

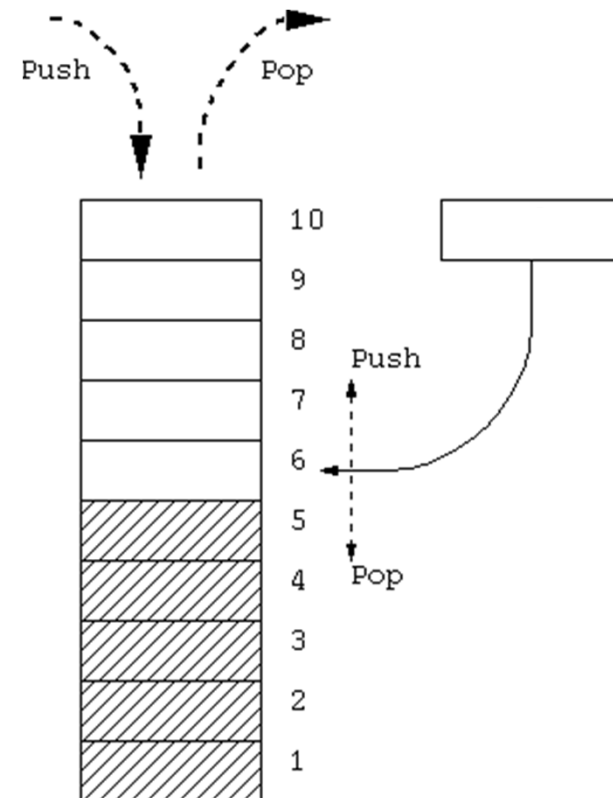
# ADT STACK (PILA)

---

**Collezione di elementi dello stesso tipo (multi-insieme) gestito con politica LIFO (Last-In -- First-Out): il primo elemento entrato è l'ultimo a uscire**

Svariate applicazioni del concetto di stack:

- **memoria** usata dal sistema operativo per **record attivazione**
- ogni volta che è opportuna gestione LIFO di item (manipolazione di oggetti, ...)

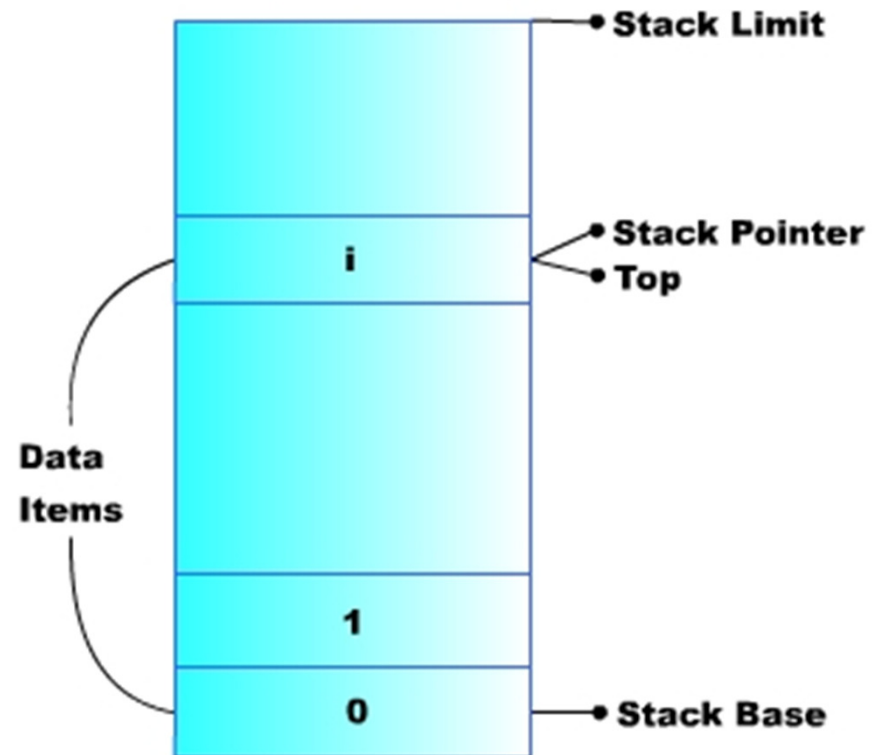


# ADT STACK (PILA)

---

Come ogni *tipo di dato astratto*, **STACK** è definito in termini di:

- *dominio* dei suoi elementi (dominio base)
- *operazioni* (*costruzione*, *selezione*, ...) e *predicati* sul tipo **STACK**



# IL COMPONENTE STACK (PILA)

---

Formalmente:

$$\text{stack} = \{ D, \mathcal{F}, \Pi \}$$

dove:

- **D** (il dominio base) può essere **qualunque**

- **$\mathcal{F}$**  (funzioni)= { **push**, **pop**, **newStack** }

**push:**  $D \times \text{stack} \rightarrow \text{stack}$  (inserimento)

**pop:**  $\text{stack} \rightarrow D \times \text{stack}$  (estrazione)

**newStack:**  $\rightarrow \text{stack}$  (costante stack vuoto)

- **$\Pi$**  (predicati)= { **isEmptyStack**, **isFullStack** }

**isEmptyStack:**  $\text{stack} \rightarrow \text{boolean}$  (test di stack vuoto)

**[isFullStack:**  $\text{stack} \rightarrow \text{boolean}$  (test di stack pieno) ]

# CONSIDERAZIONI

---

Per utilizzare istanze del *tipo di dato astratto stack*:

- è necessario che il programmatore **crei espressamente uno stack** prima di poterlo usare
- è possibile definire **più stack distinti**
- lo stack su cui si opera figura **esplicitamente** fra i parametri delle operazioni

# Articolazione del progetto (1)

---

Due file per il tipo *element* (*element.h*, *element.c*)

Due file per il tipo *stack* (*stack.h*, *stack.c*)

