Programmazione I

Il Linguaggio C

Strutture Dati – Code e Stack

Daniel Riccio

Università di Napoli, Federico II

10 dicembre 2021

Sommario

- Argomenti
  - Code e Stack

La struttura dati coda

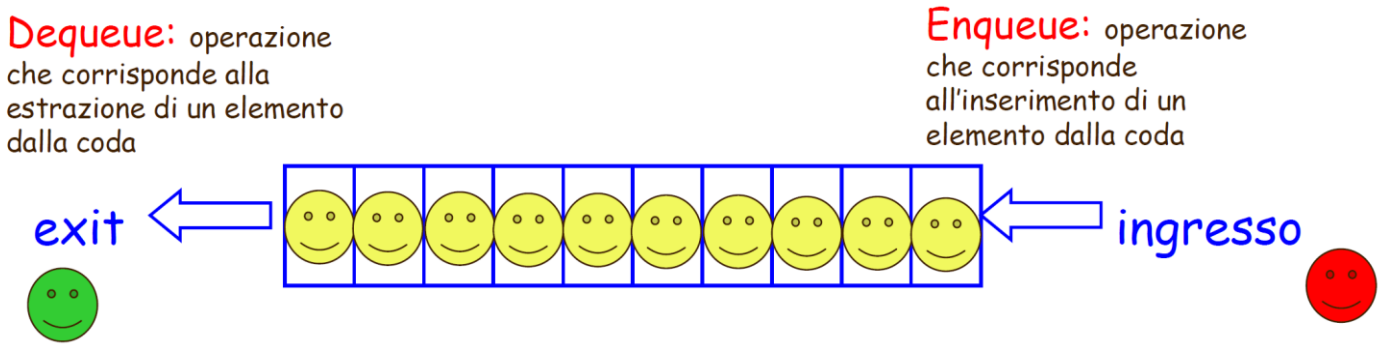
Una **coda** (o **queue**) e una struttura dati astratta le cui modalità di accesso sono di tipo FIFO.

- **FIFO** (**F**irst **I**n **F**irst **O**ut): i dati sono estratti rispetto all'ordine di inserimento.
- **LIFO** (**L**ast **I**n **F**irst **O**ut): i dati sono estratti in ordine inverso rispetto al loro inserimento.

La politica **FIFO** è comunemente adottata in numerosi contesti differenti.

Nella vita reale, è utilizzata in tutti (o quasi) i contesti in cui è necessario attendere per poter ottenere un servizio (coda al supermercato, allo sportello bancario, ecc).

In informatica, è utilizzata per la gestione dei processi su un sistema operativo, per la gestione del flusso di dati tra periferiche, ecc



La struttura dati coda

Esistono due approcci per implementare una struttura dati di tipo coda in C:

- Code con **capacita illimitata** – struttura dati di tipo lista concatenata
- Code con **capacita limitata** – implementazione ad-hoc tramite array

Una struttura dati di tipo coda può essere vista come una struttura dati di tipo lista che supporta un numero limitato di operazioni:

- **Enqueue** – inserimento in coda (o, equivalentemente, testa)
- **Dequeue** – rimozione in testa (o in coda)

Anche in questo caso ci concentriamo unicamente su implementazioni di code che ci permettano di memorizzare interi (tipo di dato int)

Vogliamo che il nostro codice sia **efficiente**: in termini di utilizzo di memoria e tempo di calcolo
Vogliamo che il nostro codice sia **modulare**: le funzioni devono essere semplici, specifiche e riutilizzabili

La coda: definizione di tipo

Il nodo del tipo di dato **coda** è equivalente al nodo del tipo di dato **lista**.
Modifichiamo la struttura dati di accesso in modo da conservare sia un puntatore alla testa, che un puntatore al nodo terminale.

