

Lezione 8

Appunti di Davide Scarlata 2024/2025


 **Prof:** Michele Garetto

 **Mail:**  michele.garetto@unito.it

 **Corso:**  C

 [Moodle corso C](#)

 [Moodle Lab matricole dispari](#)

** Data 29/03/2025

Stack (Pila)

Una pila è una struttura dati che segue il principio **LIFO (Last In First Out)**, ovvero l'ultimo elemento inserito è il primo a essere rimosso.

(Si lavora sempre sulla cima — pensa alla pila di piatti 🍽️)

Operazioni sullo stack

- `push` : inserimento di un elemento alla cima dello stack
- `pop` : rimozione di un elemento dalla cima dello stack
- `top` : prima cella libera o ultima cella occupata
- `empty` : verifica se lo stack è vuoto
- `full` : verifica se lo stack è pieno
- `peek` : visualizza l'elemento in cima alla pila senza rimuoverlo

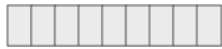
Firma delle funzioni

```
void push(tipo elemento , stack *s);
tipo pop(stack *s);
int empty(stack s);
int full(stack s);
tipo peek(stack s);
void init(stack *s);
```

tipi di implementazioni

IMPLEMENTAZIONI

TRAMITE ARRAY

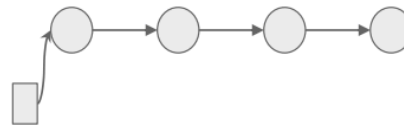


L'array ha una capienza predefinita

L'accesso avviene tramite indice

Serve una variabile (top) che ci dica fin dove l'array è pieno. Ogni push/pop deve aggiornare top

TRAMITE LISTA LINKATA



La lista può crescere senza limiti

push/pop = inserzione/estrazione dalla testa

L'elemento in coda è il primo a essere stato inserito (il più "profondo" nello stack)

Implementazione delle funzioni

```
int empty(stack* s)
{
return !(*s); // se stack vuoto (*s) == NULL
}

int full(stack* s) {
return 0;
// in questa implementazione lo stack non ha un numero massimo di elementi
}

void push(tipo elemento, stack* s) {
    if (s) {
        ins_testa(s, elemento); // inserisco in testa
    }
}

int pop(stack* s) {
    if (s) {
        if (!empty(s)) {
            top = delate_testa(s); // restituisce nodo in testa
            res = top->dato; // restituisce il dato della testa
        }
        else res = novall; // se non c'è nulla restituisco null
        //(novall=-1 per esempio)
    }
    return res;
}
```

stack con array

```
#define MAX 100 // dimensione massima dello stack
typedef struct {
    int top; // indice dell'elemento in cima alla pila
    tipo dati[MAX]; // array di dati
} stack;
```

```
void init(stack* s) {
    if (s) {
        s->top = 0; // inizializzo lo stack
    }
    else {
        printf("Errore: stack non inizializzato\n");
    }
}
```

```
int empty(stack* s) {
    return (s->top == 0); // se stack vuoto top == 0
}

int full(stack* s) {
    return (s->top == MAX); // se stack pieno top == MAX
}
```

```
void push(int el , stack *s){
    if(!full(*s)) {
        s->dato[s->top] = el; // inserisco in cima
        s->top++; // incremento il top
    }
    else {
        printf("Errore: stack pieno\n");
    }
}
```

```
int pop(stack *s) {
    int res = novall;
    if(!empty(*s)) {
```

```

        res = s->dato[s->top-1]; // restituisco il dato in cima
        s->top--; // decremento il top
    }
    else {
        printf("Errore: stack vuoto\n");
    }
    return res;

```

Queue (Coda)

Una **coda** è una struttura dati che segue il principio **FIFO (First In First Out)**, ovvero il primo elemento inserito è il primo a essere rimosso.

Si lavora sempre sulla **testa** (per rimuovere) e sulla **coda** (per inserire).

Operazioni sulla coda

- `enqueue` : inserimento di un elemento in coda
- `dequeue` : rimozione di un elemento dalla testa
- `empty` : verifica se la coda è vuota
- `full` : verifica se la coda è piena

Firma delle funzioni

```

void enqueue(tipo elemento , queue *q)

tipo(es int) dequeue(queue *q)

int empty(queue q)

int full(queue q)

tipo peek(queue q)

void init(queue *q)

```

implementazione con array

```

int successore(int i , int n) {
    return (i+1)%n; // restituisce il successore di i
}
typedef struct {
    char dato[max];
    int head; // indice dell'elemento in testa (punto da cui si toglie)
    int tail; // indice dell'elemento in coda (punto da cui si aggiunge)
    int num; // numero di elementi nella coda
} queue;

```

funzioni su queue

```

void init(queue *q) {
    if (q) {
        q->head = 0; // inizializzo la testa
        q->tail = 0; // inizializzo la coda
        q->num = 0; // inizializzo il numero di elementi
    }
    else {
        printf("Errore: coda non inizializzata\n");
    }
}

int empty(queue *q) {
    return (q->num == 0); // se coda vuota num == 0
}

int full(queue *q) {
    return (q->num == MAX); // se coda piena num == MAX (max=cardinalità)
}

void enqueue(tipo el , queue *q) {
    if(!full(*q)) {
        q->dato[q->tail] = el; // inserisco in coda
        q->tail = successore(q->tail, MAX); // incremento la coda
        q->num++; // incremento il numero di elementi
    }
    else {
        printf("Errore: coda piena\n");
    }
}

int dequeue(queue *q) {
    int res = novall;

```

```

if(!empty(*q)) {
    res = q->dato[q->head]; // restituisco il dato in testa
    q->head = successore(q->head, MAX); // incremento la testa
    q->num--; // decremento il numero di elementi
}
else {
    printf("Errore: coda vuota\n");
}
return res;
}

```

implementazione con le strutture

```

typedef struct {
    lista head; //testa della lista
    lista tail; //coda della lista
    int num; // numero di elementi nella coda
}queue;

```

```

void init(queue *q) {
    if (q) {
        q->head = NULL; // inizializzo la testa
        q->tail = NULL; // inizializzo la coda
        q->num = 0; // inizializzo il numero di elementi
    }
    else {
        printf("Errore: coda non inizializzata\n");
    }
}

```

```

void enqueue(tipo el , queue *q) {
    if(q) {
        if(!full(*q)) {
            if(!empty(*q)) {
                q->tail->next = (lista)malloc(sizeof(nodo)); //
                alloco il nodo

                q->tail = q->tail->next; // incremento la coda
                q->tail->dato = el; // inserisco in coda
                q->tail->next = NULL; // prossimo nodo NULL
            }
            else {
                q->head = (lista)malloc(sizeof(nodo)); // alloco

```

```

il nodo

        q->head->dato = el; // inserisco in testa
        q->head->next = NULL; // prossimo nodo NULL
        q->tail = q->head; // la testa è anche la coda
    }
}
else {
    printf("Errore: coda piena\n");
}
}
else {
    printf("Errore: coda non inizializzata\n");
}
}

```

```

int dequeue(queue *q) {
    int res = novall;
    if(q) {
        if(!empty(*q)) {
            res = q->head->dato; // restituisco il dato in testa
            lista tmp = q->head; // salvo la testa
            q->head = q->head->next; // incremento la testa
            free(tmp); // libero la memoria
            q->num--; // decremento il numero di elementi
        }
        else {
            printf("Errore: coda vuota\n");
        }
    }
    else {
        printf("Errore: coda non inizializzata\n");
    }
    return res;
}

```