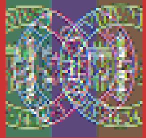


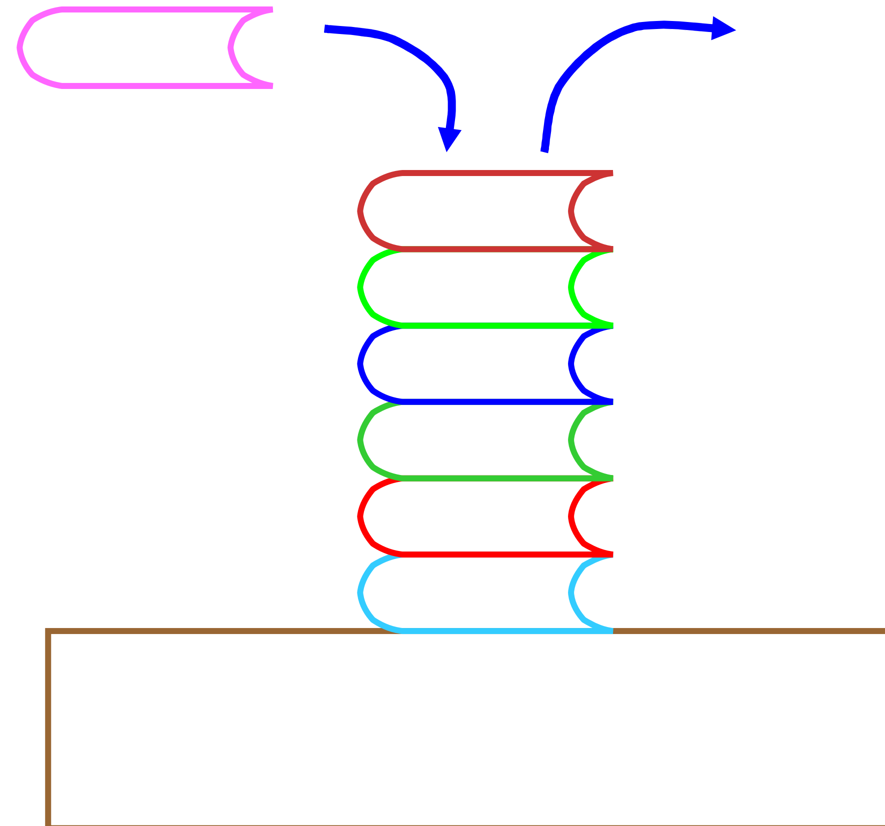
PILE E CODE

Le pile

In alcuni casi è utile disporre di strutture dati che hanno un solo punto di accesso per inserire e reperire i dati.

Ad esempio, una pila di libri.

In una struttura di questo tipo i dati (i libri) vengono inseriti solo in cima e possono essere estratti solo dalla cima.

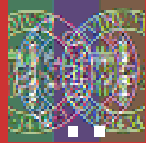


Le pile

Le strutture di questo tipo prendono il nome di Pile (Stack) o Sistemi LIFO (Last In First Out).

Esistono vari esempi di utilizzo di strutture di questo tipo:

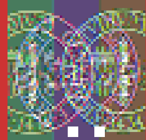
- Persone che entrano in un ascensore
- Chiamate a procedure
- ...



Le pile

Le operazioni che si possono pensare per la struttura LIFO sono:

- Verificare se è piena (IsFull)
- Verificare se è vuota (IsEmpty)
- Inserire un elemento (Push)
- Togliere un elemento (Pop)
- Far restituire il primo elemento, senza estrarlo (TopElem)
- Cancellare tutti i dati (Clear)



Le pile

In alcuni casi le strutture LIFO hanno una dimensione limitata, per cui è necessario definire un valore massimo di elementi inseribili.

Inoltre, per implementare una pila servono:

- uno spazio di memoria ordinato, dove inserire gli elementi:
- un indice, per sapere qual è l'ultimo elemento inserito.

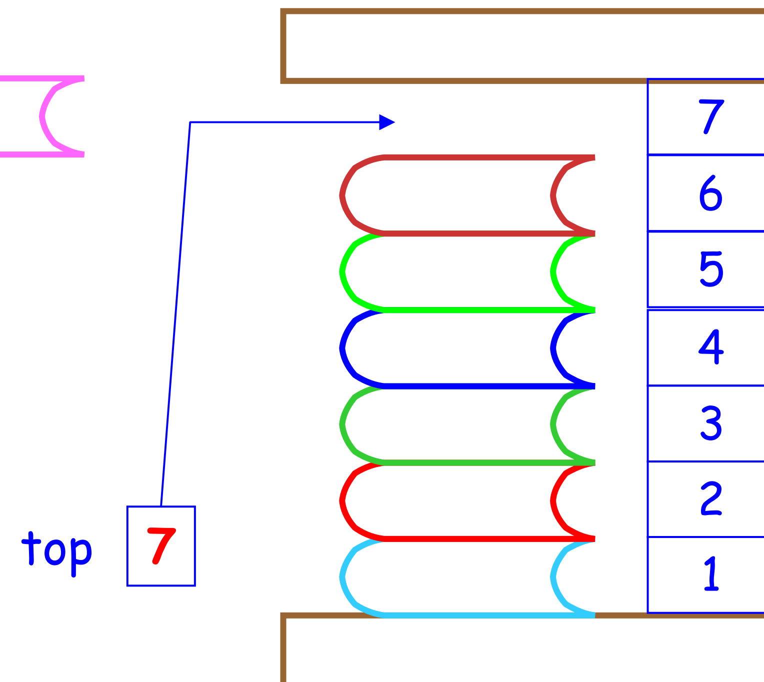
Le pile

L'indice deve tener conto di quanti elementi ci sono nella pila.

Normalmente si utilizza un array per memorizzare gli elementi e un numero intero che indica la prima posizione libera dello stack.

top = 0 → pila vuota

top = max → pila piena



Le pile

```
public class Pila
{ private int top;
  private final int MAXX;
  private int elem[];
  private static final int MAXDEFAULT = 10;

  public Pila()
  { this(MAXDEFAULT);
  }

  public Pila(int max)
  { top = 0;
    MAXX = max;
    elem = new int[MAXX];
  }
  …
}
```

Le pile

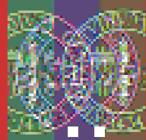
```
public class Pila
{ …

  public boolean IsFull()
  { return (top == MAXX);
  }

  public boolean IsEmpty()
  { return (top == 0);
  }

  public void Clear()
  { top = 0;
  }
  …
}
```

Le pile

Nella procedura di inserimento bisogna eseguire i seguenti passi:

1. verificare che la pila non sia piena;
2. inserire l'elemento appena passato;
3. spostare di una posizione in alto l'indice top.

```
public boolean Push(int val)
{ if (IsFull())
    return false;
  top=top+1;
  elem[top] = val;
  return true;
}
```



Le pile

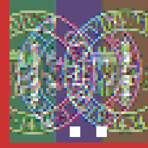
Nella procedura di estrazione bisogna eseguire i seguenti passi:

1. verificare che la pila non sia vuota;
2. decrementare il valore dell'indice;
3. leggere l'oggetto che sta in cima alla pila.

```
public int Pop()
{ if (IsEmpty())
    return 0;
  top=top-1;
  return elem[top+1];
}
```

La procedura che restituisce il valore dell'oggetto in cima è:

```
public int TopElem()
{ if (IsEmpty())
      return 0;
  return elem[top];
}
```



VALUTAZIONE ESPRESSIONI ARITMETICHE CON LO STACK--PILE

Vogliamo che un utente, da tastiera inserisca una String che descrive una operazione aritmetica pienamente parentesizzata.

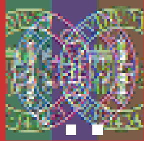
Come risposta il nostro programma deve "valutare" la espressione inserita dall'utente o dare un messaggio di errore se essa non è "ben formata" (cioè non obbedisce alle regole della sintassi delle espressioni).