<i>operazione</i>	<i>descrizione</i>
<b>push</b> : $D \times \text{stack} \rightarrow \text{stack}$	<b>inserisce un elemento</b> nello stack dato (modificando lo stack)
<b>pop</b> : $\text{stack} \rightarrow D \times \text{stack}$	<b>estrae (e rimuove) un elemento</b> dallo stack dato (modificando lo stack)
<b>newStack</b> : $\rightarrow \text{stack}$	<b>crea e restituisce uno stack vuoto</b>
<b>isEmptyStack</b> : $\text{stack} \rightarrow \text{boolean}$	Restituisce <b>vero</b> se lo stack dato è vuoto, <b>falso</b> altrimenti

## Articolazione del progetto (2)

---

Idealmente, uno stack ha ampiezza illimitata  
→ può essere vuoto, ma non *pieno*

Tuttavia, alcune *implementazioni* potrebbero porre ***limiti*** all'effettiva dimensione di uno **stack** → **ulteriore primitiva:**

<i>operazione</i>	<i>descrizione</i>
<b>isFullStack:</b> stack → boolean	Restituisce <b>vero</b> se lo stack dato è pieno, <b>falso</b> altrimenti

# ADT stack (soluzione 1 – vettore con indice)

---

Possibili implementazioni per stack:

## 1) un vettore + un indice

2) tramite allocazione dinamica di strutture (come nelle liste)

File header nel caso “vettore + indice”:

```
#include "element.h"
#define MAX 1024

typedef struct {
    element val[MAX];
    int sp; } stack;

void push(element, stack);
element pop(stack);
stack newStack(void);
boolean isEmptyStack(stack);
boolean isFullStack(stack);
```

# ADT stack (soluzione 1 – vettore con indice)

**Problema:** le funzioni `push()` e `pop()` *devono modificare lo stack* → *impossibile passare lo stack per valore*

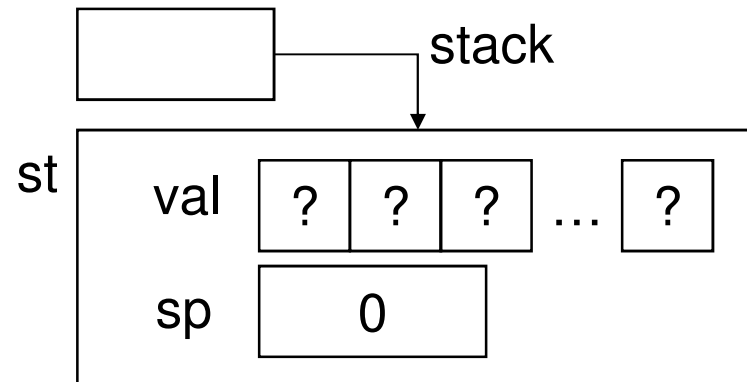
→ Occorre passaggio *per riferimento*

Due scelte:

- in modo *visibile* (sconsigliabile)
- in modo *trasparente*

A questo fine, occorre definire **tipo stack come puntatore** (a una struttura). Nuovo header nel caso “vettore + indice”:

```
typedef struct {  
    element val[MAX];  
    int sp;  
} st;  
typedef st *stack;
```





# ADT stack (soluzione 1 – vettore con indice)

---

```
#include <stdio.h>
#include "stack.h"      /* include la typedef */
#include "stdlib.h"

stack newStack(void) {
    stack s = (stack) malloc(sizeof(st));
    s -> sp = 0;
    return s;
}

boolean isEmptyStack(stack s) {
    return ((s->sp) == 0);
}

boolean isFullStack(stack s) {
    return ((s->sp) == MAX);
}
```

# ADT stack (soluzione 1 – vettore con indice)

---

```
void push(element e, stack s) {
    if ( !isFullStack(s) ) {
        s -> val[s->sp] = e;
        s->sp = s->sp + 1;
    }
    else
        perror("Errore: stack pieno");
}

element pop(stack s) {
    if ( !isEmptyStack(s) ) {
        s->sp = s->sp - 1;
        return s->val[s->sp];
    }
    else { perror("Errore: stack vuoto");
        exit(-1);
        /* che cosa si potrebbe fare altrimenti? */
    } }
}
```

# ADT stack (soluzione 2 – allocazione dinamica)

---

Possibili implementazioni per stack:

1) un vettore + un indice

**2) tramite allocazione dinamica di strutture**  
(come nelle liste)

File header in questo caso:

```
#include "element.h"
```

```
typedef struct stackN {  
    element value;  
    struct stackN *next;  
} stackNode;
```

```
typedef stackNode *stack;
```

# ADT stack (soluzione 2 – allocazione dinamica)

---

Problema: le funzioni `push()` e `pop()` *devono modificare lo stack* → *impossibile passare lo stack per valore*

→ Anche in questa soluzione occorre *passaggio per riferimento*

Lasciamo invariato header:

```
typedef struct stackN {  
    element value;  
    struct stackN *next;  
} stackNode;  
typedef stackNode *stack;
```