Struttura del nodo:

```
typedef struct Nodo_SL {
    int dato;
    struct Nodo_SL *next;
} Nodo_SL;
```

Struttura della coda:

```
typedef struct Coda {
  Nodo_SL *head;
  Nodo_SL *tail;
} Coda;
```

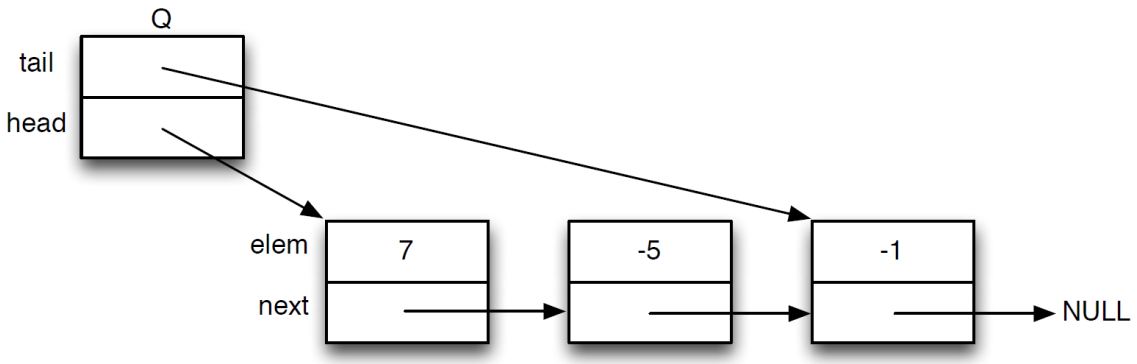
La coda: funzioni per la gestione

Consideriamo i seguenti prototipi, che definiscono le operazioni richieste su una coda.

```
int enqueue(Coda *Q, int dato);

int dequeue(Coda *Q, int *dato );

int LeggiInTesta(Coda *Q, int *dato);
```

L'operazione di rimozione in testa è efficiente per tutte le rappresentazioni concatenate viste

L'operazione di inserimento in coda è reso efficiente grazie al puntatore alla coda inserito nella struttura Coda



Creazione della coda

Per creare una coda, basta definirla, ovvero è sufficiente creare il modo di riferirsi ad essa.

L'unica cosa che esiste sempre della coda è il suo **punto di accesso** (o **radice**).

Questa è l'unica componente **allocata staticamente** e all'inizio i puntatori **head** e **tail** sono inizializzati a **NULL**, in quanto non ci sono elementi.

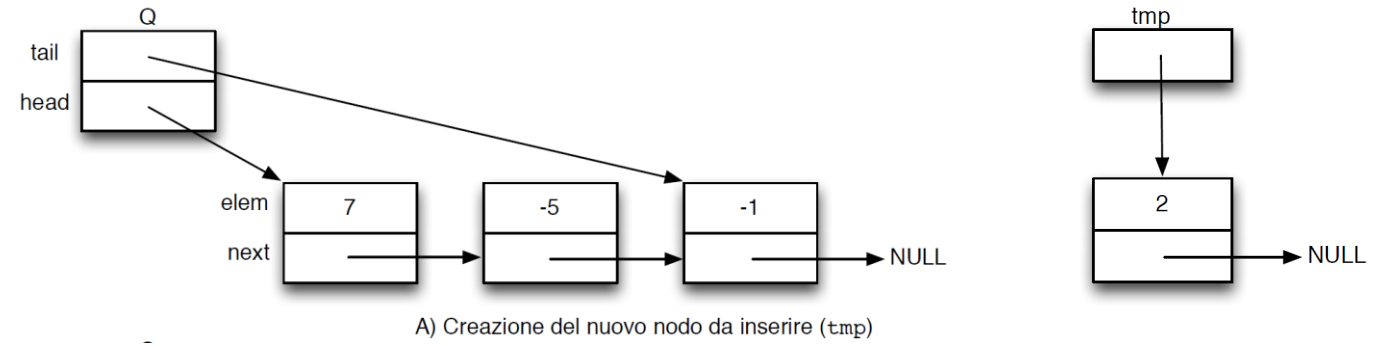
Es.:

```
Coda Q;
Q.head = NULL;
Q.tail = NULL;
```

