SOMMARIO

- Pila (stack): una struttura dati lineare a cui si può accedere soltanto mediante uno dei suoi capi per memorizzare e per estrarre dati.
  - Specifica (*descrizione*).
  - Implementazione (*realizzazione*).
- Coda (queue): una linea d'attesa che cresce aggiungendo elementi in fondo e si accorcia rimuovendo elementi dall'inizio.
  - Specifica.
  - Implementazione.

STACK

- Una pila è una sequenza $<a_1,...,a_n>$ di elementi dello stesso tipo, di cui solo l'ultimo elemento inserito, a_n, è visibile all'utente e la modalità di accesso è "ultimo elemento inserito, primo elemento rimosso" (LIFO, *last in ,first out*).
- Una pila può essere descritta in termini di operazioni che ne modificano lo stato o che ne verificano lo stato: pertanto è il primo ADT che vediamo.

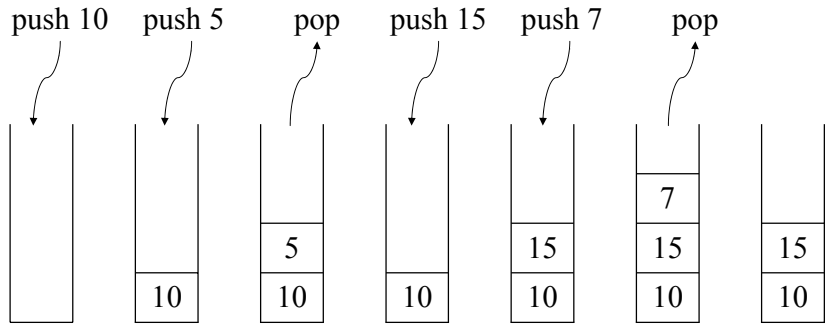
STACK: specifica

- Una possibile interfaccia è la seguente

```
public interface Stack {
  Object push(Object x);//inserisce l'oggetto x in
                                  cima alla pila
  Object pop(); //Rimuove e restituisce l'oggetto
                             in cima alla pila
  Object peek();//Restituisce l'oggetto in cima
                        alla pila senza rimuoverlo
  boolean isEmpty(); //Verifica se la pila è vuota
  void clear();  //Svuota la pila
}
```


STACK

- Vediamo graficamente alcune operazioni su una pila



STACK: esempio 1

- Un'applicazione delle pile è l'identificazione dei delimitatori corrispondenti in un'espressione, che è un esempio significativo perché questa attività fa parte di qualsiasi compilatore.
- Vediamo un esempio di
  - Uso corretto dei delimitatori

    `b+(c-d)*(e-f)`
  - Mancata corrispondenza

    `b+(c-d)*(e-f))`

STACK: esempio 1

- L'algoritmo legge un carattere dalla stringa che rappresenta l'espressione e se si tratta di un delimitatore iniziale, '(' o '[' o '{', lo memorizza in una pila. Quando viene trovato un delimitatore finale, ')' o ']' o '}', esso viene confrontato con quello estratto dalla pila: se corrispondono l'analisi continua, altrimenti termina segnalando un errore.

- Vediamo una possibile implementazione

STACK: esempio 1

```
public static boolean BalBra(Character[] expr){
   Stack st = new VectorStack();
   for (int i=0; i<expr.length;i++){
      char ch = expr[i].charValue();
      switch(ch){
        case '(': case '[': case '{':
           st.push(expr[i]);  break;
        case ')': case ']': case '}':
           try{
                char ctr = ((Character)st.pop()).charValue();
                if (!(ctr=='(' && ch==')' || ctr=='[' && ch==']'
                                          || ctr=='{' && ch=='}'))
                        return false;
                } catch(ArrayIndexOutOfBoundsException e){
                       return false;
                }  break;
        default:
        }}
   if (!st.isEmpty())  return false;
   else  return true;
}
```


