Il dizionario rappresenta il concetto matematico di relazione univoca**
  - **Relazione $R : D \rightarrow C$**
  - **Insieme *D* è il dominio (elementi detti chiavi)**
  - **Insieme *C* è il codominio (elementi detti valori)**
  - **Associazione chiave-valore**
- **Operazioni ammesse:**
  - **ottenere il valore associato ad una particolare chiave (se presente), o nil**
  - **inserire una nuova associazione chiave- valore, cancellando eventuali associazioni precedenti;**
  - **rimuovere un'associazione chiave-valore esistente**

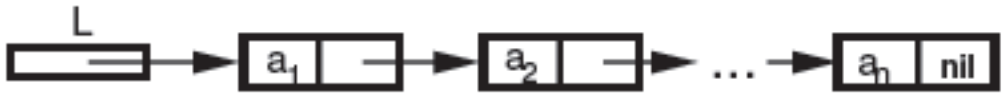
Realizzazione di strutture dati

- **Realizzazione di alcuni tipi di dato:**
  - **Sequenza ↔ lista**
  - **Dizionario ↔ lista**
  - **Insieme ↔ lista**
- **Realizzazione alternativa**
  - **Vettori**
  - **Alberi Bilanciati**
  - **Tabelle Hash**
- **La scelta della struttura di dati ha riflessi sull'efficienza e sulle operazioni ammesse**

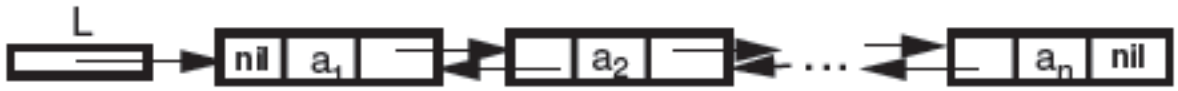
Liste

- **Liste (List, Linked List)**
  - **Una sequenza di nodi (record), contenenti dati arbitrari e 1-2 puntatori all'elemento successivo e/o precedente**
  - **Contiguità nella lista ⇏ contiguità nella memoria**
- **Realizzazione possibili**
  - ***Bidirezionale* / monodirezionale**
  - ***Con sentinella* / senza sentinella**
  - ***Circolare* / non circolare**