Es.

input: (((2 +3) * 7)+(3*5))*2

output: 100

La ADT stack qui è molto preziosa…



Esempio illustrativo

Si vuole valutare la espressione

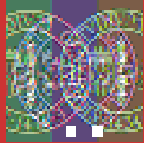
(3*(2+5))

Faremo uso di tre stack:

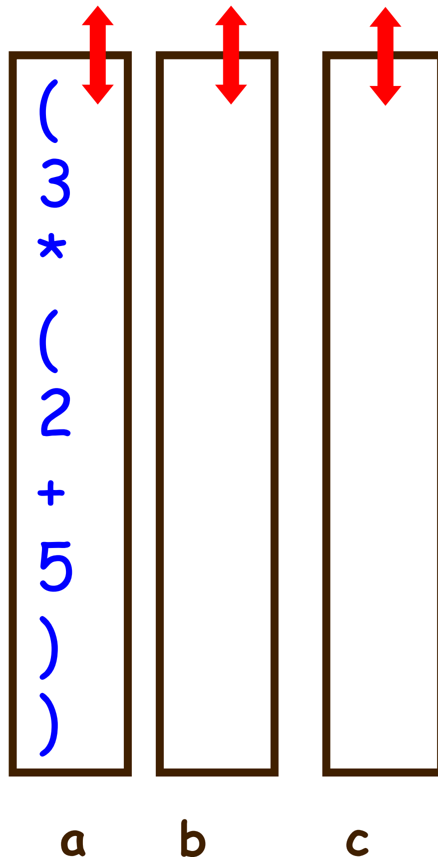
a) uno per i simboli della formula da valutare (esso conterrà dunque dei char);

b) uno per gli operatori che via via saranno "popped" dal primo stack (char);

c) uno per i valori "parziali" generati nella valutazione.

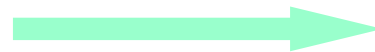


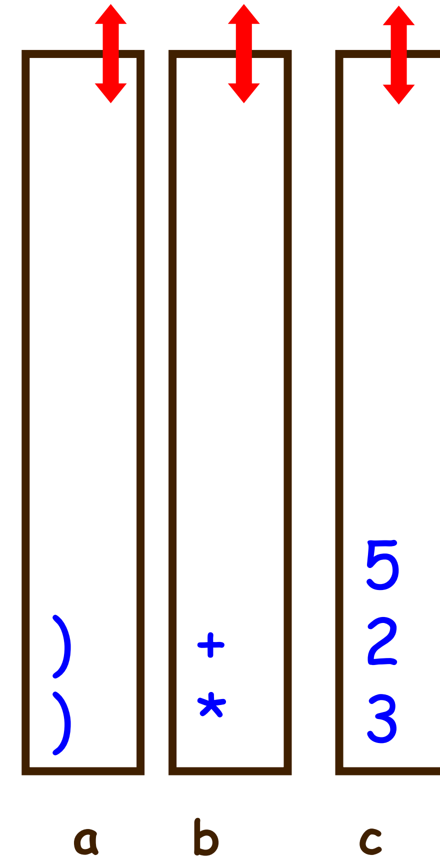
Simulazione



Regola 1: fare pop dallo stack a fino a che non si incontri una parentesi chiusa…
I numeri vengono pushati in c, gli operatori in b, le parentesi aperte trascurate.

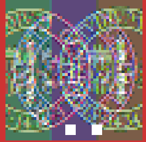
Ciò implica che dopo 7 operazioni successive di push la situazione muta come segue:



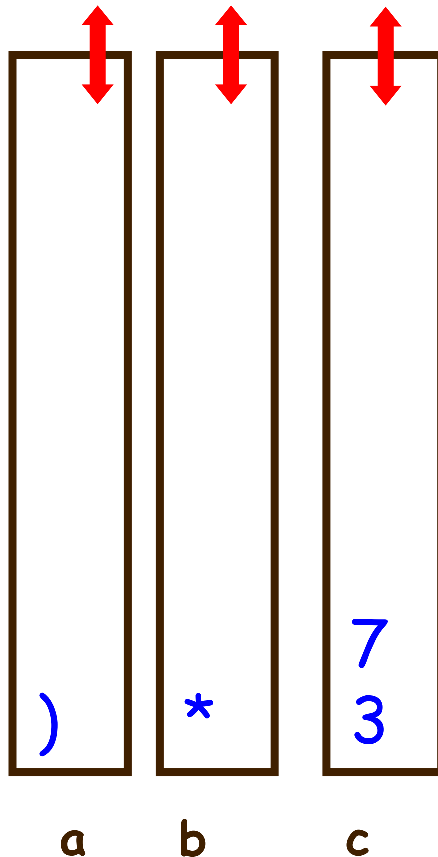


Regola 2: appena si incontra una parentesi chiusa in a, si fa pop di un operatore dallo stack b e si fa il pop di tutti gli operandi richiesti da tale operatore estraendoli dallo stack c (taluni operatori potrebbero essere unari).

Operatore e operandi vengono usati per ottenere un nuovo valore che viene pushed nello stack c.

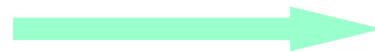


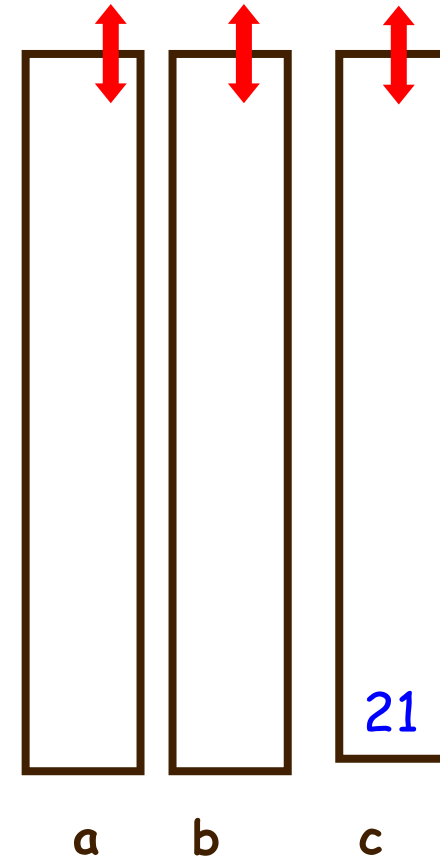
Simulazione 2



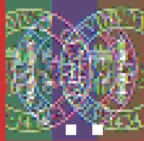
Si riapplica la regola 2: abbiamo una parentesi chiusa, dobbiamo fare pop di *, 10 e 3.

Il risultato è pushed in c.





Regola 3: se lo stack a è vuoto, si controlla lo stack b. Se esso è vuoto si controlla lo stack c. Se in c c'è solo un elemento esso è il valore cercato. Se qualcuna di queste condizioni fallisce la formula non era sintatticamente corretta e non può essere valutata.



Note sulla valutazione

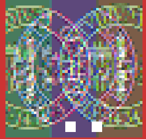
La spiegazione presentata usa 3 stack per chiarezza didattica.

Lo stack a è in realtà "virtuale". Esso contiene elementi di tipo char e richiede solo operazioni di pop fino a che risulti vuoto. Per tali ragioni esso potrebbere non richiede una vera implementazione: la String stessa inserita dall'utente leggendola carattere per carattere da sinistra a destra è una implementazione ready-made di uno stack così semplificato.

Attenzione a operandi e operatori rappresentati da gruppi di caratteri. Esempio:

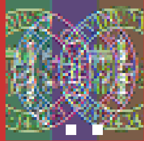
(true AND false)

questa è una formula booleana composta di 16 caratteri, ma i "blocchi" o "token" (gettoni) da tenere in conto nella valutazione sono solo 5.



Riassumendo..DEFINIZIONE ADT Stack

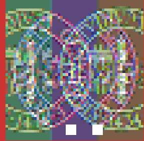
Uno stack (pila) e' un ADT che supporta due operazioni di base: inserimento (push, inserisci in cima) di un nuovo elemento e cancellazione (pop, preleva dalla cima) dell'elemento che e' stato inserito piu' di recente.



Riassumendo..Interfaccia per un ADT

Per convenzione otteniamo l'interfaccia associata all'implementazione di un ADT, cancellando le parti private e sostituendo le implementazioni dei metodi con le loro segnature. Possiamo usare diverse implementazioni aventi la medesima interfaccia senza cambiare in alcun modo il codice dei programmi che usano l'ADT.

```
class intStack //interfaccia di ADT
{ //implementazioni e membri privati nascosti
    intStack(int)
    int empty()
    void push(int)
    int pop()
}
```



Le espressioni aritmetiche

INFISSA

5*(((9+8) * (4*6)) +7)

POSTFISSA

5 9 8 + 4 6 * * 7 + *

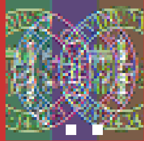
ES: INFISSA B*C POSTFISSA BC*

INFISSA A+B*C POSTFISSA ABC*+

PREFISSA o POLACCA

ES: INFISSA B*C PREFISSA *BC

INFISSA A+B*C PREFISSA +A*BC



Le espressioni aritmetiche

Trasformiamo l'espressione postfissa 5 9 8 + 4 6 * * 7 + * **in infissa** 5*(((9+8) * (4*6)) +7)

POSTFISSA

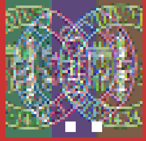
5 9 8 + 4 6 * * 7 + * // NON C'E' BISOGNO DI PARENTESI basta sostituire ogni occorrenza di ab* con (a*b), di ab+ con (a+b)

5 (9+8) (4*6) * 7 + *

5 ((9+8) * (4*6)) 7 + *

5 (((9+8) * (4*6)) + 7) *

(5 * (((9+8) * (4*6)) + 7))



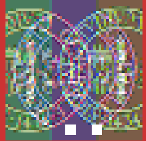
Valutazione di un’espressione postfissa

Usiamo uno stack.