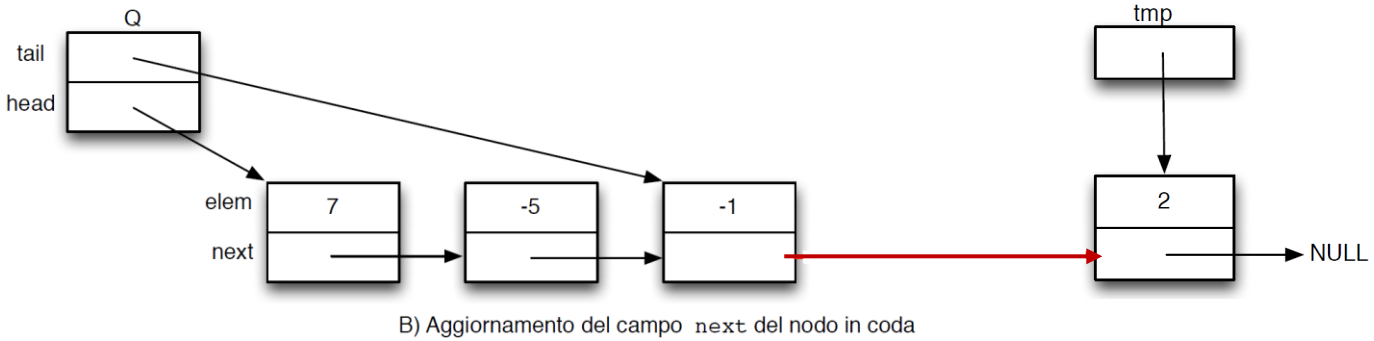
```
int CodaVuota(Coda *Q) {
  return (Q-> head == NULL);
}
```

```
void CreaCoda(Coda *Q){
   if(Q){
      Q->head = NULL;
      Q->tail = NULL;
   }
}
```

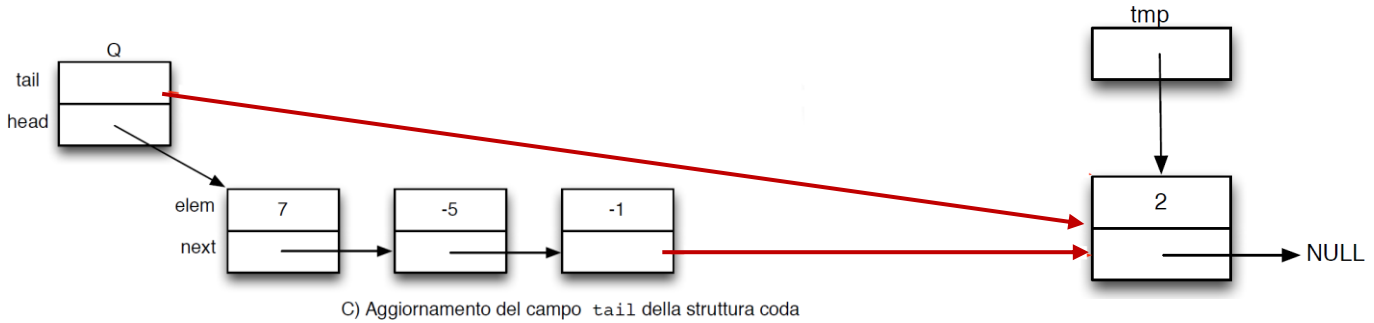
Inserimento di un elemento nella coda



A) Creazione del nuovo nodo da inserire (`tmp`)



B) Aggiornamento del campo `next` del nodo in coda



C) Aggiornamento del campo `tail` della struttura coda

Inserimento di un elemento nella coda

L'operazione di inserimento di un elemento nella struttura dati coda viene gestita come un inserimento in coda ad una lista.

Il nodo viene creato dalla stessa funzione definita per le liste, ovvero **CreaNodo_SL**

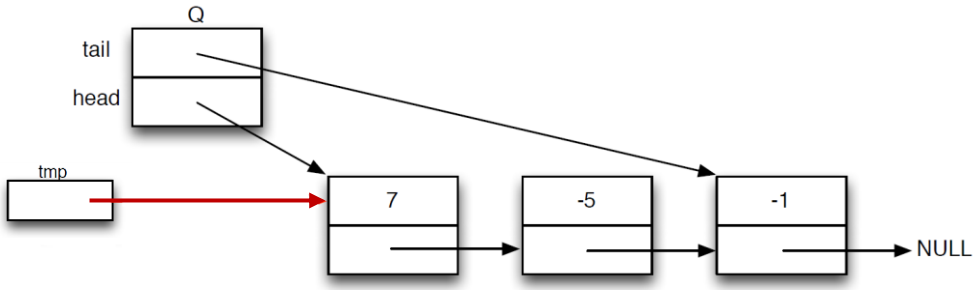
È necessario gestire in modo specifico due casi distinti:

- La coda inizialmente e vuota – sia il puntatore alla testa che alla coda puntano al nuovo nodo allocato
- La coda contiene almeno un elemento – l'operazione di inserimento è gestito come inserimento in coda in una lista concatenata. Non è necessario modificare il puntatore alla testa della coda

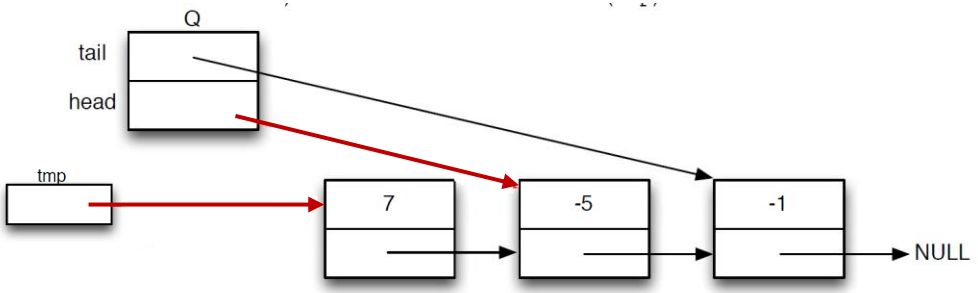
Inserimento di un elemento nella coda

```
void enqueue(Coda *Q, int dato){

   Node_SL *tmp;

   if(Q == NULL){
      return;
   } else if(CodaVuota(Q)){
      Q->head = CreaNodo_SL(dato);
      Q-> tail = Q->head;
      return;
   } else {
      tmp = CreaNodo_SL(dato);

      if(tmp != NULL) {
          Q->tail->next = tmp;
          Q->tail = tmp;
      }
      return;
   }
}
```

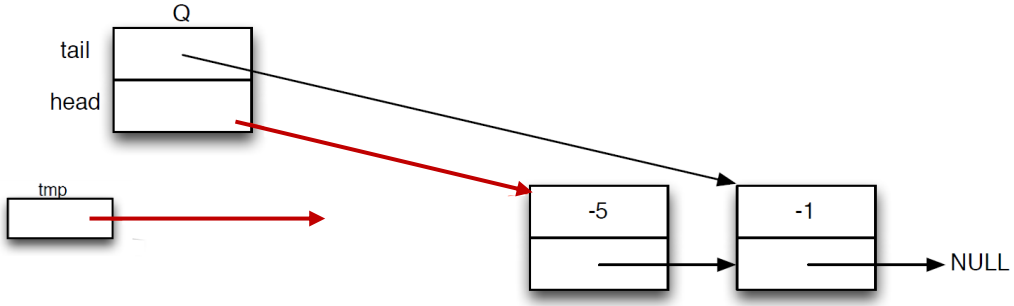
Eliminazione di un elemento dalla coda



A) Puntatore al nodo da rimuovere (`tmp`)



B) Aggiornamento puntatore alla testa della coda (`head`)



C) Deallocazione del nodo da rimuovere

Eliminazione di un elemento dalla coda

L'operazione di rimozione di un elemento nella struttura dati coda viene gestito come rimozione della testa di una lista.
È necessario gestire in modo specifico tre casi distinti:

La coda e vuota – la funzione termina (operazione non eseguita)

La coda contiene un solo elemento – è necessario aggiornare sia il puntatore alla testa che alla coda della lista

La coda contiene almeno due elementi – l'operazione è gestita come rimozione della testa in una lista concatenata. Non è necessario modificare il puntatore alla coda della lista

Eliminazione di un elemento dalla coda

```
void dequeue(Coda *Q, int *dato ){

   Node_SL *tmp;

   if(CodaVuota(Q) || dato == NULL ){
      return;
   } else if(Q->head == Q->tail) {
     *dato = Q->head->dato;
     free(Q->head);
     Q->head = Q->tail = NULL ;
     return;
   } else {
     tmp = Q->head;
     *dato = Q->head->dato;
     Q->head = Q->head->next;
     free(tmp);
     return;
   }
}
```