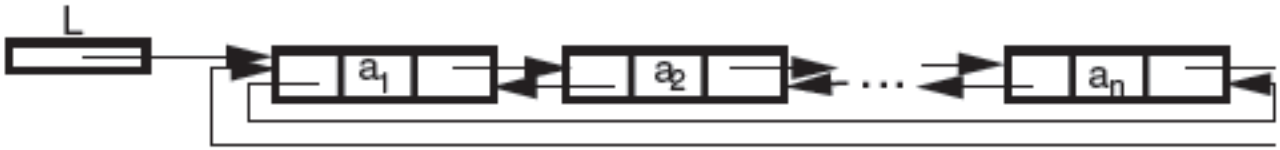
Linked List - esempi



monodirezionale



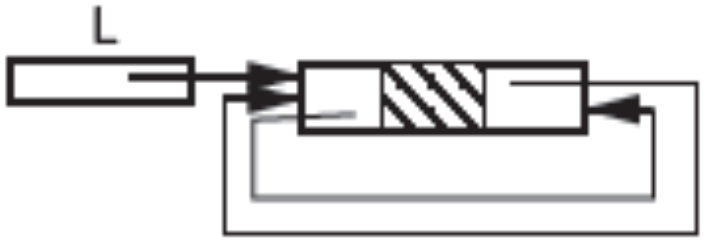
bidirezionale



bidirezionale circolare

monodirezionale con sentinella

Linked List – bidirezionale circolare con sentinella



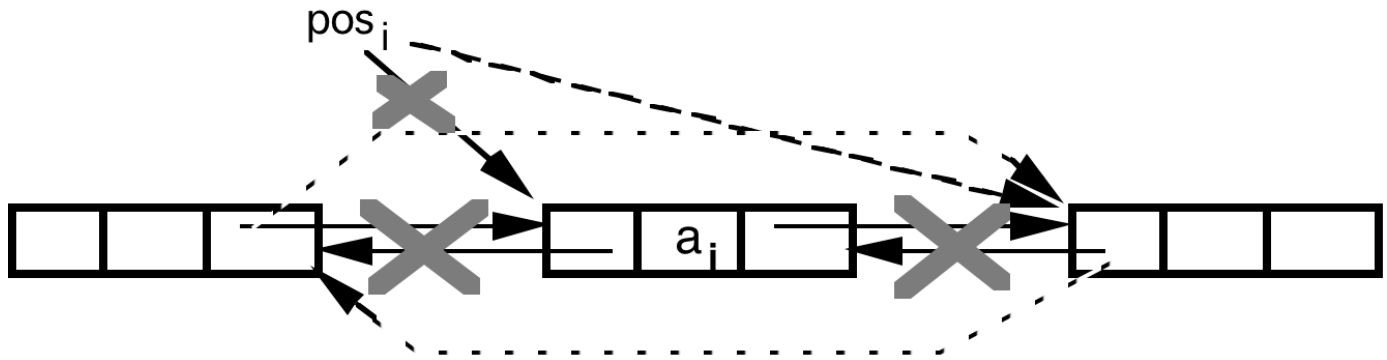
lista vuota

List - circolare, bidirezionale, con sentinella

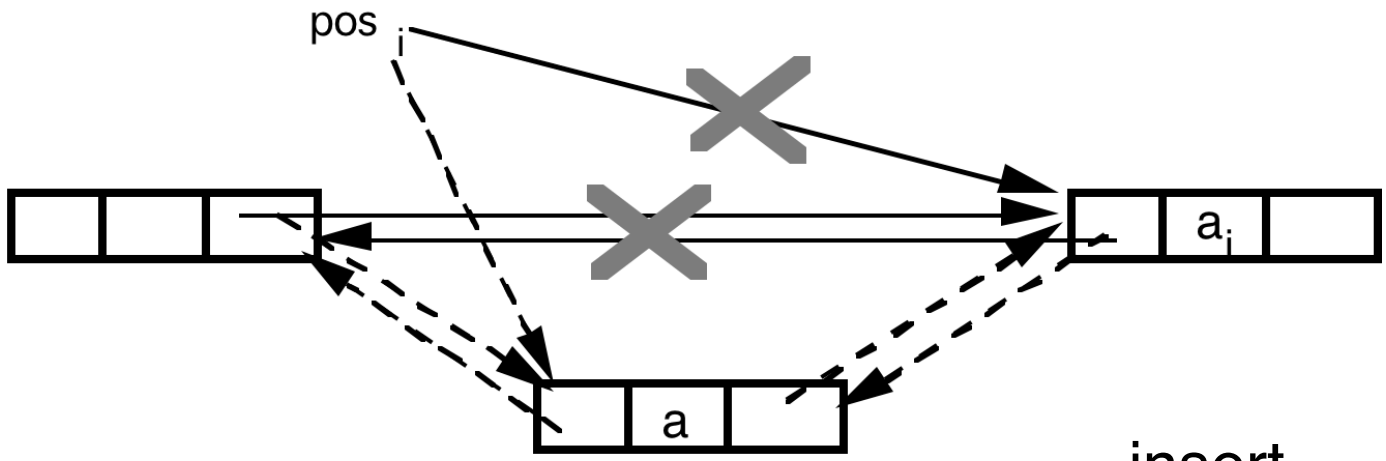
```
LIST
  LIST pred          % Predecessore
  LIST succ          % Successore
  ITEM value         % Elemento

  LIST List()
      LIST t ← new LIST
      t.pred ← t
      t.succ ← t
      return t

  boolean isEmpty()
      return pred = succ = this

  POS head()
      return succ

  POS tail()
      return pred

  POS next(POS p)
      return p.succ

  POS prev(POS p)
      return p.pred

  boolean finished(POS p)
      return (p = this)

  ITEM read(POS p)
      return p.value

  write(POS p, ITEM v)
      p.value ← v

  POS insert(POS p, ITEM v)
      LIST t ← List()
      t.value ← v
      t.pred ← p.pred
      p.pred.succ ← t
      t.succ ← p
      p.pred ← t
      return t;

  POS remove(POS p)
      p.pred.succ ← p.succ
      p.succ.pred ← p.pred
      LIST t ← p.succ
      delete p
      return t;
```