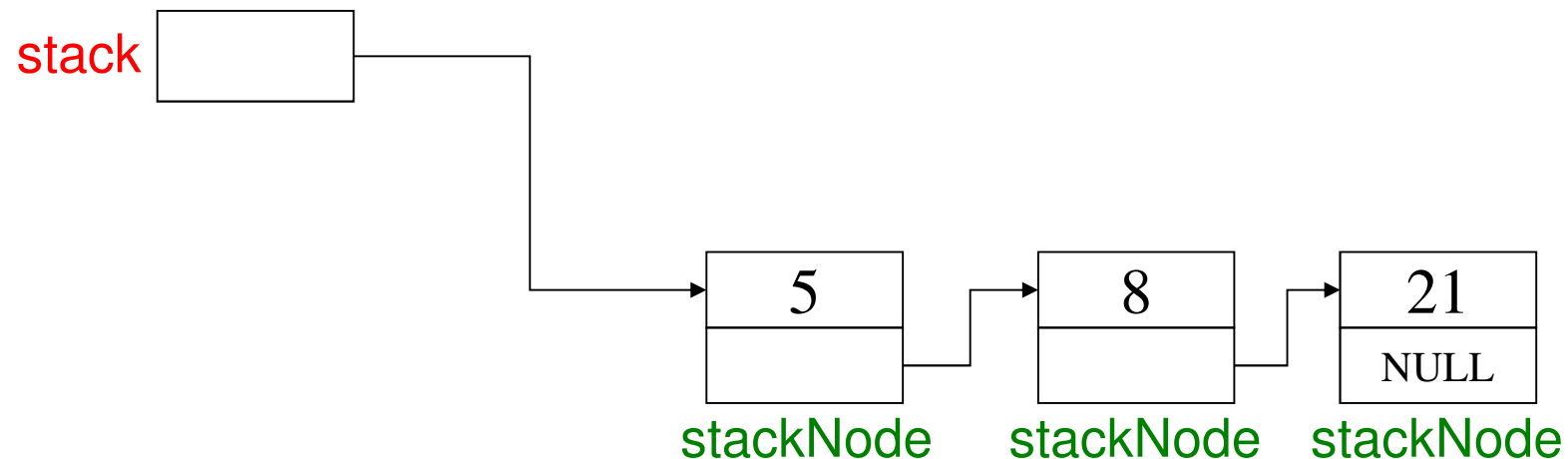
*Ma ricordiamoci che `push()` e `pop()` dovranno ricevere come parametri attuali degli indirizzi puntatori a stack*

# ADT stack (soluzione 2 – allocazione dinamica)

---

```
typedef struct stackN {  
    element value;  
    struct stackN *next;  
} stackNode;
```

```
typedef stackNode *stack;
```



# ADT stack (soluzione 2 – allocazione dinamica)

---

```
stack newStack(void) {  
    return NULL;  
}
```

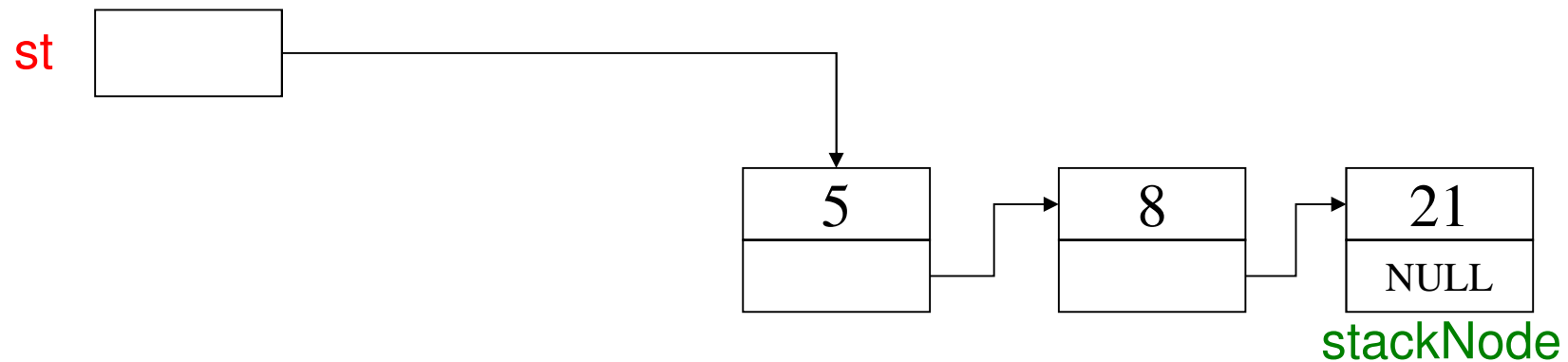
```
boolean isEmptyStack(stack s) {  
    return (s == NULL);  
}
```

```
boolean isFullStack(stack s) {  
    return 0;  
    /* nel caso di nessuna limitazione fisica alla  
    dimensione dello stack */  
}
```

# ADT stack (soluzione 2 – allocazione dinamica)

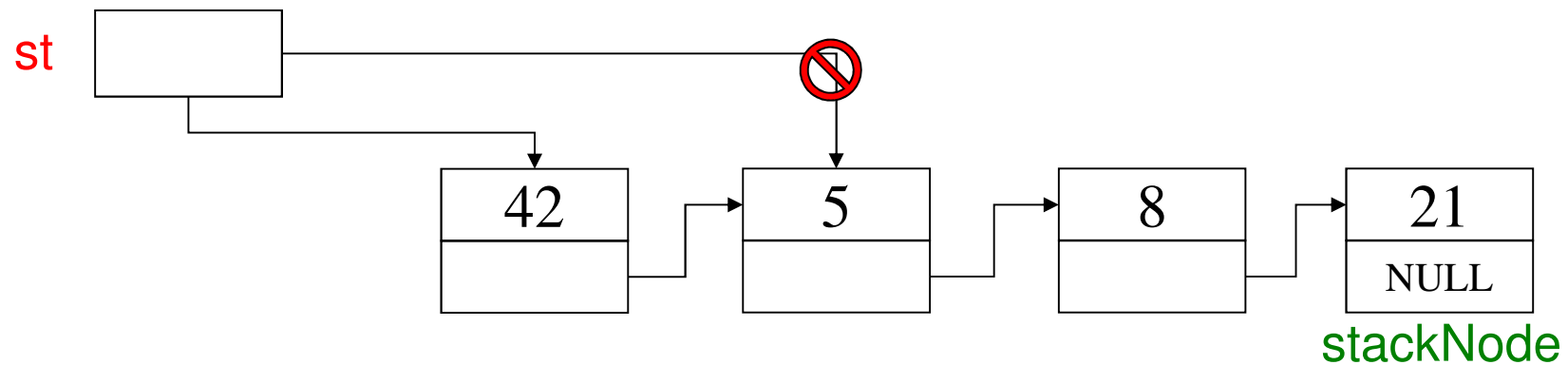
Esecuzione della funzione push(...)