Ci muoviamo da sinistra a destra.

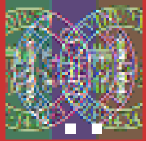
Interpretiamo ogni **operando** come il comando di **inserire** (push) l’**operando** nello stack,

mentre ogni **operatore** come il comando di **estrarre** (pop) gli **operandi** dalla cima dello stack, fare il **calcolo** e **inserire** il risultato nello **stack**.



Valutazione di un’espressione postfissa

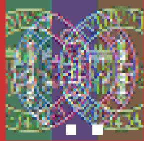
```
class Postfix
{
        public static void main(String[] args)
        {
                char[] a =args[0].toCharArray();
                int N = a.length;
                intStack s = new intStack(N);
                for (int i=0; i<N; i++)
                {

                        if (a[i]=='+')
                                s.push(s.pop()+s.pop());
                        if(a[i]=='*')
                                 s.push(s.pop()*s.pop());
                        //leggi fino al prossimo spazio
                        //converti la stringa letta in un intero val
                        s.push(val);
                }
                Out.println(s.pop());
        }
}
```



Conversione da forma infissa a forma

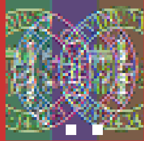
Per trasformare (A+B) nella forma postfissa AB+

1. Usiamo uno stack
2. Leggiamo da sinistra a destra
3. Ignoriamo la parentesi aperta
4. Se incontriamo un numero lo stampiamo
5. Se incontriamo un operatore lo mettiamo nello stack
6. Se incontriamo una parentesi chiusa facciamo il pop dallo stack e lo stampiamo



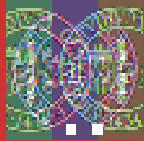
Conversione da forma infissa a forma

```
Class InfixToPostfix
{
    public static void main(String[] args)
    {
        char[] a = args[0].toCharArray();
        int N = a.length;
        charStack s = new charStack(N);
        for (int i=0; i<N; i++)
        {
                if(a[i] == ')')
                        Out.print(s.pop() + " ");
                if ( (a[i] == '+') || (a[i] == '*') )
                        s.push(a[i]);
                if ( (a[i]>='0') && (a[i]<='9') )
                        Out.print(a[i] + " ");
        }
        Out.println();
    }
}
```



Implementazioni generiche

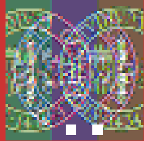
```
class Stack
{
    private Object[] s;
    private int N;
    stack(int maxN)
        { s=new Object[maxN]; N=0; }
    boolean isEmpty()
        { return (N == 0); }
    void push(Object item)
        {N=N+1; s[N] = item;}
    Object pop()
        {N=N-1; Object t = s[N+1]; s[N+1] = null; return t;}
}
```



Stack generico

Possiamo inserire nello satck qualsiasi oggetto Java, e poi usare un cast per assegnarli il tipo richiesto nel momento successivo in cui lo estraiamo dallo stack.

```
Stack s = new Stack(2);
s.push(a); s.push(b);
a = ((Item) s.pop());
b = ((Item) s.pop());
```



Stack generico

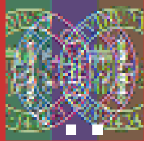
Limiti:

1. Per inserire tipi primitivi es. interi dobbiamo impiegare classi wapper come Integer

```
s.push(new Integer(x))
((Integer) s.pop()).intValue()
```

2. Si ci espone ad errori in running time dovuti ad una sbagliata conversione dei tipi

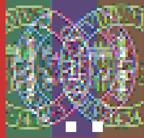
Questi motivi inducono alla scelta di creare degli ADT definiti precisamente.



CODICE ES: PILE

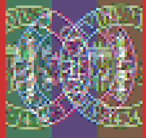
Studiare con attenzione il codice nel file
testPile.java

In esso le operazioni base degli stack sono realizzate con un array. Le situazioni "limite" sono gestite con le appropriate Exception.

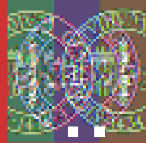


Esercizio

Scrivere un valutatore per espressioni di un tipo scelto da voi, es. formule booleane (attenti alla negazione “unaria!).



CODE



Le code

Gli stack realizzano una "strategia" First In First Out.

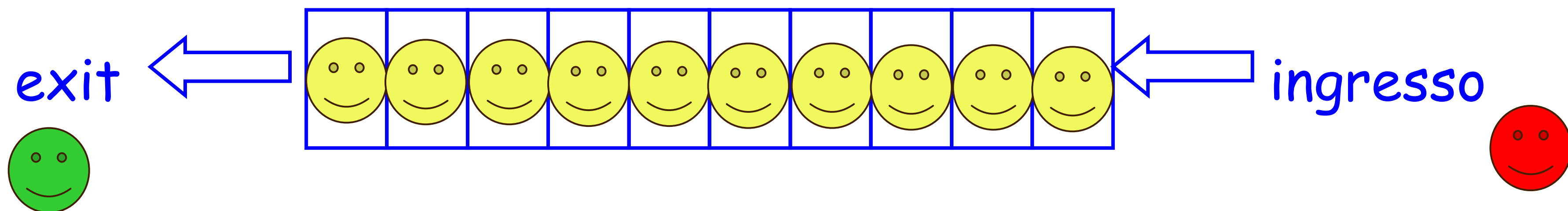
Tale "strategia" possiede una ovvia alternativa, spesso usata in molte situazioni della vita quotidiana (prenotazioni, posta, etc)

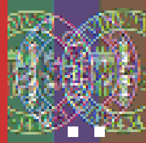
Una struttura dati di questo tipo prende il nome di Coda o Sistema FIFO (First In First Out).

Dal punto di vista ADT una CODA o QUEUE (leggi "chiù") è una sequenza di elementi che può essere "accorciata" da un lato e allungata da un altro lato:

Dequeue: operazione che corrisponde alla estrazione di un elemento dalla coda

Enqueue: operazione che corrisponde all'inserimento di un elemento dalla coda

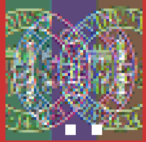




Le code

Dal punto di vista astratto per la struttura coda (*queue*) le operazioni "base" sono le seguenti:

- inserire un elemento x in coda (EnQueue(x));
- togliere un elemento dalla coda (DeQueue());
- verificare se la coda è vuota (IsEmpty());
- cancellare tutti i dati (ClearQueue());
- leggere (senza toglierlo dalla coda) il primo elemento in attesa (readHead());
- nel caso in cui si preveda una coda con capacità massima limitata verificare se la coda ha raggiunto la sua massima capacità: (IsFull()).



Code: implementazione con array

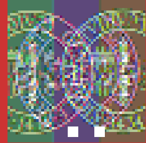
Gli array sono il supporto più naturale che viene in mente dovendo gestire una "sequenza". Essi purtroppo presentano numerosi problemi nel momento in cui debbono gestire le operazioni tipiche di una coda.

PRIMA PROPOSTA di implementazione con array
(ingenua ed inefficiente):

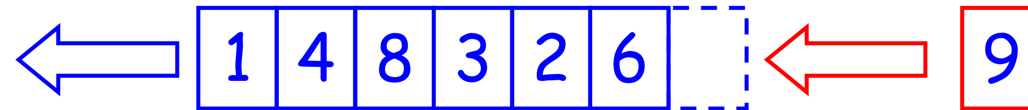
Un array di dimensioni fissate fornisce lo spazio di memoria dove vengono messi in sequenza gli elementi della coda.

L'estrazione dall'inizio della coda prevede che si estragga l'elemento di indice 0 e che tutti gli altri elementi successivi vengano "scivolati" avanti di un indice. Inefficiente!

L'inserimento avviene semplicemente inserendo un elemento nel primo indice libero in fondo all'array. Dovrò pertanto mantenere in una variabile il valore di questo indice e aggiornarlo sia all'inserimento che alla estrazione.