Nota:
LIST e POS sono tipi equivalenti

List – cancellazione e inserimento



remove



insert

Esempio List – Il catalogo di Don Giovanni

Cancellare dal *catalogo* tutte le duchesse e copiare in un'altra lista *L* tutte le spagnole:

```
donGiovanni(LIST catalogo)
   LIST L ← List()
   POS q ← catalogo.head()
   while not catalogo.finished(q) do
   |  ITEM v ← catalogo.read(q)
   |  if v.nazione = spagnola then L.insert(L.next(L.tail()), v)
   |  if v.rango = duchessa then
   |   |  q ← catalogo.remove(q)
   |  else
   |   |  q ← catalogo.next(q)
```

Complessità: vedremo in seguito

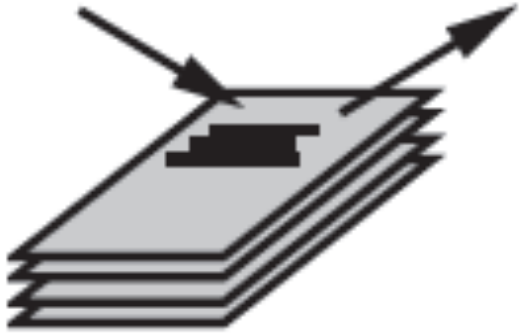
Liste - Realizzazione con vettori

- E' possibile realizzare una lista con vettori
  - Posizione ≡ indice nel vettore
  - Le operazioni **insert()** e **remove()** cosa fanno?
  - Vedremo successivamente il costo delle operazioni
- Problema
  - Spesso non si conosce a priori quanta memoria serve per memorizzare la lista
  - Se ne alloca una certa quantità, per poi accorgersi che non è sufficiente.
- Soluzione
  - Si alloca un vettore di dimensione maggiore, si ricopia il contenuto del vecchio vettore nel nuovo e si rilascia il vecchio vettore
  - Esempi: `java.util.Vector`, `java.util.ArrayList`

Stack

Una pila (stack)

- **è un insieme dinamico in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato: "*quello che per meno tempo è rimasto nell'insieme*"**
- **politica "*last in, first out*" (LIFO)**



pratiche burocratiche

Operazioni previste

STACK

% Restituisce **true** se la pila è vuota

boolean isEmpty()

% Inserisce *v* in cima alla pila

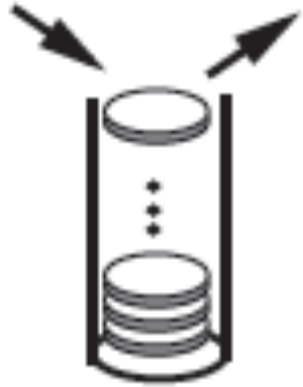
push (ITEM *v*)

% Estrae l'elemento in cima alla pila e lo restituisce al chiamante

ITEM pop()

% Legge l'elemento in cima alla pila

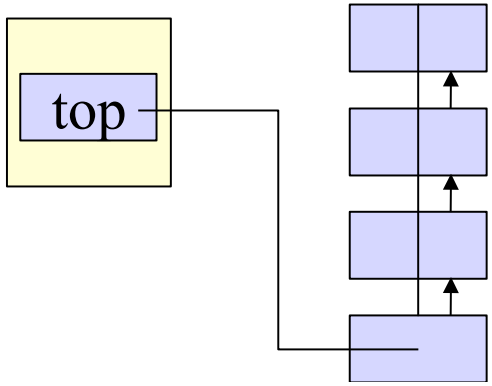
ITEM top()