Prima:



```
...  
push(42, &st);  
...
```

Eseguendo l'istruzione:



# ADT stack (soluzione 2 – allocazione dinamica)

---

```
void push(element e, stack *s) {
    stack newNode;

    if ( !isFullStack(*s) ) {
        newNode = (stack)
            malloc(sizeof(stackNode));

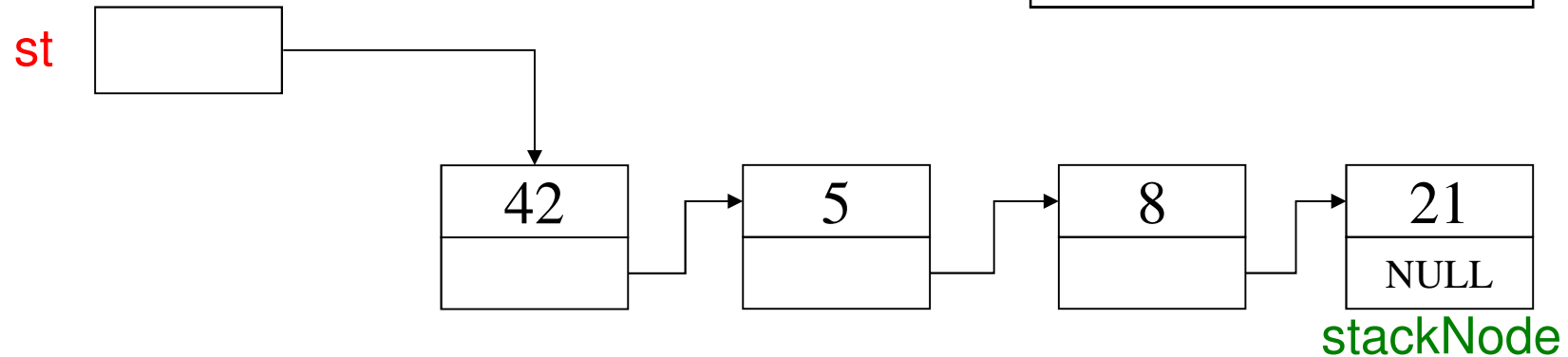
        newNode->value = e;
        newNode->next = *s;
        *s = newNode;
    }
    else
        perror("Errore: stack pieno");
}
```



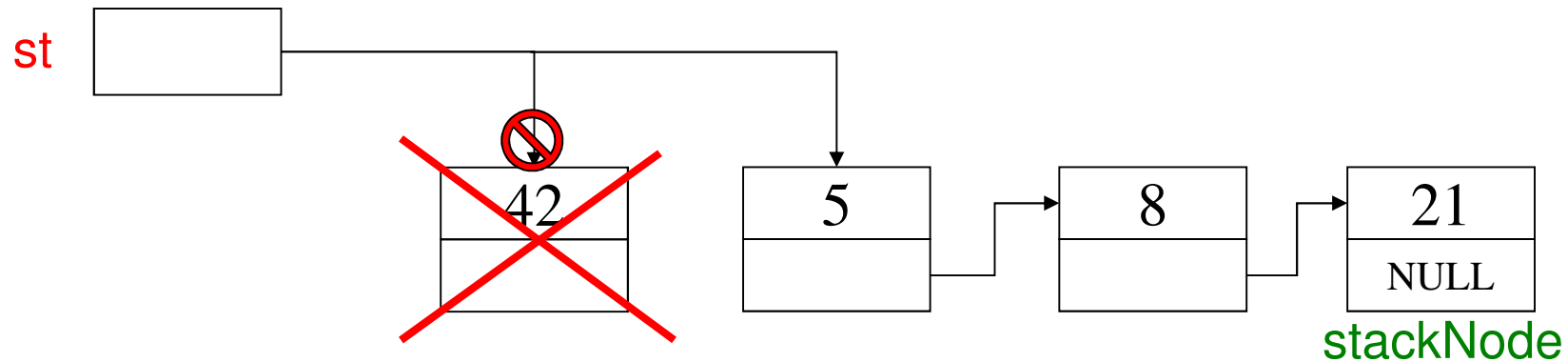
# ADT stack (soluzione 2 – allocazione dinamica)

Esecuzione della funzione **pop(...)**

Prima:



Eseguendo l'istruzione:



# ADT stack (soluzione 2 – allocazione dinamica)

---

```
element pop(stack *s) {
    element result;
    stack oldNode;

    if ( !isEmptyStack(*s) ) {
        oldNode = *s;
        result = oldNode->value;
        *s = oldNode->next;
        free(oldNode); /* operazione distruttiva!!! */
        return result;
    }
    else {
        perror("Errore: stack vuoto");
        exit(-1);
    }
}
```

# Esempio di main() che usa STACK (di interi)

---

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

int main(void) {
    stack s1 = newStack(); // creazione di uno stack vuoto
    int choice; /* scelta menu utente */
    int value; /* input utente */

    instructions(); /* mostra il menu */
    printf("? ");
    scanf("%d", &choice);

    /* while user does not enter 3 */
    while (choice!=3) {
```

# Esempio di main() che usa STACK (di interi)

---

```
switch (choice) {
    /* inserisci un valore nello stack */
    case 1:
        printf("Enter an integer: ");
        scanf("%d", &value);
        push(value, &s1);
        printStack(s1);
        break;