STACK: esempio 1

Vediamo l'analisi di alcune espressioni

```
2 + [ ( 3 - 2 ) * ( 5 - a ) + b ]

Stack:          Letto: 2        +[(3-2)*(5-a)+b]
Stack:          Letto: +        [(3-2)*(5-a)+b]
Stack:          Letto: [        (3-2)*(5-a)+b]
Stack: [        Letto: (        3-2)*(5-a)+b]
Stack: (        Letto: 3        -2)*(5-a)+b]
Stack: (        Letto: -        2)*(5-a)+b]
Stack: (        Letto: 2        )*(5-a)+b]
Stack: (        Letto: )        *(5-a)+b]
Stack: [        Letto: *        (5-a)+b]
Stack: [        Letto: (        5-a)+b]
Stack: (        Letto: 5        -a)+b]
Stack: (        Letto: -        a)+b]
Stack: (        Letto: a        )+b]
Stack: (        Letto: )        +b]
Stack: [        Letto: +        b]
Stack: [        Letto: b        ]
Stack: [        Letto: ]

                                OK
```

STACK: esempio 1

```
2 + [ ( 3 - 2 ) ) * ( 5 - a ) + b ]

Stack:          Letto: 2        +[(3-2))*(5-a)+b]
Stack:          Letto: +        [(3-2))*(5-a)+b]
Stack:          Letto: [        (3-2))*(5-a)+b]
Stack: [        Letto: (        3-2))*(5-a)+b]
Stack: (        Letto: 3        -2))*(5-a)+b]
Stack: (        Letto: -        2))*(5-a)+b]
Stack: (        Letto: 2        ))*(5-a)+b]
Stack: (        Letto: )        )*(5-a)+b]
Stack: [        Letto: )        *(5-a)+b]

                                NO
```


STACK: esempio 1

```
2 + [ ( 3 - 2 ) * ( 5 - a ) + b ] }

Stack:          Letto: 2        +[(3-2)*(5-a)+b]}
Stack:          Letto: +        [(3-2)*(5-a)+b]}
Stack:          Letto: [        (3-2)*(5-a)+b]}
Stack: [        Letto: (        3-2)*(5-a)+b]}
Stack: (        Letto: 3        -2)*(5-a)+b]}
Stack: (        Letto: -        2)*(5-a)+b]}
Stack: (        Letto: 2        )*(5-a)+b]}
Stack: (        Letto: )        *(5-a)+b]}
Stack: [        Letto: *        (5-a)+b]}
Stack: [        Letto: (        5-a)+b]}
Stack: (        Letto: 5        -a)+b]}
Stack: (        Letto: -        a)+b]}
Stack: (        Letto: a        )+b]}
Stack: (        Letto: )        +b]}
Stack: [        Letto: +        b]}
Stack: [        Letto: b        ]}
Stack: [        Letto: ]        }
Stack:          Letto: }

                                NO
```


STACK: esempio 2

- Come altro esempio di applicazione delle pile si considera l'addizione di numeri molto grandi, cioè superiori al massimo numero rappresentabile.

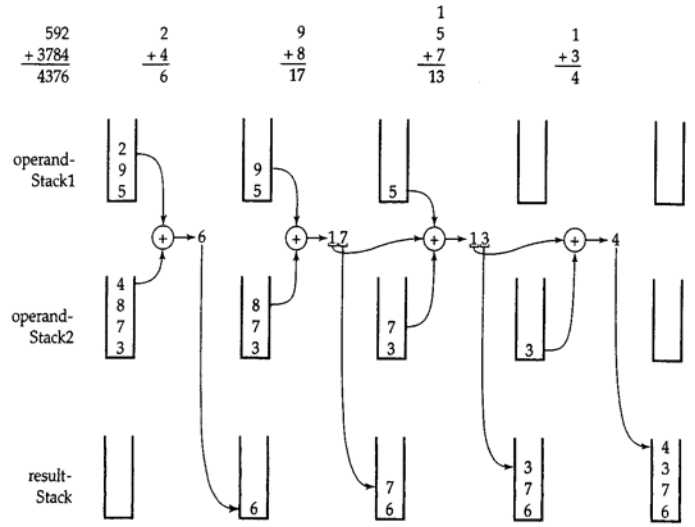
- Una possibile soluzione consiste nel trattare tali numeri come stringhe di cifre numeriche, memorizzando i valori corrispondenti a queste cifre su due pile, poi eseguire l'addizione dei singoli valori estratti.