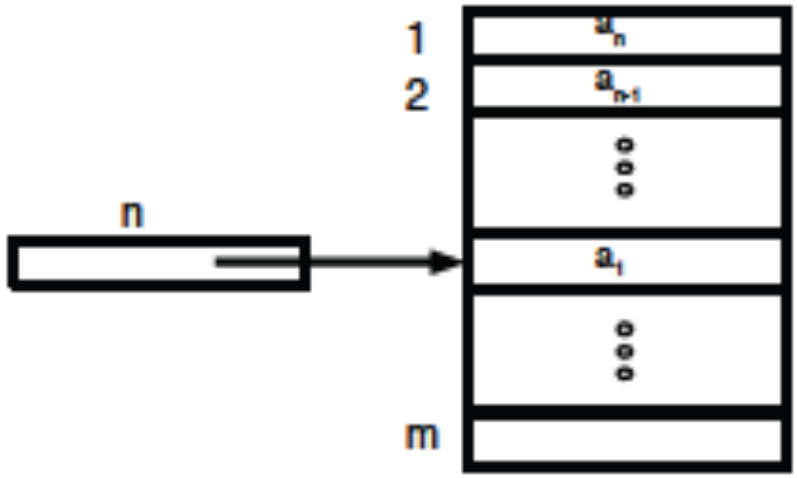
tubetto di pastiglie

Stack

- **Possibili utilizzi**
  - **Nei linguaggi con procedure: gestione dei record di attivazione**
  - **Nei linguaggi stack-oriented:**
    - **Le operazioni elementari prendono uno-due operandi dallo stack e inseriscono il risultato nello stack**
    - **Es: Postscript, Java bytecode**



- **Possibili implementazioni**
  - **Tramite *liste bidirezionali***
    - **puntatore all'elemento top**
  - **Tramite *vettore***
    - **dimensione limitata, overhead più basso**



Stack

- **Reverse Polish Notation (PNG), o notazione postfissa**
  - **Espressioni aritmetiche in cui gli operatori seguono gli operandi**
  - **Definita dalla grammatica: <expr> ::= <numeral> | <expr> <expr> <operator>**
  - **Esempi:**
    - **(7 + 3) × 5 si traduce in 7 3 + 5 ×**
    - **7 + (3 × 5) si traduce in 7 3 5 × +**
- **Valutazione PNG, stack based (esempio: Java bytecode)**
  - **push 7 7**
  - **push 3 3 7**
  - **push 5 5 3 7**
  - **op. ×: pop, pop, push 15 7**
  - **op. +: pop, pop, push 22**

Stack - pseudocodice

```
STACK
ITEM[] A                    % Elementi
integer n                   % Cursore
integer m                   % Dim. max

STACK Stack(integer dim)
    STACK t ← new STACK
    t.A ← new integer[1 ... dim]
    t.m ← dim
    t.n ← 0
    return t

ITEM top()
    precondition: n > 0
    return A[n]

boolean isEmpty()
    return n = 0

ITEM pop()
    precondition: n > 0
    ITEM t ← A[n]
    n ← n − 1
    return t

push(ITEM v)
    precondition: n < m
    n ← n + 1
    A[n] ← v
```

Queue

Una coda (queue)

- **è un insieme dinamico in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato: "*quello che per più tempo è rimasto nell'insieme*"**
- **politica "*first in, first out*" (*FIFO*)**

Operazioni previste

QUEUE

% Restituisce **true** se la coda è vuota

boolean isEmpty()

% Inserisce *v* in fondo alla coda

enqueue (ITEM *v*)