    /* estrai un valore dallo stack */
    case 2:
        /* se lo stack non è vuoto */
        if ( !isEmpty(s1) ) {
            printf( "The popped value is %d.\n",
                pop(&s1) ); }
        printStack(s1);
        break;
```

# Esempio di main() che usa STACK (di interi)

---

```
default:
    printf("Invalid choice.\n\n");
    instructions();
    break;
} /* end switch */
printf("? ");
scanf("%d", &choice);
} /* end while */
printf("End of run.\n");
return 0; /* terminazione con successo */
}
```

# Esempio di main() che usa STACK (di interi)

---

```
/* mostra le istruzioni all'utente */  
void instructions(void)  
{  
    printf("Enter choice:\n 1 to push a value on the  
    stack\n 2 to pop a value off the stack\n 3 to  
    end program\n" );  
}
```

# Esempio di main() che usa STACK (di interi)

---

```
void printStack(stack s)
{
    /* se lo stack è vuoto */
    if (isEmptyStack(s)) {
        printf( "The stack is empty.\n\n" );
    }
    else {
        printf("The stack is:\n");
        while (!isEmptyStack(s)) {
            printf("%d --> ", s->value);
            s = s->next;
        }

        printf("NULL\n\n");
    }
}
```

e se avessi  
usato pop () ?

# Esempio di screenshot a runtime (1)

---

```
Enter choice:  
1 to push a value on the stack  
2 to pop a value off the stack  
3 to end program  
? 1  
Enter an integer: 5  
The stack is:  
5 --> NULL  
  
? 1  
Enter an integer: 6  
The stack is:  
6 --> 5 --> NULL
```



## Esempio di screenshot a runtime (2)

---