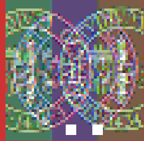
Le code



In seguito ad ogni estrazione, gli elementi rimanenti vengono fatti avanzare di una posizione.

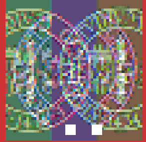
Per l'inserimento viene invece occupata l'ultima posizione libera.

È possibile implementare una coda di questo tipo per mezzo di un array.



Simulazione

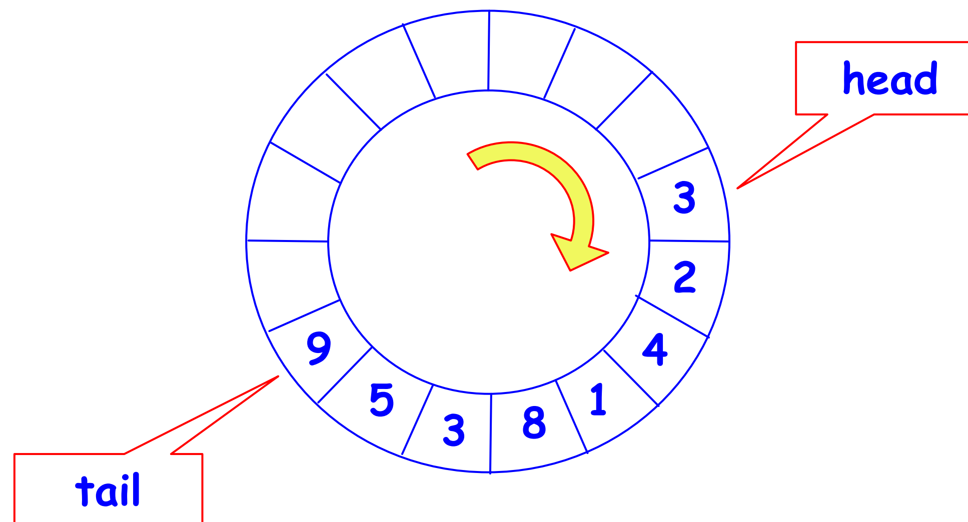
Operazione		0	1	2	3	4	5	6		
Enqueue 9		1	4	8	3	2	6		9	Posizione libera= 6
		1	4	8	3	2	6	9		Posizione libera= 7 (queue full)
Dequeue	1	4	8	3	2	6	9			Posizione libera= 6
Dequeue	4	8	3	2	6	9				Posizione libera= 5
Dequeue	8	3	2	6	9					Posizione libera= 4
Enqueue 7		3	2	6	9				7	Posizione libera= 4
		3	2	6	9	7				Posizione libera= 5

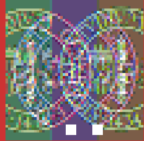


Le code circolari

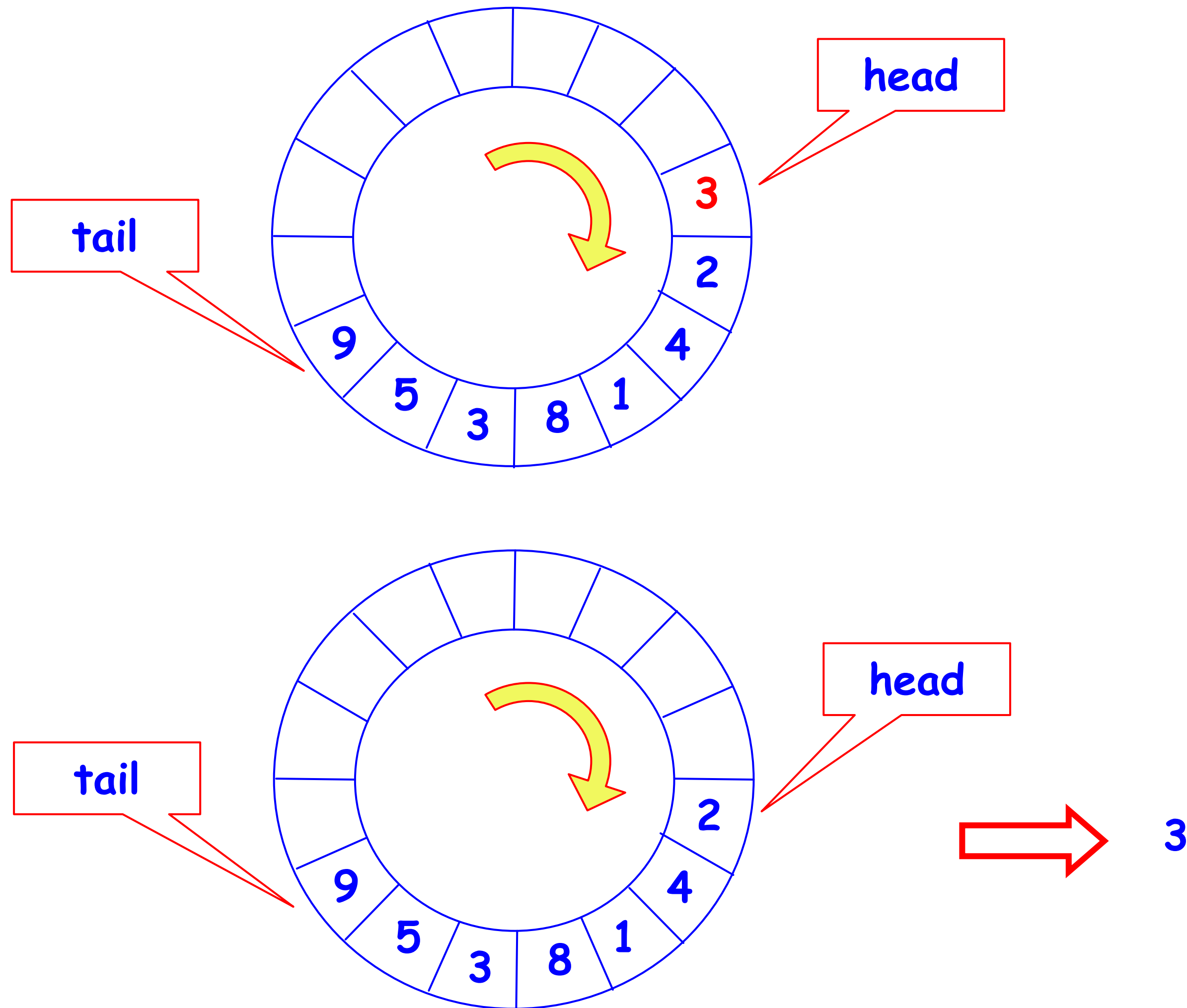
Tuttavia eseguire uno shift di tutti gli elementi dopo ogni estrazione è troppo oneroso.

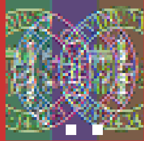
Per tale motivo è stata pensata una struttura, denominata coda circolare, nella quale esistono due indici, head e tail, che indicano il primo elemento e l'ultimo.