% Estrae l'elemento in testa alla coda e lo restituisce al chiamante

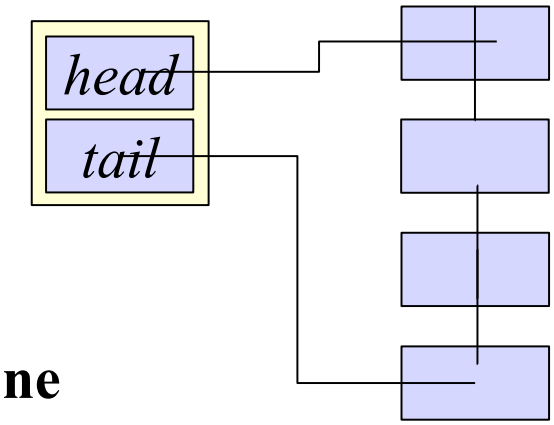
ITEM dequeue()

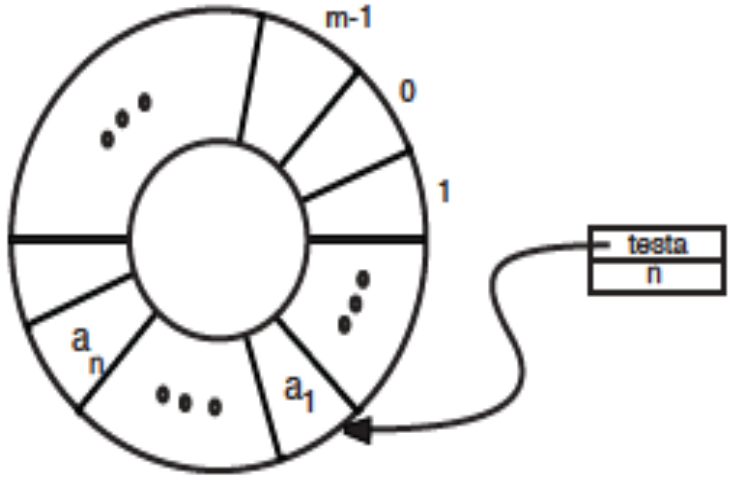
% Legge l'elemento in testa alla coda

ITEM top()

Queue

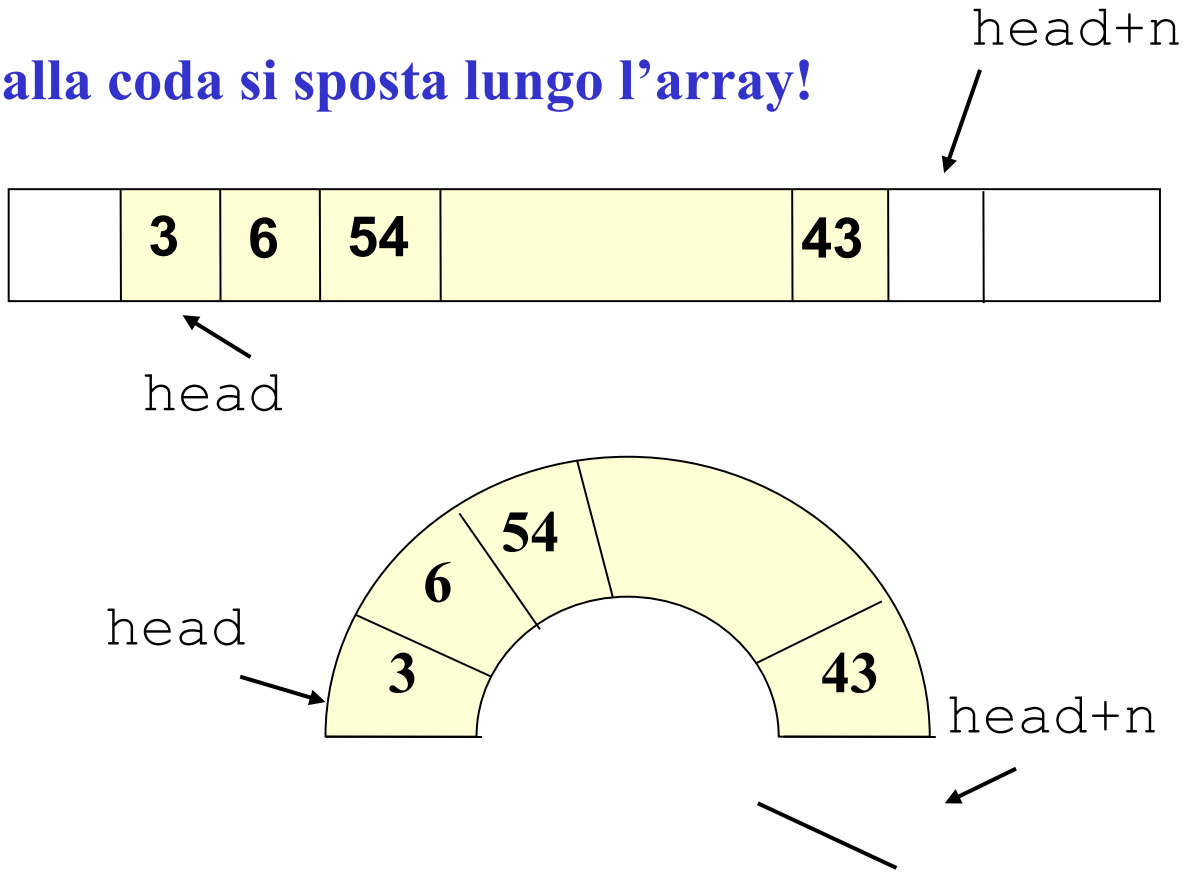
- **Possibili utilizzi**
  - **Nei sistemi operativi, i processi in attesa di utilizzare una risorsa vengono gestiti tramite una coda**
  - **La politica FIFO è fair**
- **Possibili implementazioni**
  - **Tramite liste monodirezionali**
    - **puntatore *head* (inizio della coda), per estrazione**
    - **puntatore *tail* (fine della coda), per inserimento**
  - **Tramite array circolari**
    - **dimensione limitata, overhead più basso**



