ADT - Abstract Data Type

E' un tipo di dato astratto che prescinde dalla sua concreta implementazione.
Per ogni **ADT** vengono definite:

- **collezioni di dati:** a partire da quali dati si costruisce una struttura del nuovo tipo
- **operazioni:** cosa **devono**(ma non **come**) fare le operazioni definite sul nuovo tipo
- **complessità:** eventuali vincoli di complessità sulle operazioni definite

Si parla invece di **implementazione concreta** di un **ADT** quando abbiamo:

- **struttura dati** con cui memorizzare la collezione di dati
- collezione di **procedure** con cui realizzare le operazioni definite

Pile(Stack)

In una pila i dati vengono estratti in ordine inverso rispetto a quello in cui sono stati inseriti. Usa dunque un meccanismo **LIFO** (**Last In First Out**), ovvero l'ultimo ad entrare è il primo ad uscire

Operazioni

- **Push: inserisce** un elemento in **cima** alla pila
- **Pop: estrae** un elemento dalla **cima** della pila
- **Top: restituisce** l'elemento in **cima**
- **Empty:** restituisce **true** se la pila è vuota, **false** altrimenti
- **Size:** restituisce la **dimensione** corrente della pila

Collezione di dati

Qualunque elemento di tipo **T** di dati

Assiomi Stack

SIZE(S), EMPTY(S) e PUSH(S, t) sono sempre definiti
POP(S) e TOP(S) sono definiti se e solo se EMPTY(S) restituisce falso
EMPTY(S), SIZE(S) e TOP(S) non modificano la pila S
EMPTY(S) restituisce vero se e solo se SIZE(S) restituisce 0
la sequenza PUSH(S, t);POP(S) restituisce t e non modifica la pila S
la sequenza PUSH(S, t);TOP(S) restituisce t
PUSH(S, t) incrementa SIZE(S) di 1
POP(S) decrementa SIZE(S) di 1

Implementazione concreta: array

Usiamo un array statico di **M** celle per definire una implementazione concreta dell'ADT **pila**.

Usando il meccanismo **LIFO**:

- gli elemento presenti nella pila occupano sempre le prime posizioni degli array
- quando ci sono **N** elementi, il prossimo elemento da estrarre è in posizione **N**

Ciò aggiunge un ulteriore assioma:

- **Push** è definito soltanto se **Size(Stack)** < **M**.

```
PUSH(S, t)
  if S.N ≠ S.M then
      S.N ← S.N + 1
      S[N] ← t
  else
       error overflow
SIZE(S)
  return S.N
EMPTY(S)
  if S.N == 0 then
      return true
  return false
```

```
TOP(S)
  if S.N == 0 then
       error underflow
  else
      return S[S.N]
POP(S)
  if S.N == 0 then
       error underflow
  else
      S.N ← S.N − 1
      return S[S.N + 1]
```

Implementazioni concrete delle operazioni dell'ADT Pila

Una volta implementato l'**ADT**, il programmatore può interagire con l'array(o pila) solo tramite le specifiche dell'ADT.

Infatti l'implementazione concreta e la struttura dati:

- sono **nascosti** dietro un'**interfaccia**
- possono essere **modificate** senza fare modifiche a programmi che usano l'ADT

Implementazione concreta: lista

Usiamo una lista per realizzare la pila.
In questo caso conviene utilizzare una **lista con sentinella**.
Teniamo inoltre il conto del numero di elementi presenti nella lista.

```
PUSH(S, t)
  S.N ← S.N + 1
  t.next ← S.sen.next
  S.sen.next ← t
SIZE(S)
  return S.N
EMPTY(S)
  if S.N == 0 then
      return true
  return false
```

```
TOP(S)
  if S.N == 0 then
      error underflow
  else
      return S.sen.next
POP(S)
  if S.N == 0 then
      error underflow
  else
      S.N ← S.N − 1
      t ← S.sen.next
      S.sen.next ← S.sen.next.next
      return t
```

S.N è il numero di elementi

Confronto tra implementazioni

Complessità delle operazioni: Tutte $O(1)$
Complessità spaziale: $O(M)$ per l'array e $O(N)$ per la lista. Tuttavia la lista introduce un overhead dovuto ai puntatori

Per l'array inoltre bisogna definire a priori un numero massimo di elementi, per la lista no.