```
? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL
```

```
? 2
The popped value is 4.
The stack is:
6 --> 5 --> NULL
```

```
? 2
The popped value is 6.
The stack is:
5 --> NULL
```

```
? 2
The popped value is 5.
The stack is empty.
```

## Esempio di screenshot a runtime (3)

---

? 2

The stack is empty.

? 4

Invalid choice.

Enter choice:

1 to push a value on the stack

2 to pop a value off the stack

3 to end program

? 3

End of run.

# IL COMPONENTE CODA FIFO (FIFOQueue)

---

Una coda è un *contenitore di elementi gestito con politica FIFO* (First-In -- First-Out): **il primo elemento entrato è anche il primo a uscire**

- Le operazioni sono simili a quelle di uno stack: in particolare, *enqueue()* inserisce un elemento, e *dequeue()* lo estrae (rimuovendolo)

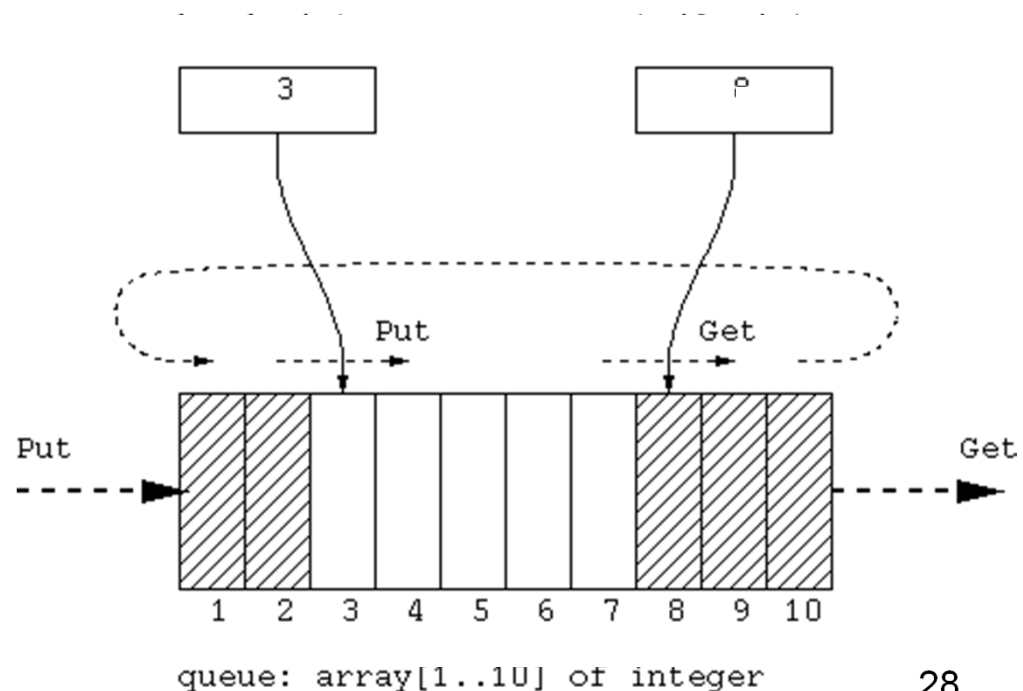
**Implementazione basata su vettore o su lista:** a differenza dello stack, per gestire la politica FIFO conviene avere accesso *sia al primo elemento (estrazione) sia all'ultimo (inserimento)*

# ADT CODA

Numerose applicazioni delle code in computer science/engineering:

- **accesso a risorse condivise in mutua esclusione** (coda di accesso alla CPU, spooling di stampa, ...)
- **code di pacchetti** nei dispositivi di rete per l'instradamento (router)

*Anche code con  
priorità  
differenziate*



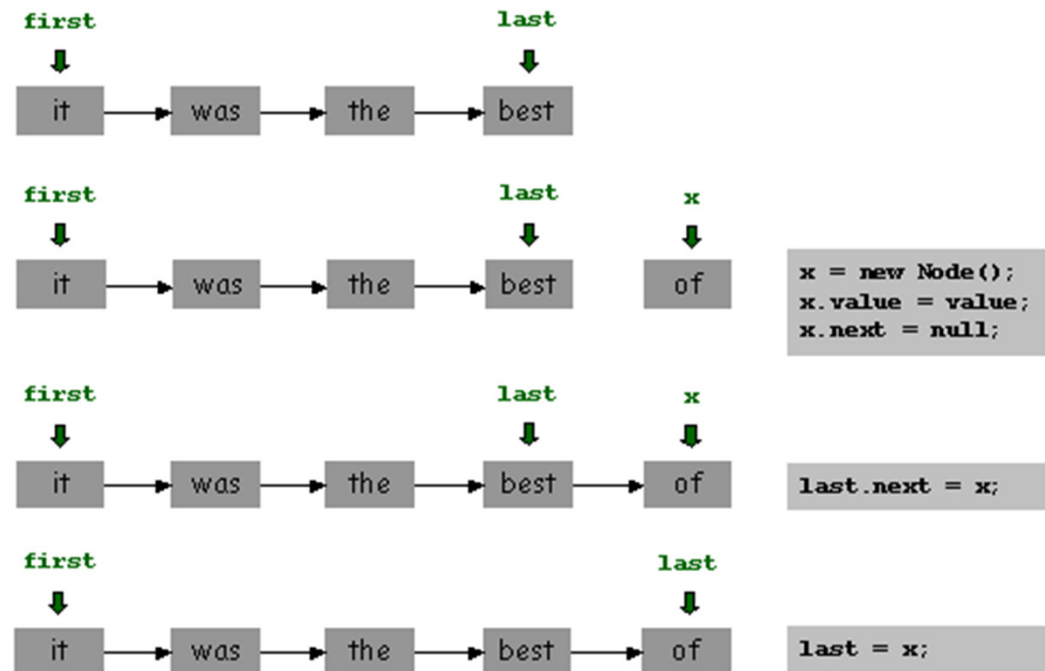
# IL COMPONENTE CODA FIFO (FIFOQueue)

Formalmente:

$\text{FIFOQueue} = \{ D, \mathcal{S}, \Pi \}$

dove:

- $D$  (dominio-base) può essere qualunque
- $\mathcal{S}$  (funzioni) = { **createEmptyQueue**, **enqueue**, **dequeue** }
- $\Pi$  (predicati) = { **isEmptyQueue** }



# IL COMPONENTE CODA FIFO (`FIFOQueue`)

---

Un file per il tipo *element* (`element.h`)

Due file per il tipo *FIFOQueue*

(`FIFOQueue.h`, `FIFOQueue.c`)

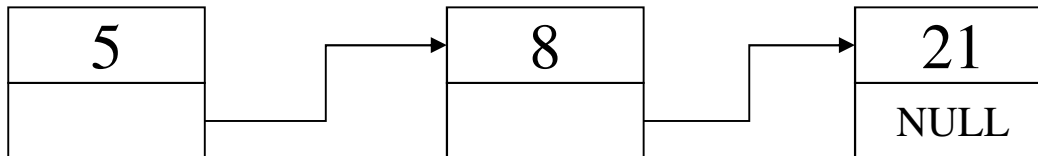
<i>operazione</i>	<i>descrizione</i>
<b>enQueue</b> : $D \times \text{FIFOQueue} \rightarrow \text{FIFOQueue}$	<i>inserisce un elemento</i> nella coda data ( <i>modificandola</i> )
<b>deQueue</b> : $\text{FIFOQueue} \rightarrow D \times \text{FIFOQueue}$	<i>estrae (e rimuove) un elemento</i> dalla coda data ( <i>modificandola</i> )
<b>createEmptyQueue</b> : $\rightarrow \text{FIFOQueue}$	crea e restituisce una coda vuota
<b>isEmptyQueue</b> : $\text{FIFOQueue} \rightarrow \text{bool}$	Restituisce <i>vero</i> se la coda data è vuota, <i>falso</i> altrimenti

# IMPLEMENTAZIONE (FIFOQueue)

---

Possibili implementazioni:

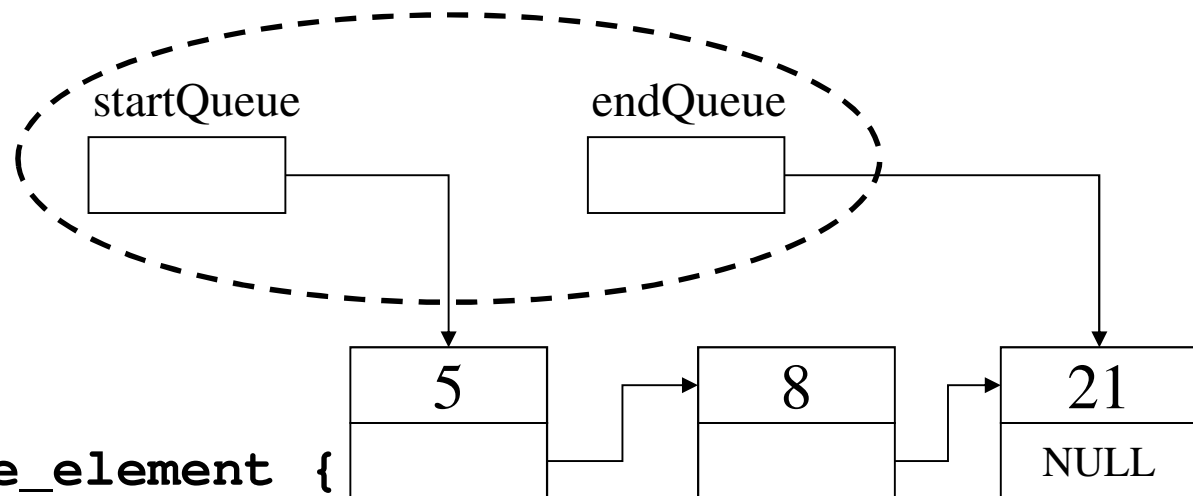
- 1) Usando un vettore + due indici (cattivo uso della memoria, limiti alla dimensione massima, ...)
- 2) Usando una rappresentazione collegata analoga al caso delle liste