STACK: esempio 2

Vediamo un possibile algoritmo:

- *Leggere le cifre del primo numero e inserirle nel primo stack;*
- *Leggere le cifre del secondo numero e inserirle nel secondo stack;*
- *Inizializzare a zero la variabile temp;*
- *Finché c'è una pila non vuota*
  - *Estrarre un numero da ogni pila non vuota e sommarlo a temp;*
  - *Inserire nello stack del risultato la cifra delle unità della somma;*
  - *Memorizzare in temp il riporto;*
- *Inserire l'ultimo riporto nella pila del risultato, se non è zero;*
- *Estrarre i numeri dalla pila del risultato e visualizzarli.*

STACK: esempio 2



STACK: esempio 3

- Interessante notare, nel contesto di questo argomento, che la Java Virtual Machine è basata sul concetto di pila.
- Ogni programma in esecuzione ha una pila, il *run-tile stack* (si veda la ricorsione), che contiene le informazioni (*activation record*) di tutti i metodi attivi in quel momento.

STACK: ADT

- Il fatto notevole, a questo punto, è che abbiamo sviluppato degli interessanti algoritmi basati su pile senza conoscere l'implementazione della pila!
- Questo è possibile perchè la pila è definita come *tipo di dato astratto*, un tipo di dato specificato mediante le operazioni possibili su di esso.
- Il contratto espresso dall'interfaccia ha permesso di sviluppare un programma basato sulle pile solo utilizzando le informazioni relative al loro comportamento, non alla loro realizzazione.

STACK: implementazione

- Si consideri ora la realizzazione della pila. Si sono usate le operazioni `push()` e `pop()`, come fossero disponibili, ma devono essere realizzate come metodi che operano sui dati della pila.
- Una semplice realizzazione consiste nell'usare un *array flessibile*, cioè la classe `Vector` di Java. Questa classe rappresenta un array che può *modificare dinamicamente* le sue dimensioni e fornisce molti metodi per manipolare tale array.

STACK: implementazione

```
import java.util.*;
public class VectorStack implements Stack {
  Vector data;
   public VectorStack(){
       data = new Vector();}
   public Object push(Object x) {
       data.add(x);
       return x;}
   public Object pop() {
       return data.remove(data.size()-1);}
   public Object peek() {
       return data.lastElement();}
   public boolean isEmpty() {
       return data.isEmpty();}
   public void clear() {
       data.clear();}
}
```

eccezione

STACK: prestazioni

- Le operazioni `push()` e `pop()` vengono eseguite a tempo costante *O(1)*, essendo realizzate con un vettore.
- L'inserimento di un elemento in una *pila piena* richiede l'assegnazione di maggiore memoria e gli elementi del vettore pieno sono copiati in quello nuovo. Quindi inserire elementi nel caso peggiore richiede un tempo *O(n)*. Tuttavia si può pensare di avere ancora tempo costante in media.

QUEUE

- Una coda è una sequenza $<a_1,...,a_n>$ di elementi dello stesso tipo, della quale si distinguono due elementi, il primo e l'ultimo inserito (a_1 e a_n, denominati la testa e il fondo della coda). La modalità di accesso è "primo elemento inserito, primo elemento rimosso" (FIFO: *first in ,first out*).

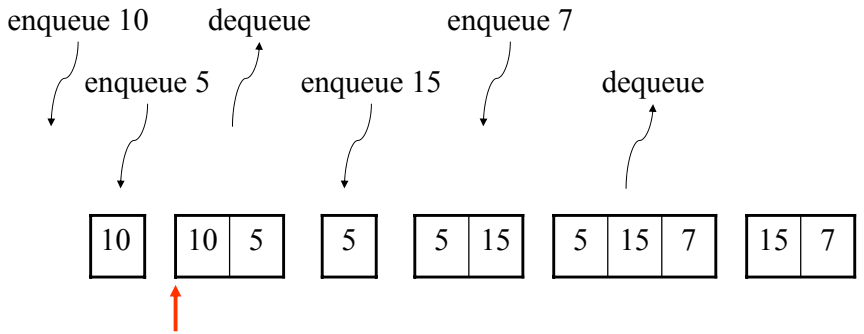
QUEUE : specifica

- Una possibile interfaccia è la seguente

```
public interface Queue {
  boolean isFull();//Verifica se la coda è piena
  boolean isEmpty();//Verifica se la coda è vuota
  void enqueue(Object el);//Inserisce l'elemento el
                                  in fondo alla coda
  Object dequeue();//Estrae il primo elemento della
                                                coda
  Object front();//Restituisce il primo elemento
                    della coda senza estrarlo
  void clear();//Svuota la coda
}
```