Funzioni aggiuntive sulle code

La seguente funzione permette di svuotare completamente una coda e di verificare se una coda e vuota.
L'implementazione della funzione di distruzione di una coda, rimuove iterativamente l'elemento in testa.

```
void LiberaCoda(Coda *Q)
{
   Node_SL *tmp;

   if(Q != NULL ) {

      while(Q->head != NULL ) {
         tmp = Q->head
         Q->head = Q->head->next;
         free(tmp);
      }
      Q->head = Q->tail = NULL ;
   }
}
```

Funzioni aggiuntive sulle code

L'operazione di accesso al primo elemento in coda e gestita come l'operazione di selezione dell'elemento in testa in una lista.

```
void LeggiInTesta(Coda *Q, int *dato){
    if(CodaVuota(Q) || dato == NULL){
        return;
    } else {
        *dato = Q->head->dato;
        return;
    }
}
```

Header file

```c
#ifndef _CODE_H
#define _CODE_H

#include <stdio.h>
#include <stdlib.h>
#include "liste.h"


typedef struct Coda {
Nodo_SL *head;
Nodo_SL *tail;
} Coda;


// Funzioni per la gestione delle code
void CreaCoda(Coda *Q);
void enqueue (Coda *Q, int dato);
void dequeue (Coda *Q, int *dato );
int CodaVuota(Coda *Q);
void LeggiInTesta (Coda *Q, int *dato);
void StampaCoda(Coda Q);
void LiberaCoda(Coda *Q);

#endif
```

Main file

```c
#include <stdio.h>
#include <stdlib.h>
#include "code.h"

int main(int argc, char *argv[])
{
   int i;
   int dato=0;
   Coda Q;

   CreaCoda(&Q);

   for(i=1; i<argc; i++)
      enqueue(&Q, atoi(argv[i]));

   StampaCoda(Q);

   dequeue(&Q, &dato);

   printf("\nElemento estratto: %d", dato);

   StampaCoda(Q);

   LiberaCoda(&Q);

   return 0;
}
```

```
D:\Lezioni\ProgrammazioneI\Esercizi>gcc Code.c liste.c CodeMain.c -o Code.exe

D:\Lezioni\ProgrammazioneI\Esercizi>Code.exe 3 5 8 6 4

[Coda]-> 3 -> 5 -> 8 -> 6 -> 4 -|
Elemento estratto: 3
[Coda]-> 5 -> 8 -> 6 -> 4 -|
```

La struttura dati stack

Uno **stack** (o pila) e una struttura dati astratta le cui modalità di accesso sono di tipo LIFO.

Una struttura dati pila supporta essenzialmente due sole operazioni

- **Push** – inserisce un oggetto in cima alla pila
- **Pop** – rimuove l'oggetto in cima alla pila e ne ritorna il valore

Nonostante la sua semplicità, la struttura dati astratta stack è utilizzata in numerosi contesti informatici.

Ad esempio, quasi tutti i linguaggi di alto livello utilizzano uno stack di record di attivazione per gestire la chiamata a funzione.

Una struttura dati di tipo stack può essere vista come una struttura dati di tipo lista che supporta un numero limitato di operazioni.