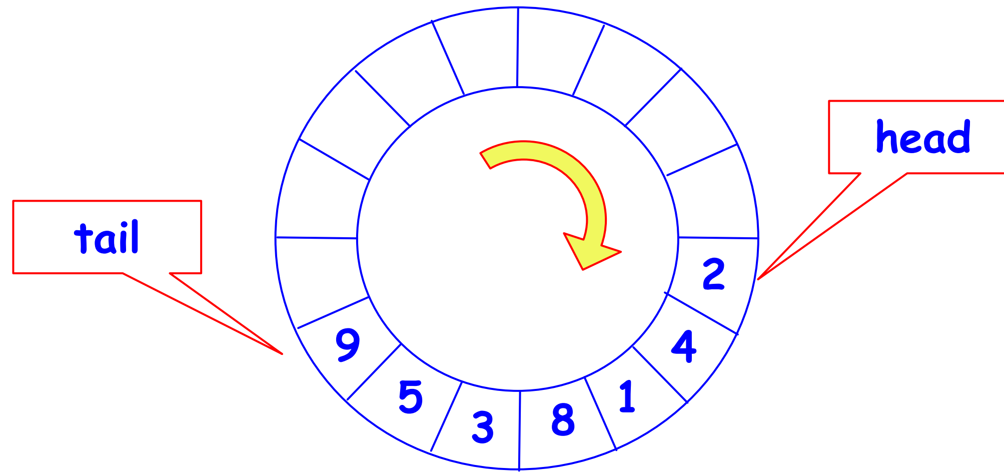


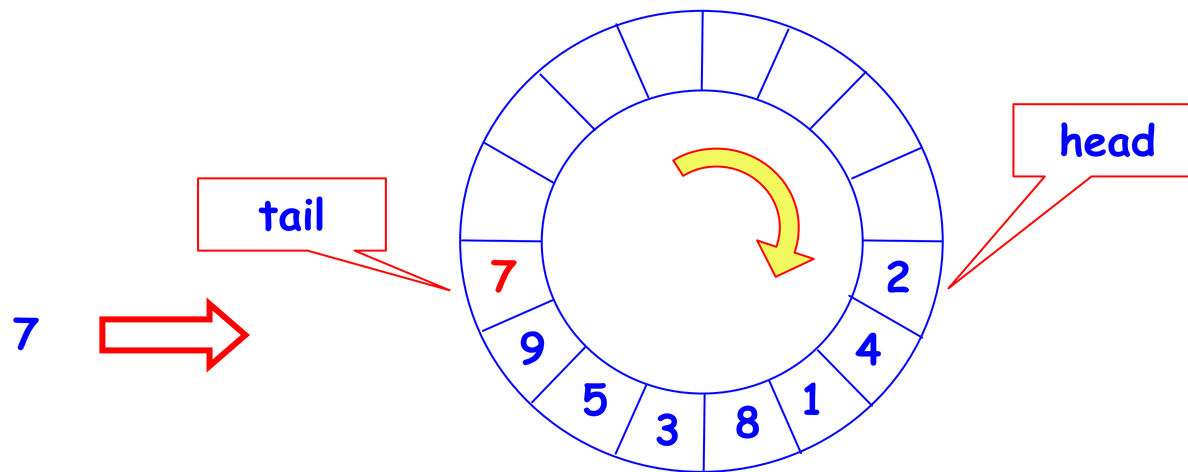
Le code circolari

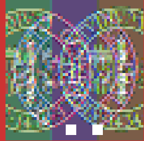




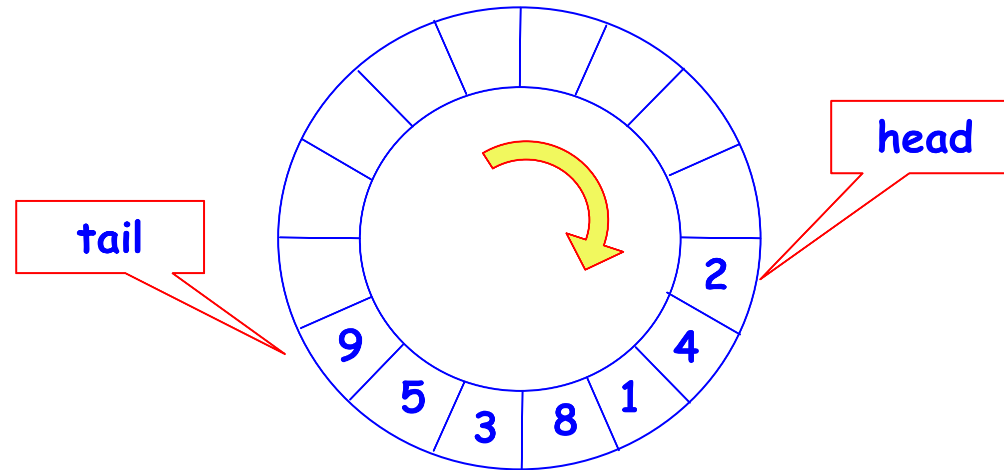
Le code circolari





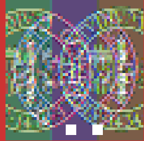


Le code circolari

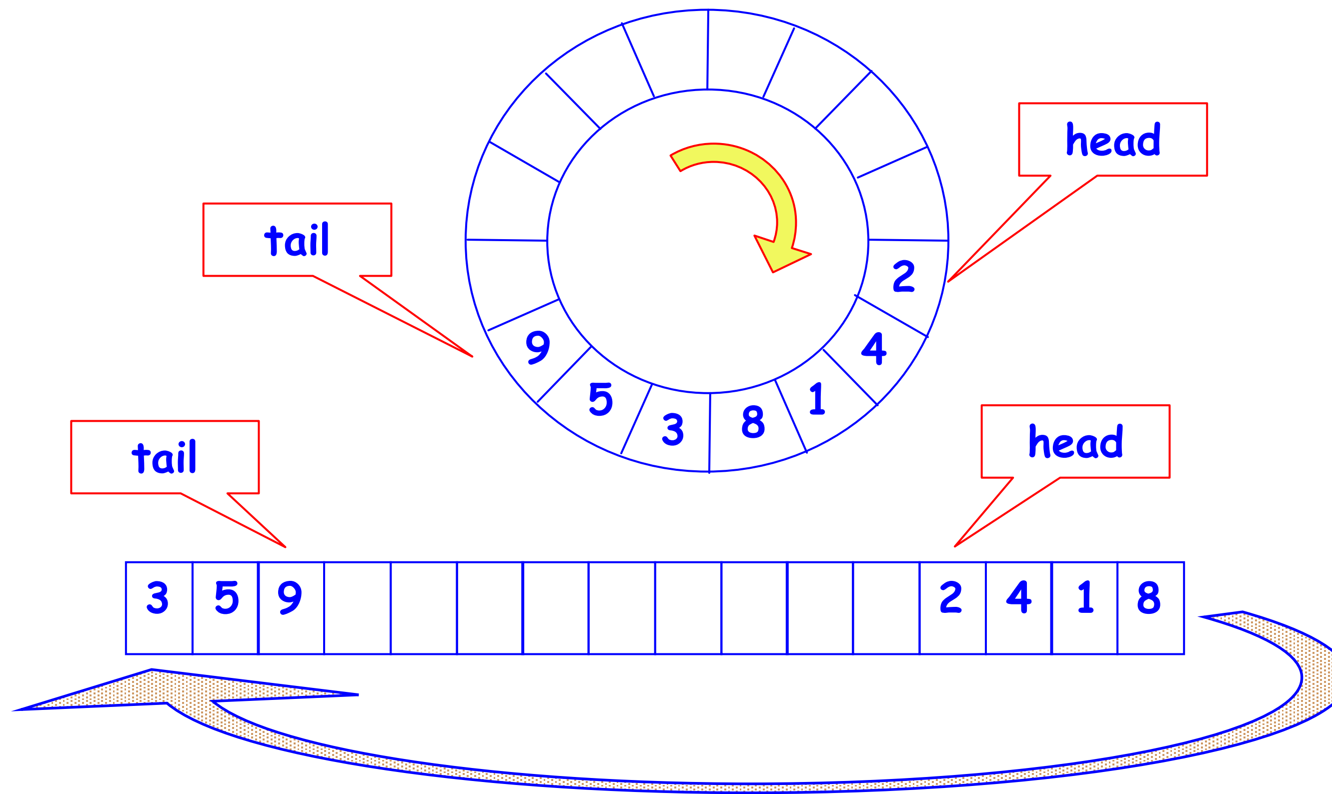


Il vantaggio di una simile struttura logica è che non è necessario effettuare shift per ogni inserimento, ma basta una sola assegnazione (più la modifica della variabile **head**).

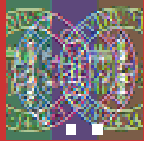
Ogni operazione di Enqueue o Dequeue comporta l'avanzamento di uno degli indici (**tail** per Enqueue, **head** per Dequeue).



Code circolari- diagramma



L'espressione andrebbe corretta in "code con implementazione circolare"



Operazioni modulari

Piccola nota:

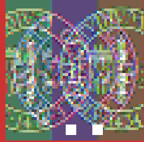
Sia A l'array di supporto e sia DIM=A.length;

Una **operazione di Dequeue** aggiorna l'indice di testa come segue:

```
indiceTesta=(indiceTesta+1)%DIM;
```

Una **operazione di Enqueue** aggiorna l'indice di coda come segue:

```
indiceCoda=(indiceCoda+1)%DIM
```



Code: implementazione con array(2)

La prima ingenua proposta è facilmente superata se accettiamo due idee:

a) Pensiamo all'array come ad un anello "chiuso" in cui l'ultimo elemento è precedente del primo;
b) Anziché mantenere una sola variabile per indicare il primo elemento libero in fondo alla coda, ne manteniamo anche un'altra che indica la posizione della testa della coda.

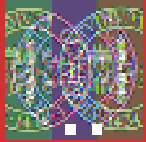
SECONDA PROPOSTA di implementazione mediante array.

Un **array di dimensioni fissate** fornisce lo spazio di memoria dove vengono messi in sequenza gli elementi della coda.

Due variabili intere indiceTesta, indiceCoda mantengono il valore della posizione del primo elemento in coda e del posto dove si può inserire un successivo elemento.