```
typedef struct queue_element {  
    element value;  
    struct queue_element * next;  
} queueNode;
```

# IMPLEMENTAZIONE (FIFOQueue)

A differenza dello stack, per gestire la politica FIFO conviene avere accesso *sia al primo elemento (estrazione) sia all'ultimo (inserimento)*

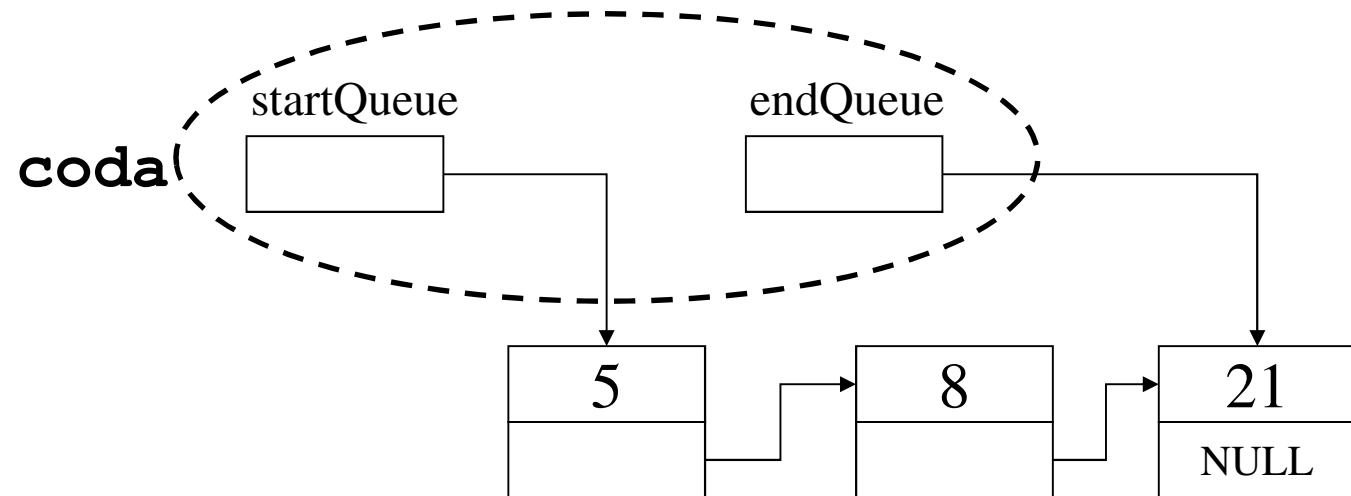


```
typedef struct queue_element {  
    element value;  
    struct queue_element * next;  
} queueNode;
```

```
typedef queueNode *startQueue;  
typedef queueNode * endQueue;
```



# IMPLEMENTAZIONE (FIFOQueue)



Esercizio:

si realizzi una implementazione alternativa con **una unica variabile di tipo coda**, struttura che contenga i due puntatori **startQueue** e **endQueue**.

Andrà passata **per valore o per riferimento** nelle operazioni elementari?

```
typedef struct {  
    queueNode *startQueue;  
    queueNode *endQueue} coda;
```

# IMPLEMENTAZIONE (FIFOQueue)

---

```
void createEmptyQueue (startQueue * start,  
endQueue * end)  
{  
    *start = NULL;  
    *end = NULL;  
    return;  
}
```

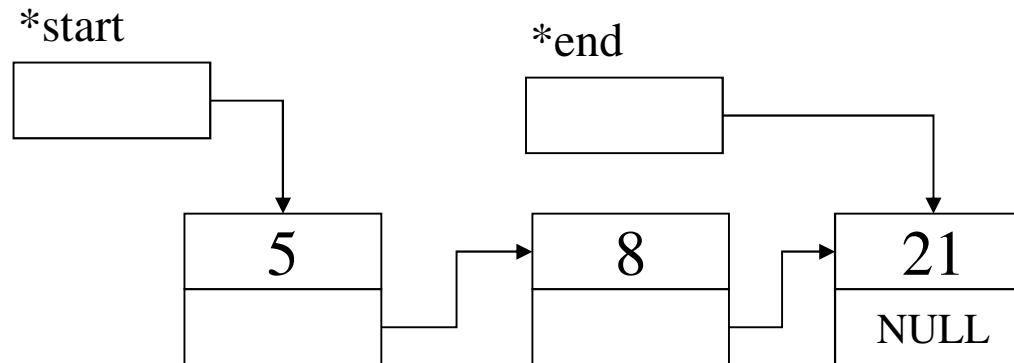


```
int isEmptyQueue (startQueue aQueue)  
{    return aQueue == NULL; }
```

**NOTA:** Quando creo una coda vuota, devo modificare i puntatori *start* ed *end* → devo ***passarli per riferimento***

# IMPLEMENTAZIONE – enqueue()

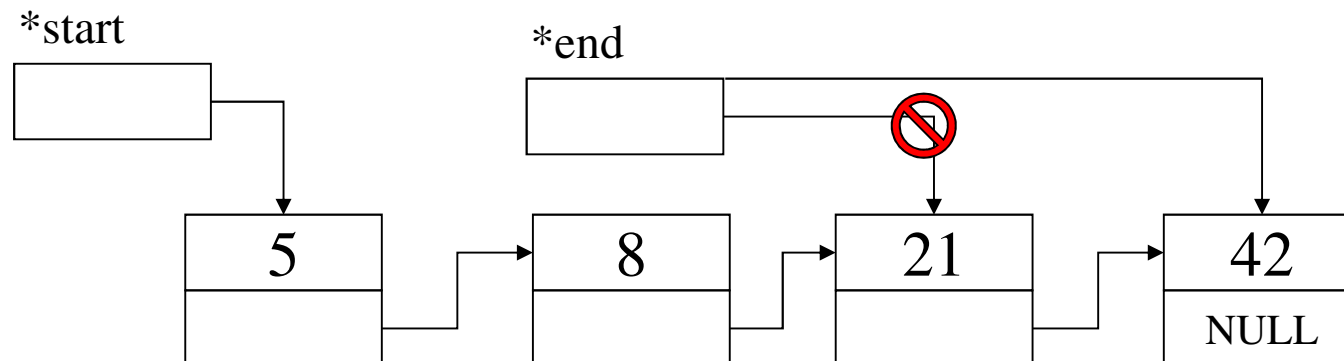
Inserisco gli ultimi arrivati in fondo alla coda → devo modificare l'ultimo puntatore. Nota che se la coda è vuota, devo modificare anche il primo puntatore



...