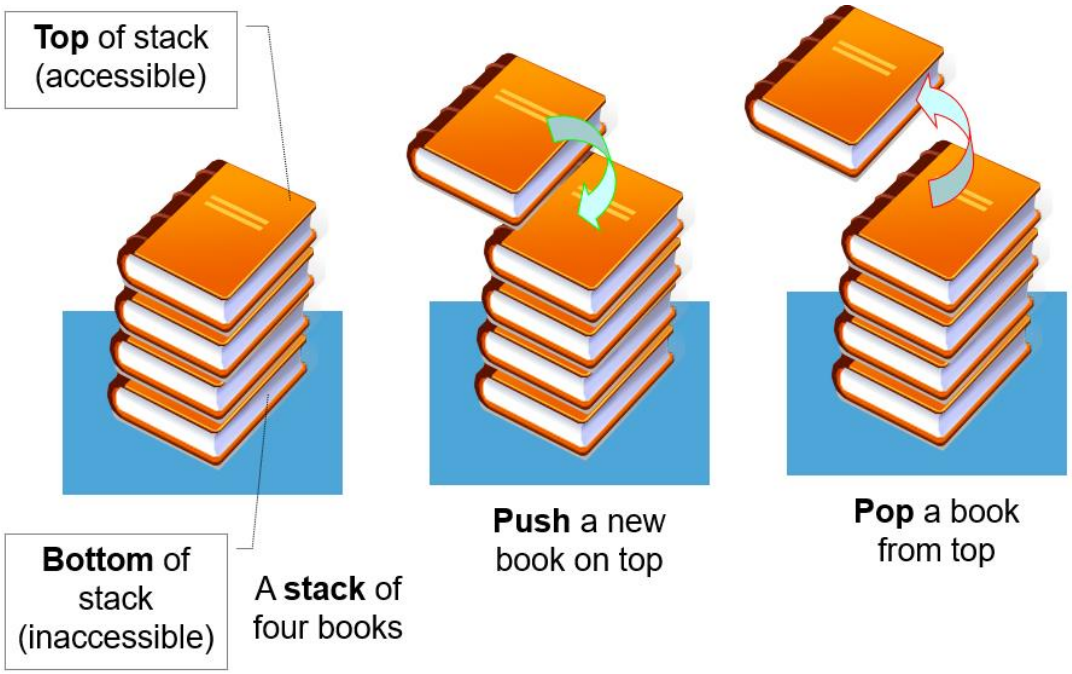
Le operazioni **Push** e **Pop** sono equivalenti a inserimento e rimozione in testa (o, equivalentemente, in coda) della lista.

La struttura dati stack

Esistono diversi approcci per implementare una struttura dati di tipo pila in C

- capacita illimitata – implementa una struttura dati di tipo lista
- capacita limitata – implementata con un array

Anche in questo caso ci concentriamo unicamente su implementazioni di pile che permettano di memorizzare interi (tipo di dato **int**).



Lo stack: la definizione di tipo

Una struttura dati di tipo **stack** con **capienza illimitata** può essere implementata tramite una lista concatenata.

Utilizziamo una libreria che implementa liste concatenate per implementare una struttura dati di tipo stack.

Le operazioni di inserimento e rimozione in testa sono efficienti per tutte le rappresentazioni concatenate viste

Le operazioni in cima allo stack possono essere implementate efficientemente come operazioni in testa ad una lista.

Il tipo di dato stack è equivalente al tipo di dato lista

È necessario includere il file di header in cui e definito il tipo di dato lista per poter definire il tipo di dato stack

```
# include " liste .h"

typedef Lista_SL Stack;
```

Lo stack: la definizione di tipo

Le seguenti funzioni ci permettono di:

- creare un oggetto di tipo stack. Inizialmente lo stack non contiene elementi
- Stampare tutti gli elementi nello stack
- Inserire un elemento nello stack
- Estrarre un elemento dallo stack
- Leggere un elemento in cima allo stack
- svuotare l'intero contenuto di un oggetto di tipo stack
- testare se lo stack è vuoto

```
void CreaStack(Stack *S);
void StampaStack(Stack S);
void Push(Stack *S, int dato);
void Pop(Stack *S, int *dato);
void Top(Stack *S, int *dato);
void LiberaStack(Stack *S);
int StackVuoto(Stack *S);
```

Lo stack: le funzioni di gestione

```
void CreaStack(Stack *S){
    if(S)
        S->next = NULL;
}

void StampaStack(Stack S){

    Nodo_SL *tmp = S.next;

    printf("\n[Stack]-> ");
    while(tmp && tmp->next){
        printf("%d -> ", tmp->dato);
        tmp = tmp->next;
    }

    if(tmp)
        printf("%d -|", tmp->dato);
}
```

Lo stack: le funzioni di gestione

```
void Push(Stack *S, int dato){
   if(S)
      InserisciInTesta_SL(S, dato);
}

void Pop(Stack *S, int *dato){
   if(S && S->next){
      *dato = S->next->dato;
      CancellaInTesta_SL(S);
   }
}

void Top(Stack *S, int *dato){
   if(S && S->next){
      *dato = S->next->dato;
   }
}
```

```
void LiberaStack(Stack *S){
   if(S)
      LiberaLista_SL(S);
}

int StackVuoto(Stack *S) {
   return (!S || S->next == NULL);
}
```