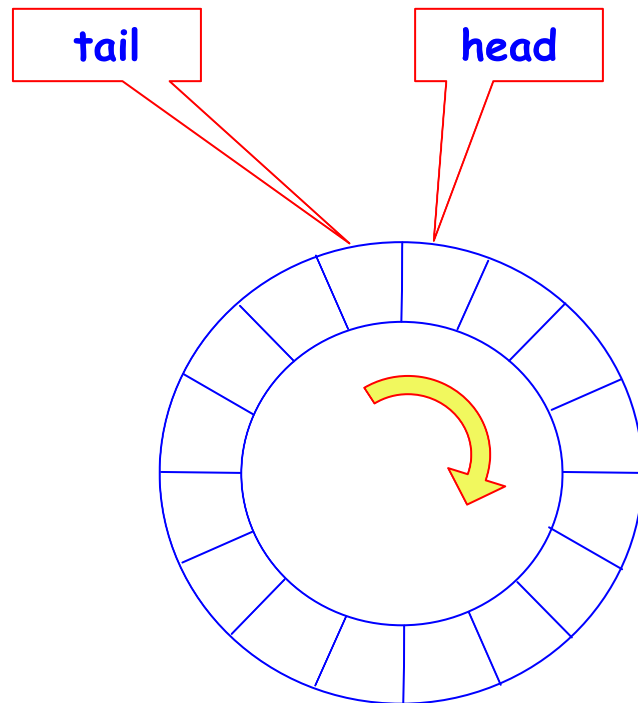
L' **estrazione** dall'inizio della coda prevede che si estragga l'elemento di indice indiceTesta, dopo l'estrazione tale indice viene incrementato di una unità. Se tale aumento porta oltre il limite dell'array esso viene riportato a 0. Ciò si realizza usando l'operatore aritmetico "modulo lunghezza dell'array -1". Questo semplice accorgimento elimina l'inefficienza della prima proposta.

L'**inserimento** avviene inserendo un elemento nel primo indice libero in fondo all'array e aggiornando (con analogo conto "modulare") l'indiceCoda.

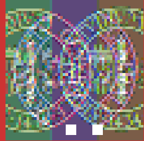


Le code circolari

Gli indici head e tail vengono sempre incrementati, rispettivamente per le operazioni di DeQueue e EnQueue.

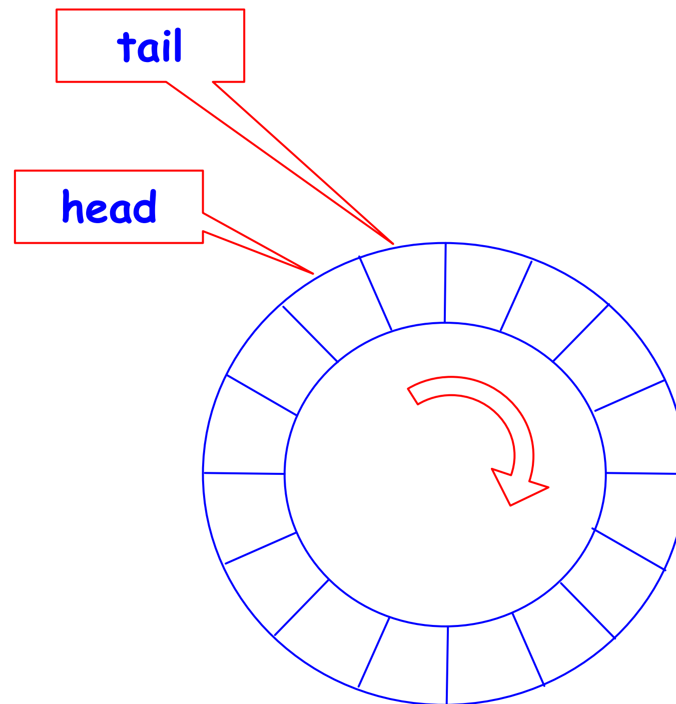


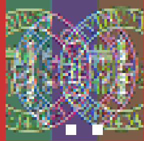
Chiaramente il valore dell'indice tail potrà raggiungere ma non superare il valore dell'indice head (a seguito di operazioni di Enqueue → riempimento della coda).



Le code circolari

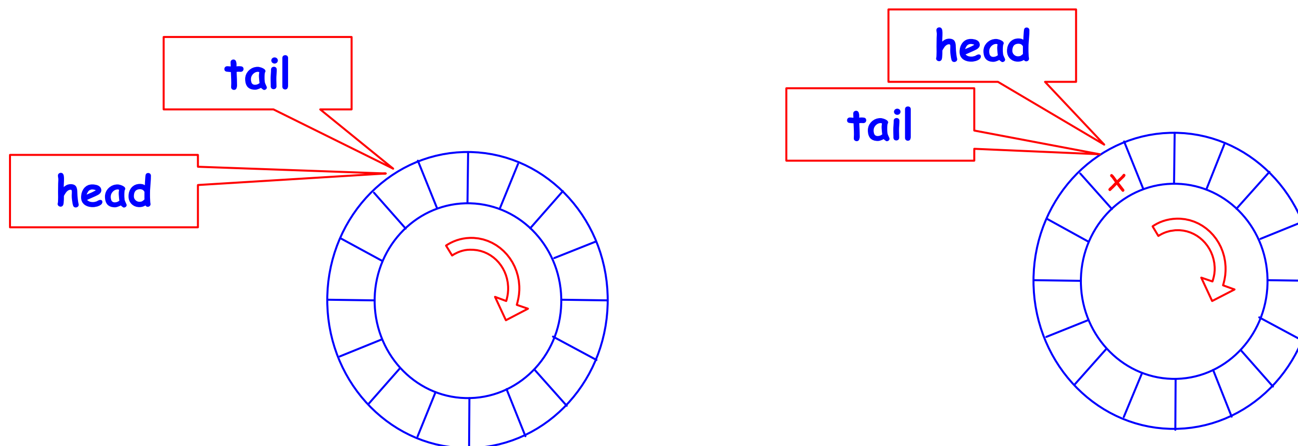
Analogamente head non potrà superare tail (dopo operazioni di Dequeue → svuotamento della coda).

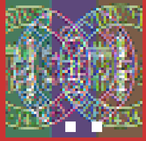




Le code circolari

Se però i due puntatori coincidono, dobbiamo poter distinguere le condizioni di coda vuota o coda con un solo elemento.

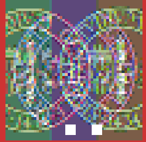




Le code circolari

Ci sono due modi per risolvere questo problema:

1) Lasciare sempre una casella vuota e far indicare a tail la prima posizione vuota;
2) Usare una variabile booleana per sapere se la coda contiene elementi.

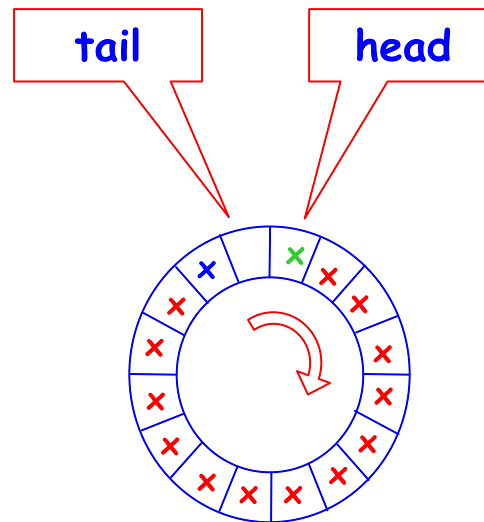


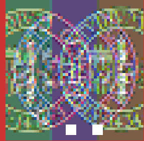
Le code circolari

Nel primo caso gli indici head e tail si possono sovrapporre solo se la coda è vuota.

Head punta al primo elemento della coda e tail punta alla prima posizione libera dopo l'ultimo elemento (tranne se la coda è vuota).

Come si può notare rimane sempre una casella vuota.