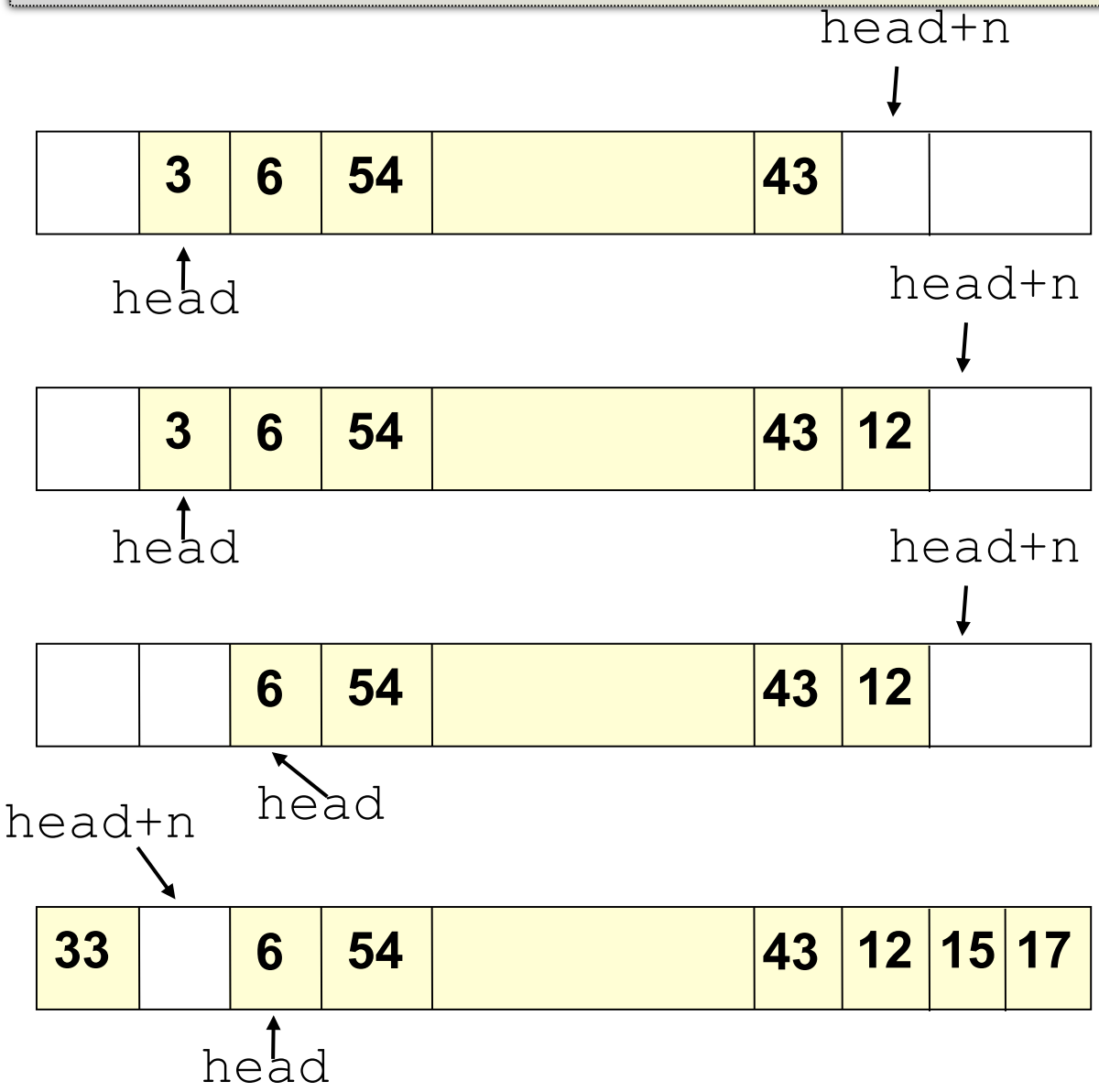


Coda: Realizzazione tramite vettore circolare

- La “finestra” dell’array occupata dalla coda si sposta lungo l’array!
- Dettagli implementativi
  - L'array circolare può essere implementato con un'operazione di modulo
  - Bisogna prestare attenzione ai problemi di overflow (buffer pieno)



Coda: Realizzazione tramite vettore circolare



Coda - pseudocodice

QUEUE

```
ITEM[] A                          % Elementi
integer n                         % Dim. attuale
integer testa                     % Testa
integer m                         % Dim. max

QUEUE Queue(integer dim)
    QUEUE t ← new QUEUE
    t.A ← new integer[0 ... dim − 1]
    t.m ← dim
    t.testa ← 0
    t.n ← 0
    return t

ITEM top()
    precondition: n > 0
    return A[testa]

boolean isEmpty()
    return n = 0

ITEM dequeue()
    precondition: n > 0
    ITEM t ← A[testa]
    testa ← (testa + 1) mod m
    n ← n − 1
    return t

enqueue(ITEM v)
    precondition: n < m
    A[(testa + n) mod m] ← v
    n ← n + 1
```

Esercizi

Modifcare lo pseudicodice relativo a top(), isEmpty(), dequeue() e enqueue() nella struttura Queue nel caso si abbia la seguente definizione di Queue:

Item [] *A* % Elementi
integer *n* % Dim. Attuale
integer *testa* % Testa
integer *m* % Dim. Massima

QUEUE Queue(**integer** *dim*)
    QUEUE *t* ← **new** QUEUE
    t.A ← **new integer** [1… *dim*]
    t.m ← *dim*
    t.testa ← 1
    t.n ← 0
    return *t*

Esercizi

Modifcare lo pseudicodice relativo a top(), isEmpty(), dequeue() e enqueue() nella struttura Queue nel caso si utilizzi un vettore circolare a due indici: uno per la testa e uno per la coda:

```
Item [] A          % Elementi
integer n          % Dim. Attuale
integer testa      % Testa
integer coda       % Coda
integer m          % Dim. Massima

QUEUE Queue(integer dim)
    QUEUE t ← new QUEUE
    t.A ← new integer [0… dim-1]
    t.m ← dim
    t.testa ← 0
    t.coda ← 0
    t.n ← 0
    return t
```