Main file

```c
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

int main(int argc, char *argv[]){
   int dato;
   int i;
   Stack S;

   for(i=1; i<argc; i++)
      Push(&S, atoi(argv[i]));

   StampaStack(S);

   Pop(&S, &dato);

   printf("\nElemento estratto: %d", dato);

   StampaStack(S);

   LiberaStack(&S);

   return 0;
}
```

```
D:\Lezioni\ProgrammazioneI\Esercizi>gcc Stack.c StackMain.c Liste.c -o Stack.exe

D:\Lezioni\ProgrammazioneI\Esercizi>Stack.exe 3 5 8 6 4

[Stack]-> 4 -> 6 -> 8 -> 5 -> 3 -|
Elemento estratto: 4
[Stack]-> 6 -> 8 -> 5 -> 3 -|
Elimino 6
Elimino 8
Elimino 5
Elimino 3
```

Esempio: bilanciamento delle parentesi

Un problema algoritmico che può essere facilmente risolto facendo uso di una struttura dati stack è quello di verificare il bilanciamento delle parentesi
in una stringa.

Assumiamo che la stringa possa contenere altre informazioni oltre alle parentesi: ci interessa unicamente verificare che le parentesi siano bilanciate.

Algoritmo per valutare il bilanciamento delle parentesi:

Scorriamo la stringa in input da sinistra verso destra

Se il carattere corrente non è una parentesi (aperta o chiusa), ignoriamo il carattere

Se il carattere corrente è una parentesi aperta, inseriamo nella pila tale carattere

Se il carattere corrente è una parentesi chiusa, rimuoviamo la cima dello stack e verifichiamo che sia una parentesi aperta di tipo compatibile con il carattere corrente. Se le due parentesi non sono compatibili, restituiamo false.

Al termine della procedura se lo stack è vuoto restituiamo true, altrimenti false.

Es.:

1 + 3*(4+2) – {4*[7-(3*6)]} (bilanciate)

1 + 3*(4+2) – {4*[7-(3*6)} (non bilanciate)

Esempio: bilanciamento delle parentesi

```
int balanced_parenthesis(char *str){
   int c;
   unsigned int i,n;
   Stack S;

   n = strlen(str);
   CreaStack(&S);

   for(i=0; i<n; i++){
      switch(str[i]){
         case '(':
            Push (&S, str[i]);
         break ;
         case '{':
            Push (&S, str[i]);
         break ;
         case '[':
            Push (&S, str[i]);
         break ;
      case ')':
         Pop(&S, &c);
         if(c != '(') {
            LiberaStack(&S);
            return 0;
         }
      break ;
      case '}':
         Pop(&S, &c);
         if(c != '{'){
            LiberaStack(&S);
            return 0;
         }
      break ;
      case ']':
         Pop(&S, &c);
         if(c != '['){
            LiberaStack (&S);
            return 0;
         }
      break ;
      }
}
   if(!StackVuoto(&S)){
        LiberaStack(&S);
        return 0;
     } else {
        LiberaStack(&S);
        return 1;
     }
}
```

Esempio: bilanciamento delle parentesi

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "stack.h"

int balanced_parenthesis(char *str);

int main(int argc, char *argv[])
{
   int risultato;

   risultato = balanced_parenthesis(argv[1]);

   if(risultato)
      printf("\nLeparentesi sono bilanciate\n");
   else
      printf("\nLeparentesi non sono bilanciate\n");
}
```

```
D:\Lezioni\ProgrammazioneI\Esercizi>gcc Stack.c ParentesiBilanciate.c Liste.c -o ParentesiBilanciate.exe

D:\Lezioni\ProgrammazioneI\Esercizi>ParentesiBilanciate.exe (2+3)*[4+{3*2}]