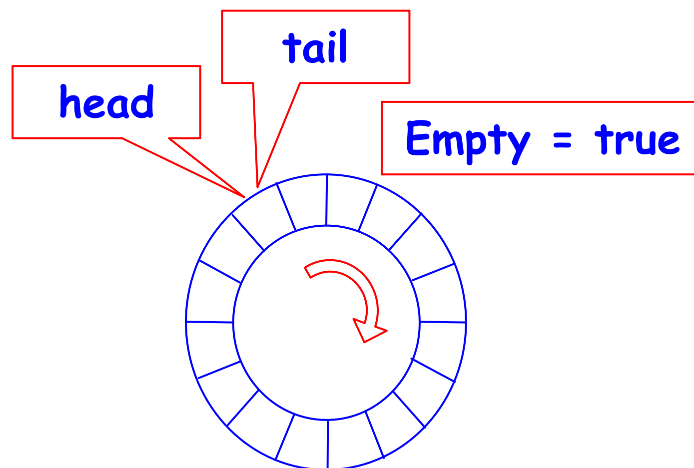


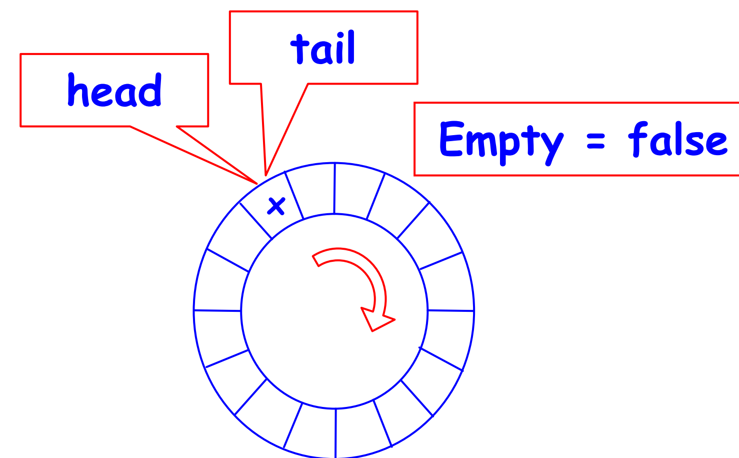


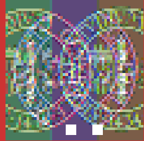
Le code circolari

Nel secondo caso la head punta alla prima posizione piena e tail punta all'ultima piena (tranne se la coda è vuota).

In tal caso se gli indici si sovrappongono la coda può essere vuota o con un solo elemento. Tramite una variabile booleana (necessaria) si possono distinguere i due casi.



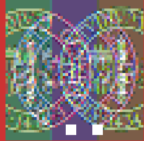




Le code circolari

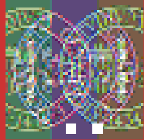
Nella seguente implementazione prenderemo in esame il primo caso.

```
public class Queue
{ private int head, tail;
  private final int MAX;
  private int elem[];
  private static final int MAXDEFAULT = 10;

  public Queue()
  { this(MAXDEFAULT);
  }

  public Queue(int max)
  { head = tail = 0;
    MAX = max;
    elem = new int[MAX];
  }
 …
```



Le code circolari

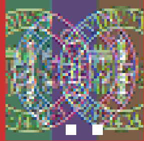
```
public class Queue
{ …

  public boolean IsFull()
  { return (head == (tail+1) % MAX);
  }

  public boolean IsEmpty()
  { return ( head == tail );
  }

  public void ClearQueue()
  { head = tail = 0;
  }
  …
}
```



Le code circolari

Qualunque operazione che coinvolga gli indici deve essere fatta modulo la dimensione dell'array.

```
public int FirstElem()
{ if(IsEmpty())
      return 0;
   return elem[head];
}
```



Le code circolari

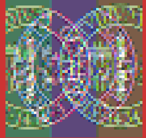
```
public boolean EnQueue(int val)
{ if(IsFull())
    return false;
  elem[tail] = val;
  tail = ++tail % MAX;
  return true;
}

public int DeQueue()
{ if(IsEmpty())
    return 0;
  int val = elem[head];
  head = ++head % MAX;
  return val;
}
```

Codice

Studiare con attenzione i file JAVA:

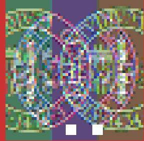
1. **`coda1.java`**
   (implementazione con array che usa la strategia 1)
2. **`coda2.java`**
   (implementazione con array che usa la strategia 2)

In queste due implementazioni sono previste anche le eccezioni per le operazioni "impossibili" o "proibite".



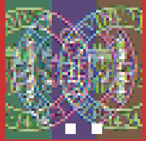
Pile e Code implementate usando le LISTE

Eccezioni su Pile e Code



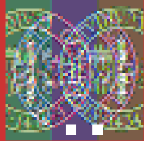
Le pile: metodi con eccezioni

```
public class Stack
   {  private Object [ ] elem;
      private int top;
      private final int MAX;
      private static final int MAXDEFAULT = 10;


          public Stack( )
       { this( MAXDEFAULT );
       }
          public Stack( int max )
       {   MAX = max;
           elem = new Object[ MAX ];
           top = 0;
       }
```



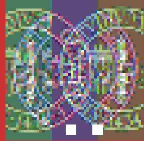
Le pile : metodi con eccezioni

```
public Object topElem( )
    {
        if( isEmpty( ) )
            return null;
        return elem[ top ];
    }
```



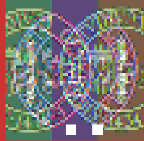
Le pile : metodi con eccezioni

```
public Object pop( ) throws Underflow
        {
            if( isEmpty( ) )
                throw new Underflow( );
            return elem[ --top ];
        }
public void push( Object x ) throws Overflow
        {
            if( isFull( ) )
                throw new Overflow( );
            elem[ top++ ] = x;
        }
```



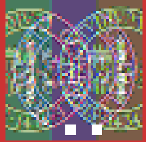
Le pile : metodi con eccezioni

```
public class Overflow extends Exception
   {
   }

public class Underflow extends Exception
   {
   }
```



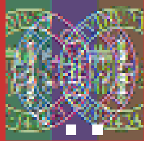
Le pile : metodi con eccezioni

```
//Un semplice metodo di stampa
      public void printStack( )
      {
         if( this.isEmpty( ) )
            System.out.print( "Empty Stack" );
         else
         {
           for(int i=0; i<top; i++)
             System.out.print( elem[i] + " " );
         }
      }
```



Le pile : metodi con eccezioni

```
//Test della classe Stack
public static void main( String [ ] args )
{   Stack s = new Stack( 12 );
        try {for( int i = 0; i < 10; i++ )
                s.push( new Integer( i ) );}
    catch( Overflow e ) {System.out.println("Unexpected overflow" ); }
    System.out.println("\nSize Stack = "+ s.getSize() );
        while( !s.isEmpty( ) )
     try {System.out.println( s.pop( ) );}
     catch (Underflow e) {System.out.println("Unexpected underflow"
);}
    System.out.println("\nSize Stack = "+ s.getSize());
}
```



Pile implementate usando liste legate

Lo stack si presta ad essere implementato con una lista legata in cui si effettua un push inserendo in testa e un pop cancellando sempre in testa.

In questo caso la lista legata può essere molto semplificata rispetto ad una generica lista legata.

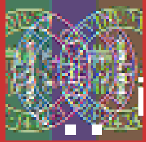
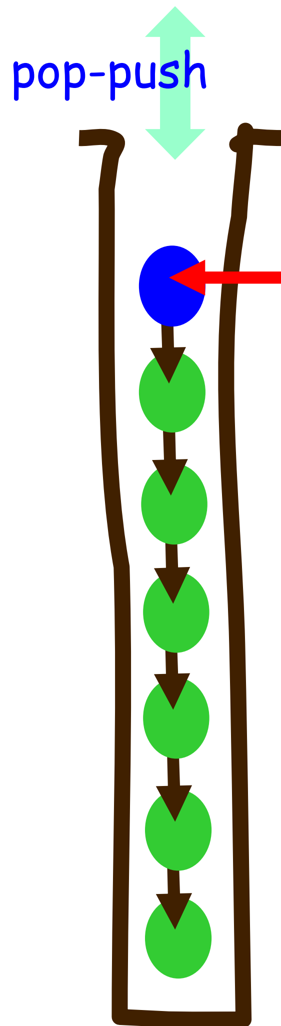


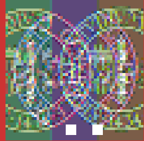
illustrazione della implementazione dello stack con le linked list



Head o Top : questo riferimento è il vero punto di accesso a tutta la struttura stack

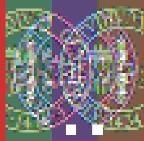
Le operazioni di pop e push si possono effettuare solo sull'elemento in blu.

Nessun bisogno di tenere il "conto" di quanti elementi sono nello stack.



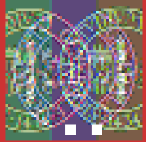
ESEMPIO 1: Il codice si trova nel file testPile2.java.

