3 Strutture dati astratte

> *Picking the wrong data structure for the job can be disastrous in terms of performance. Identifying the very best data structure is usually not as critical, because there can be several choices that perform similarly.*
>
> Steven S. Skiena, *The Algorithm Design Manual*

3.1 Pila

```
ITEM[] A                                  % elementi
int n                                     % cursore
int m                                     % dimesione massima

// crea una pila vuota
STACK Stack(int dim)
    STACK t = new STACK
    t.A = new int[0...dim-1]
    t.m = dim
    t.n = 0
    ritorna t

// leggi l'elemento in cima alla pila
ITEM top
    precondition: n > 0
    ritorna A[n]

// restituisce vero se la pila è vuota
bool isEmpty
    ritorna n == 0

// estrae l'elemento in cima alla pila e lo restituisce al chiamante
ITEM pop
    precondition: n > 0
    ITEM t = A[n]
    n--
    ritorna t

// inserisce v in cima alla pila
push(ITEM v)
    precondition: n < m
    n++
    A[n] = v
```

3.2 Coda

```
ITEM[] A                                  % elementi
int n                                     % dimensione attuale
int testa                                 % testa
int m                                     % dimesione massima

// crea una cosa vuota
QUEUE Queue(int dim)
    QUEUE t = new QUEUE
    t.A = new int[0...dim-1]
    t.m = dim
    t.testa = 0
    t.n = 0
    ritorna t

// legge l'elemento in testa alla coda
ITEM top
    precondition: n > 0
    ritorna A[testa]

// restituisce vero de la coda è vuota
ITEM isEmpty
    ritorna n == 0

// estrae l'elemento in testa alla coda e lo restituisce al chiamante
ITEM dequeue
    precondition: n > 0
    ITEM t = A[testa]
    testa = (testa + 1) mod m
    n--
    ritorna t

// inserisce v in fondo alla coda
ITEM enqueue
    precondition: n < m
    A[(testa + n) mod m][v]
    n++
```

3.3 Lista

Il costo delle operazioni per questa struttura è $\mathcal{O}(1)$

```
LIST                    % bidirezionale con sentinella
LIST pred               % predecessore
LIST succ               % successore
LIST value              % elemento

LIST List
    // la sentinella fa riferimento a sé stessa
    t.pred = t
    t.succ = t
    ritorna t

POS head
    ritorna succ

POS tail
    ritorna pred

POS next
    ritorna p.succ

POS prev
    ritorna p.pred

bool finished(pos p)
    ritorna p = this

ITEM read(pos p)
    ritorna p.value

write(pos p)
    ritorna p.value

// posso fare queste operazioni essendo sicuro di
   avere sempre un predecessore
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
    ritorna p

POS remove(POS p)
    p.pred.succ = p.succ
    p.succ.pred = p.pred
    LIST t = p.succ
    delete p
    ritorna t
```

3.4 Sequenza

Una struttura dati *dinamica*, *lineare* che rappresenta una sequenza *ordinata* di valori, dove lo stesso valore può comparire più volte.

```
Sequence

// interpretare
bool isEmpty        % vero se la sequenza è vuota
bool finished       % vero se p è uguale a pos_0
 oppure a pos_{n+1}

// leggere
POS head            % posizione del primo elemento
POS tail            % posizione dell'ultimo elemento

// iterare
POS next  % posizione dell'elem. che segue p
POS prev  % posizione dell'elem. che precede p

// modifica

/* inserisce l'elemento di tipo ITEM nella
   posizione p, ritorna la nuova posizione, che
   diviene il predecessore di p                */
POS insert(POS p, ITEM v)

/* rimuove l'elemento contenuto nella posizione p,
   ritorna il successore di p, che diviene il
   predecessore di p                           */
POS remove(pos p)

/* legge l'elemento di tipo ITEM contenuto nella
   posizione p                                 */
read(POS p)

/* scrive l'elemento v di tipo ITEM nella
   posizione p                                 */
write(POS p, ITEM v)
```