Leparentesi sono bilanciate

D:\Lezioni\ProgrammazioneI\Esercizi>ParentesiBilanciate.exe (2+3)*[4+{3*2}

Elimino 91
Leparentesi non sono bilanciate
```

Stack con array

Per alcuni problemi algoritmici che fanno uso di una struttura dati di tipo stack il numero massimo di oggetti da memorizzare può essere determinato a priori

Implementazione efficiente tramite rappresentazione dello stack con array

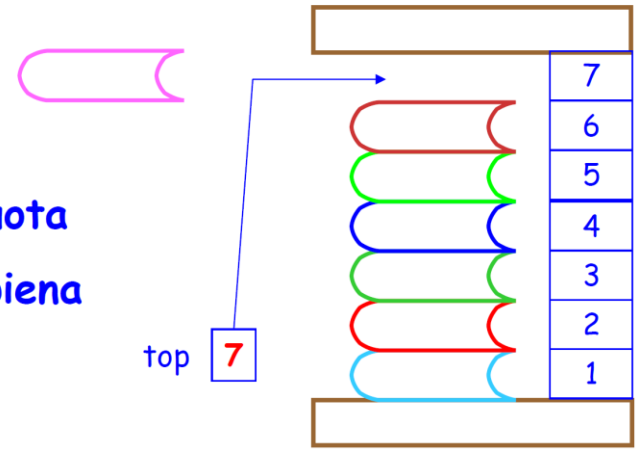
Caratteristiche specifiche della libreria per stack con capienza limitata:

- La funzione **CreaStack**() richiede come argomento la dimensione dello stack
- Aggiungiamo una nuova funzione di libreria per determinare se lo stack è pieno

L'indice deve tener conto di quanti elementi ci sono nello stack.
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



Stack con array

```
typedef struct AStack {
   unsigned int size;
   unsigned int last;
   int *dato;
} AStack;

void CreaAStack(AStack *S, unsigned int n);
void LiberaAStack (AStack *S);
int AStackVuoto(AStack *S);
int AStackPieno(AStack *S);
int APush(AStack *S, int dato);
int APop(AStack *S, int *dato);
int ATop(AStack *S, int *dato);
```

Stack con array

```
void CreaAStack(AStack *S, unsigned int n) {
  S->size = 0;
  S->last = 0;
  S->dato = (int *)malloc(n*sizeof(int));
  if(S->dato != NULL)
    S->size = n;
  return;
}

void LiberaAStack(AStack *S) {
  if(S != NULL) {
    free (S->dato);
    S->dato = NULL ;
    S->last = 0;
    S->size = 0;
  }
}
```

```
int AStackVuoto(AStack *S) {
  return S == NULL || S->last == 0;
}

int AStackPieno(AStack *S) {
  return S == NULL || S->last == S->size ;
}
```

Stack con array

```
int APush(AStack *S, int dato) {
  if(AStackPieno(S)) {
    return 1;
  } else {
    S->dato[S-> last++] = dato;
    return 0;
  }
}

int APop(AStack *S, int *dato) {
  if(AStackVuoto(S) || dato == NULL ) {
    return 1;
  } else {
    *dato = S->dato[S->last-- - 1];
    return 0;
  }
}
```

```
int ATop(AStack *S, int *dato){
  if(AStackVuoto(S) || dato == NULL ){
    return 1;
  } else {
    *dato = S-> dato[S-> last - 1];
    return 0;
  }
}

void StampaAStack(AStack S){
   int i=0;

   printf("\n[Stack]-> ");
   if(AStackVuoto(&S)==0){
     for(i=S.last-1; i>0; i--)
       printf("%d -> ", S.dato[i]);

     printf("%d -|", S.dato[0]);
   }
}
```

Main file

```c
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"


int main(int argc, char *argv[]){
   int dato;
   int i;
   AStack S;

   CreaAStack(&S, 100);

   for(i=1; i<argc; i++)
      APush(&S, atoi(argv[i]));

   StampaAStack(S);

   APop(&S, &dato);

   printf("\nElemento estratto: %d", dato);

   StampaAStack(S);

   LiberaAStack(&S);

   return 0;
}
```

```
D:\Lezioni\ProgrammazioneI\Esercizi>gcc Stack.c StackMain.c Liste.c -o Stack.exe

D:\Lezioni\ProgrammazioneI\Esercizi>Stack.exe 3 5 8 6 4

[Stack]-> 4 -> 6 -> 8 -> 5 -> 3 -|
Elemento estratto: 4
[Stack]-> 6 -> 8 -> 5 -> 3 -|
```

Esercizi

Implementare una funzione che inverta l'ordine degli elementi in una coda facendo uso di uno stack.