```
enqueue(42, &start, &end);
```

...



# IMPLEMENTAZIONE – enqueue()

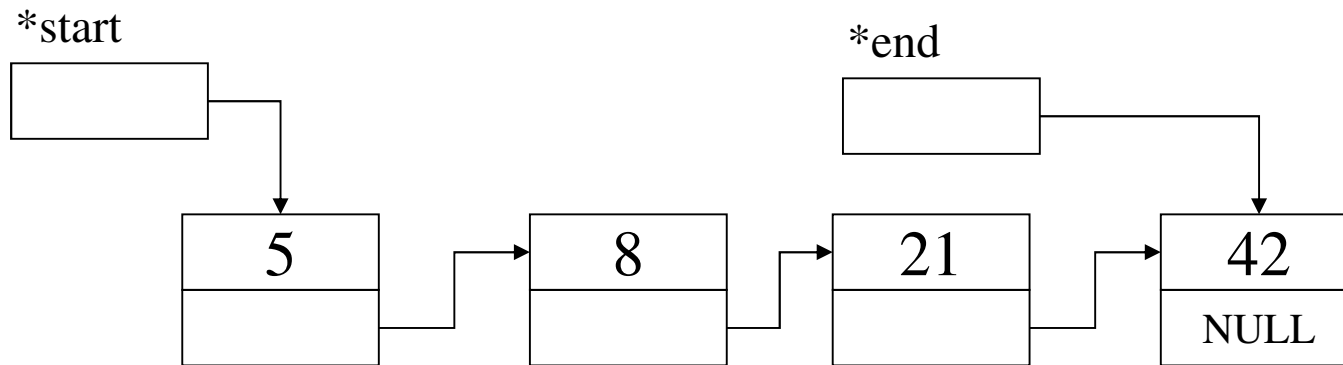
---

```
void enqueue(element el, startQueue *start, endQueue *end)
{
    queueNode *newNodePtr;
    newNodePtr = (queueNode *) malloc(sizeof(queueNode));

    if (newNodePtr != NULL) {
        newNodePtr->value = el;
        newNodePtr->next = NULL;
        if (isEmptyQueue(*start)) /* coda vuota */
            *start = newNodePtr;
        else (*end)->next = newNodePtr;
        *end = newNodePtr;
    }
    else printf("procedure enqueue: Element not inserted.
        A problem occurred while allocating new
        memory.\n");
    return;
}
```

# IMPLEMENTAZIONE – deQueue()

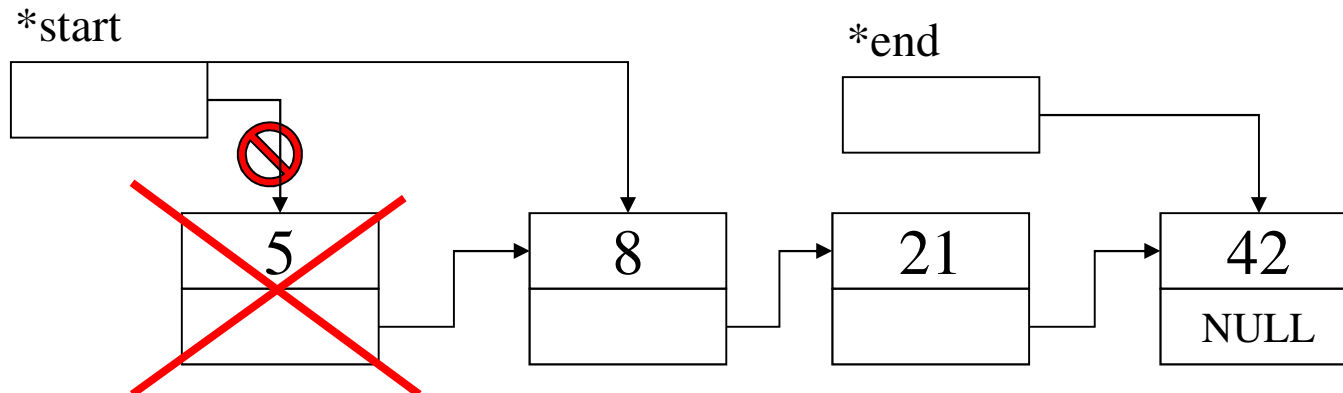
**Restituisco il primo elemento** in cima alla coda → devo modificare il primo puntatore. Devo anche **deallocare** la memoria corrispondente



...

```
result = deQueue(&start, &end); // restituisce 5...
```

...



# IMPLEMENTAZIONE - deQueue()

---

```
element deQueue(startQueue * start, endQueue * end)
{
    element result;
    startQueue temp;

    if (isEmptyQueue(*start)) {
        printf("function deQueue: Fatal error,
            required an element from an empty
            queue...\n");
        exit(-1);
    }
    else {result = (*start)->value;
        temp = *start;
        *start = (*start)->next;
        if (isEmptyQueue(*start)) *end = NULL;
        free(temp);
    }
    return result;
}
```