SOMMARIO

- Pila (stack):
  - Specifica: interfaccia.
  - Algoritmi basati su stack.
  - Implementazione:
    - Strutture indicizzate (*array*): array di dimensione variabile.
    - Strutture collegate (*nodi*).
  - Prestazioni.

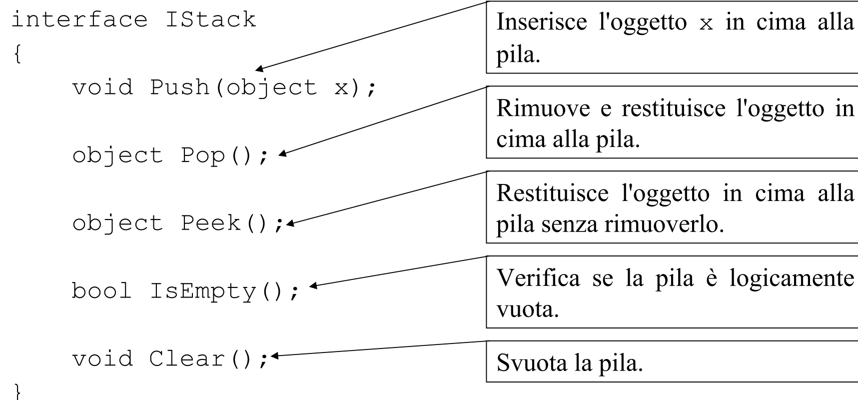
STACK

- *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).*

- È una struttura dati lineare a cui si può accedere soltanto mediante uno dei suoi capi per memorizzare e per estrarre dati.
- Una pila può essere descritta in termini di operazioni che ne modificano lo stato o che ne verificano lo stato: pertanto è un ADT.

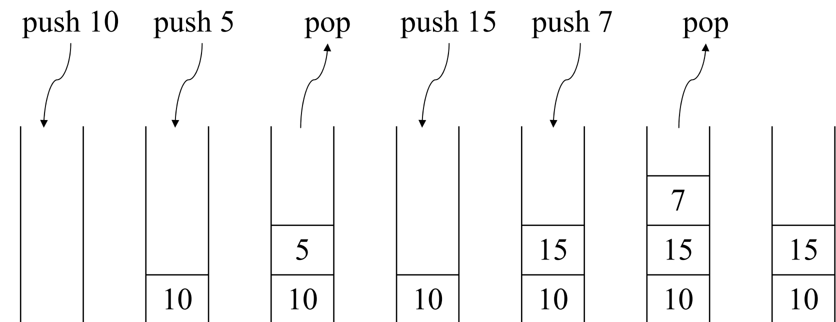
STACK: specifica

- Una possibile (e minima) interfaccia è la seguente:



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:
  - 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 bool BalBra(char[] expr){
    IStack st = new ArrayStack(256);
    for (int i = 0; i < expr.Length; i++){
        char ch = expr[i];
        switch (ch)
        {
            case '(': case '[':case '{':
                st.Push(expr[i]); break;
            case ')':case ']':case '}':
                try{
                    char ctr = (char)st.Pop();
                    if (!(ctr == '(' && ch == ')' || ctr == '[' &&
                              ch == ']'|| ctr == '{' && ch == '}'))
                        return false;
                }
                catch (Exception e){
                    return false;
                } break;
        }
    }
    if (!st.IsEmpty()) return false;
    else return true;}
```

L'oggetto di tipo `ArrayStack` è assegnato al tipo `IStack`.

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.

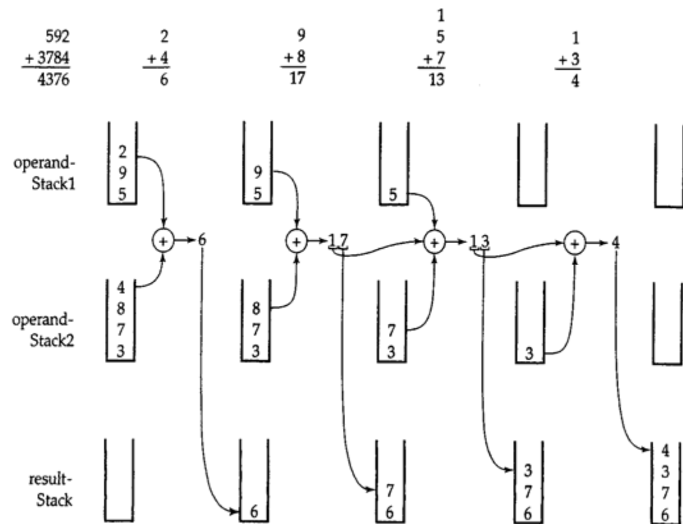
- Una possibile soluzione consiste nel trattare tali numeri come stringhe di cifre numeriche, memorizzando i valori corrispondenti a queste cifre su due pile, poi eseguire l'addizione dei singoli valori estratti.

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 chiamata ad un metodo è basata sul concetto di pila.

- Ogni programma in esecuzione ha una pila, il *run-time stack* (si veda la ricorsione), che contiene le informazioni (*activation record*) di tutti i metodi invocati in quel momento.

STACK: ADT

- Il fatto notevole, a questo punto, è che abbiamo sviluppato degli interessanti algoritmi basati su pile <u>senza conoscere l'implementazione della pila</u>!

- 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 espresse dall'interfaccia, ma devono essere implementate, cioè realizzate come metodi che operano sui dati della pila.
- L'implementazione può essere realizzata sfruttando due diversi tipi di strutture dati: strutture indicizzate (*array*) o strutture collegate (*nodi*).

- L'implementazione con array consiste nell'usare un *array flessibile*, cioè un array che può *modificare dinamicamente* le sue *dimensioni*, perchè, in generale, non è noto a priori il numero di oggetti che una pila deve contenere.

STACK: implementazione

- Si realizza una struttura dati complessa utilizzandone una primitiva, l'array.
- Il pregio di tale implementazione è il *basso costo computazionale* per inserimenti ed estrazioni, *O(1)*, mentre il punto critico riguarda l'*uso della memoria*.
- Una sequenza di `Push()` può riempire la pila, mentre una sequenza di `Pop()` può lasciare un numero elevato di posizioni vuote, quindi nasce la necessità di ridimensionare opportunamente l'array.

STACK: implementazione

- Nota sull'implementazione di un *array flessibile*:
  - Allocare un nuovo array per ogni elemento da inserire ha un costo *O(n)*: è necessario copiare tutti i valori dal vecchio array al nuovo. Quindi ogni operazione di inserimento risulta troppo onerosa, *O(n)*. La stessa cosa vale per ridurre la dimensione dell'array.
  - È necessario utilizzare un accorgimento, la tecnica del *raddoppio* e del *dimezzamento*: il costo del ridimensionato dell'array può essere distribuito su più operazioni ottenendo un *costo medio* costante, *O(1)*.

STACK: implementazione con array

```
class ArrayStack: IStack
{
    int top;
    int isize;
    object[] data;

    public ArrayStack(int dim){
        isize = dim;
        data = new object[dim];
        top = -1;
    }

    public void Push(object x){
        if (top == data.Length - 1)
            RaddoppiaArray();
        data[++top] = x;
    }
```

I dati della pila sono contenuti in un array di `object`.

Metodo privato che implementa il concetto di array flessibile.

STACK: implementazione con array

```
public object Peek()
{
    if (IsEmpty())
        throw new Exception("Pila vuota.");
    return data[top];
}

public bool IsEmpty()
{
    return top==-1;
}

public void Clear()
{
    top=-1; ;
}
```

STACK: implementazione con array

```
public object Pop()
{
    if (IsEmpty())
        throw new Exception("Pila vuota.");
    if ( (top<data.Length/4) && (top>=isize/4) )
        DimezzaArray();
    return data[top--];
}
```

Metodo privato che implementa il concetto di array flessibile.

- Il controllo su `top<data.Length/4` serve per evitare di invocare un dimezzamento della dimensione dell'array non appena si è eseguito un raddoppio della sua dimensione.
- Il controllo su `top>=isize/4` assicura di non scendere mai sotto la dimensione iniziale della pila.

STACK: implementazione con array