QUEUE

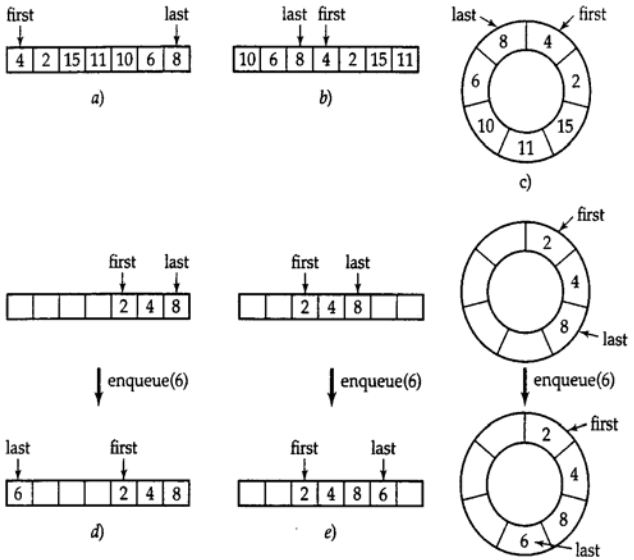
- Vediamo graficamente alcune operazioni su una coda



QUEUE: array circolare

- Una possibile realizzazione di una coda usa un array. Gli elementi vengono aggiunti alla fine della coda e estratti dall'inizio, pertanto possono esserci *celle libere all'inizio* dell'array. Per non essere sprecate devono essere usate per accodare nuovi elementi. Quindi la fine della coda può trovarsi all'inizio dell'array. Tale situazione è illustrata dall'*array circolare*.
- Vediamo una rappresentazione grafica

QUEUE: array circolare



QUEUE: array circolare

- Tuttavia si opera su array non circolari, quindi i metodi `enqueue()` e `dequeue()` devono considerare la possibilità di "girare" intorno all'array, cioè di riusare le celle libere, quando si aggiunge o rimuove un elemento.

- Vediamo una possibile implementazione.

QUEUE : implementazione

```
public class ArrayQueue implements Queue {
   int first, last, size;
   Object[] data;

   public ArrayQueue(int n){
        size=n;
        data = new Object[size];
        first=last=-1;
   }
   public boolean isFull() {
        return first==0 && last==size-1 || first==last+1;
   }
   public boolean isEmpty() {
        return first==-1;
   }
   public Object front() { return data[first];}
   public void clear() { last=first=-1;}}
    ...
```


QUEUE : implementazione

```
   ...
   public void enqueue(Object el) {
        if (last==size-1 || last==-1){
               data[0]=el;
               last=0;
               if (first ==-1)
                      first=0;
        }else
               data[++last]=el;
   }
   public Object dequeue() {
        Object tmp = data[first];
        if (first==last)
               last=first=-1;
        else if (first == size-1)
               first=0;
        else
               first++;
        return tmp;
}
```


QUEUE : prestazioni

- Le operazioni di accodamento ed estrazione vengono eseguite a tempo costante *O(1)*, essendo realizzate con un array.

- Non è stato considerato il problema di aumentare la dimensione della coda in caso risultasse piena. Dovrebbe essere gestito da metodi opportuni definiti nella classe `ArrayQueue`. Con considerazioni analoghe a quelle per le pile.

QUEUE : prestazioni

- Per semplificare l'implementazione si può pensare di non realizzare un array circolare: ad ogni `dequeue()` tutti gli elementi presenti vengono fatti "traslare" di una posizione indietro in modo da avere sempre la posizione iniziale nella prima cella dell'array.
- Tuttavia tale soluzione porta ad avere un costo lineare *O(n)* nel caso medio per il metodo `dequeue()`.