ESEMPIO 2: Si studino i dettagli di un valutatore di espressioni aritmetiche con gli operatori + e *. Tale esempio usa la implementazione degli stack mediante liste legate. Codice nella directory “valutatore”



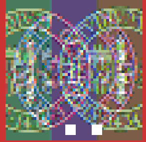
Le code : metodi con eccezioni

```
public class Queue
   {private Object [ ] elem;
     private final int  MAX;
     private int  head, tail;
     private static final int MAXDEFAULT = 10;

public Queue( )
      {
          this( MAXDEFAULT );
      }

public Queue( int max )
      {
          head = tail = 0;
          MAX = max;
          elem = new Object[ MAX ];
      }
```



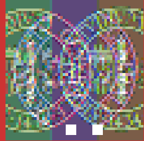
Le code : metodi con eccezioni

```
public boolean isEmpty( )
        {
            return  head == tail ;
        }

public boolean isFull( )
        {
            return (head == (tail + 1) % MAX);
        }
```



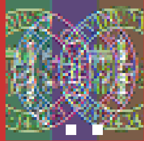
Le code : metodi con eccezioni

```
public void makeEmpty( )
        {
            head = tail = 0;
        }

public Object firstElem( )
        {
            if( isEmpty( ) )
                return null;
            return elem[ head ];
        }
```



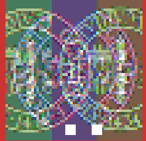
Le code : metodi con eccezioni

```
public Object dequeue( ) throws Underflow
    {
        if( isEmpty( ) ) throw new Underflow();
        Object frontItem = elem[ head ];
        elem[ head ] = null;
        head = ++ head % MAX;
        return frontItem;
    }

public void enqueue( Object x ) throws Overflow
    {
        if( isFull( ) )  throw new Overflow( );
        elem[ tail ] = x;
        tail = ++ tail % MAX;
    }
```



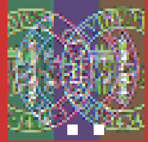
Le code : metodi con eccezioni

```
//Test della classe Queue
    public static void main( String [ ] args )
    {
        Queue q = new Queue(11);

        try
        {
            for( int i = 0; i < 10; i++ )
                q.enqueue( new Integer( i ) );
        }
        catch( Overflow e ) { System.out.println( "Unexpected overflow" ); }

        while( !q.isEmpty( ) )
          try
          {
            System.out.println( q.dequeue( ) );
          }
         catch( Underflow e ) { System.out.println( "Unexpected underflow" ); }
    }
```



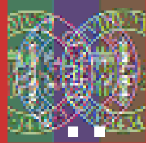
Esercizio

Invertire l'ordine degli elementi di una pila S usando una coda Q.



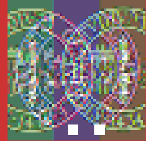
Esercizio

```
public class Ex1 {

  public static void reverseStack (Stack S) throws Overflow,Underflow
  {
    Queue Q = new Queue(S.getSize()+1);
                    //Implementazione con coda circolare

    while( !S.isEmpty( ) )
          Q.enqueue(S.pop());

    while( !Q.isEmpty( ) )
          S.push( Q.dequeue( ) );

    return;
  }
```



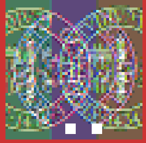
Esercizio

```
public static void main (String[] args) {
  //Testiamo il funzionamento del metodo reverseStack
   su uno stack di elementi di tipo intero

    Stack S = new Stack( 10 );

      try
        {
           for( int i = 0; i < 10; i++ )
           S.push( new Integer( i ) );
         }
       catch( Overflow e ) {System.out.println(
                                   "Unexpected overflow" ); }
      S.printStack();
      System.out.println("\nSize Stack = "+ S.getSize() );
```



Esercizio

```
……
try
      {
        reverseStack(S);
      }
      catch( Overflow e ) { System.out.println(
                                      "Unexpected overflow" ); }
      catch( Underflow e ) {System.out.println(
                                    "Unexpected underflow" ); }

      S.printStack();
      System.out.println("\nSize Stack = "+ S.getSize() );
 }
```



Implementazione delle CODE in struttura

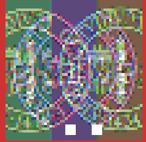
Le liste legate sono un metodo “ideale” per implementare le code. Anzi: è sufficiente una implementazione molto semplificata di tali liste in quanto alcune operazioni non sono necessarie.

Si deve infatti solo prevedere l’inserimento in coda alla lista e il delete della testa.

Si debbono inoltre prevedere due riferimenti: uno alla testa e uno alla coda.

Si veda la implementazione nel file JAVA:

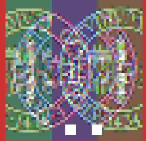
`code3.java`



Soluzioni alternative

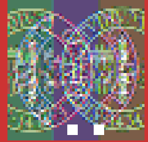
- E’ possibile implementare una pila, sfruttando i meccanismi di ereditarietà tipici della programmazione ad oggetti definendo una nuova classe `StackListByInheritance` che estende opportunamente la classe `LinkedList`.

- E’ possibile ancora implementare una pila per composizione utilizzando un oggetto *private* della classe `LinkedList`.



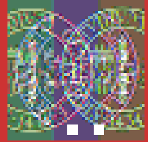
Esercizio

Trasferire gli elementi di una pila **S1** in una pila **S2** in modo che gli elementi di **S2** risultino avere lo stesso ordinamento che avevano in **S1** (usare una terza pila di supporto).



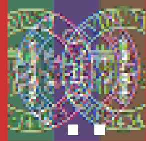
Esercizio

```
public class Ex2 {
 public static void transferStack (Stack S1, Stack S2) throws
   Overflow,Underflow
 {
   //Stack S3 di appoggio

   Stack S3 = new Stack(S1.getSize());

   while( !S1.isEmpty( ) )
        S3.push(S1.pop());

   while( !S3.isEmpty( ) )
        S2.push(S3.pop());
 }
```



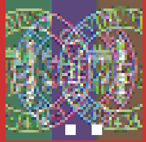
Esercizio

```
public static void main (String[] args) {

   Stack S1 = new Stack( 10 );
   Stack S2 = new Stack( 10 );

     try
       {
          for( int i = 0; i < 10; i++ )
          S1.push( new Integer( i ) );
        }
      catch( Overflow e ) {System.out.println("Unexpected overflow" );
   }

     System.out.print( "S1: ");
     S1.printStack();
     System.out.print( "\nS2: ");
     S2.printStack();  ……
```



Esercizio

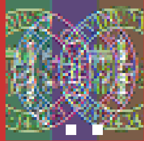
```
………
try
      {
        transferStack(S1,S2);
      }
      catch(Overflow e ) {System.out.println( "Unexpected overflow" ); }
      catch(Underflow e ) { System.out.println( "Unexpected underflow" ); }

    System.out.print( "\nS2: ");
      S2.printStack();

  }
}
```



Esercizio - (Num. 2 – pag. 156 Drozdek)

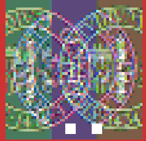
Scrivere un programma per convertire un numero da notazione decimale ad una notazione espressa in una base (o radice) compresa tra 2 e 9. La conversione si effettua con ripetute divisioni per la base in cui si sta convertendo il numero prendendo i resti delle divisioni in ordine inverso.

IDEA: Utilizzare una pila dove memorizzare i resti. Per ottenere i dati nell'ordine corretto, basterà semplicemente svuotare la pila.



Conversione tra basi di numerazione

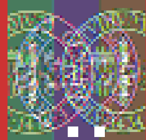
```
public static String decToBasex(int n, int x) throws
   Overflow,Underflow
  {
    String b= "";

    Stack S= new Stack();
    do{
       S.push(new Integer (n % x));
       n = n / x;
      } while(n!=0);

    do
       {
        b= b + ((Integer) S.pop()).toString();
      } while(!S.isEmpty());

      return b;
  }
```



Addizione di numeri molto grandi

Il valore massimo dei numeri interi in un qualsiasi linguaggio di programmazione è limitato.

Il problema può essere risolto trattando questi numeri come stringhe di cifre numeriche, memorizzando i valori corrispondenti a queste due cifre su due pile ed eseguendo l’addizione estraendo numeri dalle pile.



Esempio

592	2	9	5	1
+ 3784	+ 4	+ 8	+ 7	+ 3
4376	6	17	13	4

operand-Stack1: [2, 9, 5] → [9, 5] → [5] → [ ] → [ ]

operand-Stack2: [4, 8, 7, 3] → [8, 7, 3] → [7, 3] → [3] → [ ]

2 + 4 → 6

9 + 8 → 17

5 + 7 (+ 1) → 13

(1 +) 3 → 4

result-Stack: [ ] → [6] → [7, 6] → [3, 7, 6] → [4, 3, 7, 6]