```
    private void RaddoppiaArray()
    {
        object[] tmp = new object[2*data.Length];
        for (int i = 0; i < data.Length; i++)
            tmp[i] = data[i];
        data = tmp;
    }

    private void DimezzaArray()
    {
        object[] tmp = new object[data.Length/2];
        for (int i = 0; i <= top; i++)
            tmp[i] = data[i];
        data = tmp;
    }
}
```


STACK: prestazioni

- Le operazioni `Push()` e `Pop()` vengono eseguite a costo costante *O(1)*, essendo realizzate con un array.
- 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 costo *O(n)*. Tuttavia si può pensare di avere ancora *costo* costante in *media*: tale operazione avviene in media dopo *n* inserimenti, quindi il *costo distribuito* per ogni operazione è *O(1)*. Lo stesso vale per l'estrazione di un elemento.

STACK: implementazione con nodi

- Una implementazione alternativa consiste nell'uso di *strutture collegate*: i dati sono memorizzati in nodi, ogni nodo ha un riferimento al precedente.
- In questo caso non vi sono problemi di memoria: si alloca solo il numero necessario di nodi. Ogni operazione ha costo *O(1)*.
- Ogni inserimento provoca la creazione di un oggetto nodo. Ogni estrazione rilascia un oggetto nodo.

STACK: implementazione con nodi

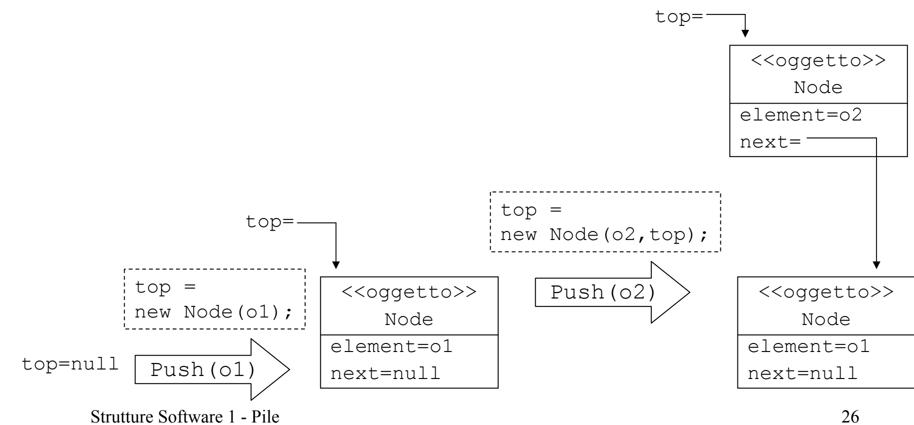
```
class NodeStack: IStack
{
    Node top;

    public NodeStack()
    {
        top = null;
    }

    public void Push(object x)
    {
        if (top == null)
            top = new Node(x);
        else
            top=new Node(x,top);
    }
```


STACK: implementazione con nodi

- Una possibile rappresentazione grafica delle operazioni di inserimento di due oggetti in una pila vuota.



STACK: implementazione con nodi

```
  public object Pop(){
      if (IsEmpty())
          throw new Exception("Pila vuota.");
      object tmp = top.element;
      top = top.next;
      return tmp;
  }

  public object Peek(){
      if (IsEmpty())
          throw new Exception("Pila vuota.");
      return top.element;
  }

  public bool IsEmpty(){
      return top==null;
  }

  public void Clear(){
      top=null;
  }
}
```


STACK: considerazioni

- Entrambe le implementazioni hanno metodi con costo *O(1)*, quindi è necessario eseguire un *confronto* sui *tempi di esecuzione* delle due implementazioni (per esempio, effettuando 10^7 chiamate ai metodi `Push()` e `Pop()`):
  - Se non è necessario risparmiare memoria e si conosce la massima dimensione della pila, allora l'implementazione con array è più veloce (per esempio, *ArrayStack = 8.281E-001 sec* e *NodeStack = 1.531E+000 sec*).
  - Se è importante risparmiare memoria e non si conosce il numero di oggetti da inserire nella pila, allora l'implementazione con nodi è più veloce (per esempio, *ArrayStack = 1.984E+000 sec* e *NodeStack = 1.641E+000 sec*).