Utilizzi

- Chiamate ricorsive di funzioni
- Visita in profondità di grafi
- Valutazione di espressione in notazione postfissa

Code (queue)

In una coda i dati vengono estratti nell'ordine in cui sono stati inseriti.
Usa dunque un meccanismo **FIFO** (**First In First Out**) ovvero il primo ad entrare è il primo ad uscire.

Operazioni

- **Enqueue:** inserisce un elemento nella coda
- **Dequeue:** estrae un elemento dalla coda
- **Front:** restituisce il primo elemento dalla coda
- **Empty:** restituisce **true** se la coda è vuota, **false** altrimenti
- **Size:** restituisce la **dimensione** corrente della coda

Collezione di dati

Qualunque elemento di tipo **T** di dati

Assiomi queue

SIZE(Q), EMPTY(Q) e ENQUEUE(Q, t) sono sempre definiti
DEQUEUE(Q) e FRONT(Q) sono definiti se e solo se EMPTY(Q) restituisce falso
EMPTY(Q), SIZE(Q) e FRONT(Q) non modificano la coda Q
EMPTY(Q) restituisce vero se e solo se SIZE(Q) restituisce 0
se SIZE(Q)$= N$ e viene effettuata ENQUEUE(Q, t), allora dopo N esecuzione di DEQUEUE(Q) abbiamo FRONT(Q)$= t$
se FRONT(Q)$= t$ allora DEQUEUE(Q) estrae t dalla coda
ENQUEUE(Q, t) incrementa SIZE(Q) di 1
DEQUEUE(Q) decrementa SIZE(Q) di 1

Implementazione concreta: array

Usiamo un array statico di **M** celle per definire una coda.
In questo non conviene tenere gli elementi nelle prime posizioni dell'array.

Se eseguiamo $Dequeue(Coda)$ e l'elemento da estrarre è nella prima posizione, dovremmo spostare tutti gli altri elementi, e ciò avrebbe un costo lineare.
Stesso concetto si applica alla funzione $Queue$

Usiamo dunque l'array in maniera **circolare** tenendo conto di dove si trova l'inizio(head) e la fine(tail) della coda.

Dunque, data una coda Q, $Q.head$ indica la posizione da dove estrarre l'elemento successivo.
Analogamente, $Q.tail$, indica la posizione dove inserire l'elemento successivo.

Quindi per controllare che la **coda** sia vuota, basta controllare che $Q.head = Q.tail$

```
SIZE(Q)
  if Q.tail ≥ Q.head then
      return Q.tail − Q.head
  return Q.M − (Q.head − Q.tail)
EMPTY(Q)
  if Q.tail == Q.head then
      return true
  return false
NEXTCELL(Q, c)
  if c ≠ Q.M then
      return c + 1
  return 1
DEQUEUE(Q)
  if SIZE(Q)== 0 then
      errorunderflow
  else
      t ← Q[Q.head]
      Q.head ← NEXTCELL(Q,Q.head)
      return t
```

```
ENQUEUE(Q, t)
  if SIZE(Q) ≠ Q.M − 1 then
      Q[Q.tail] ← t
      Q.tail ← NEXTCELL(Q,Q.tail)
  else
      erroroverflow
FRONT(Q)
  if SIZE(Q)== 0 then
      errorunderflow
  else
      return Q[Q.head]
```

Implementazione concreta: lista

Usiamo una lista semplice per rappresentare la coda.
Aggiungiamo un puntatore all'ultimo elemento della coda.

Per controllare che la coda sia vuota, basta vedere se $Q.head = nil$

```
ENQUEUE(Q, t)
  if Q.N == 0 then
      Q.head ← t
      Q.tail ← t
  else
      Q.tail.next ← t
      Q.tail ← t
  Q.N ← Q.N + 1
SIZE(Q)
  return Q.N
EMPTY(Q)
  if Q.N == 0 then
      return true
  return false
```

```
FRONT(Q)
  if Q.N == 0 then
      error underflow
  else
      return Q.head
DEQUEUE(Q)
  if Q.N == 0 then
      error underflow
  else
      t ← Q.head
      Q.head ← Q.head.next
      Q.N ← Q.N − 1
      return t
```

Confronto tra implementazioni

Complessità delle operazioni: Tutte $O(1)$
Complessità spaziale: $O(M)$ per l'array e $O(N)$ per la lista. Tuttavia la lista introduce un overhead dovuto ai puntatori

Per l'array inoltre bisogna definire a priori un numero massimo di elementi, per la lista no.

Utilizzi

- Buffer
- Visita in ampiezza di grafi
- Simulazione