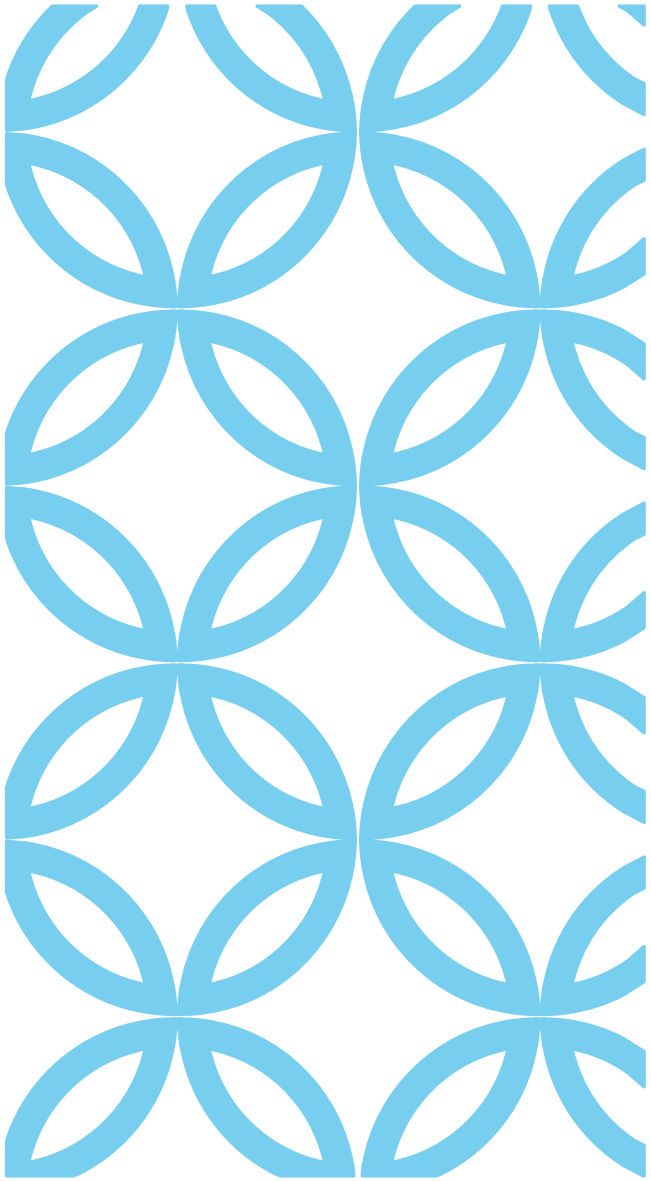
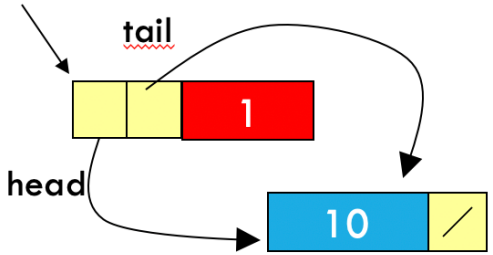


Corso di Programmazione e Strutture Dati
Docente di Laboratorio: Marco Romano
Email: marromano@unisa.it

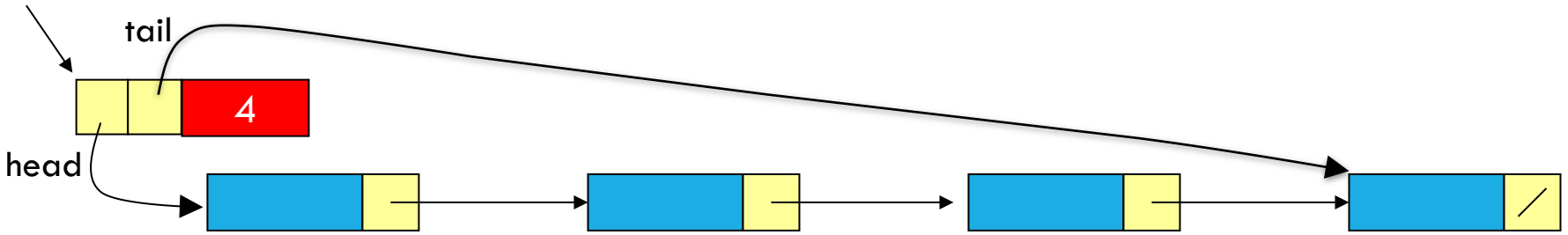
OTTIMIZZAZIONE DELLE LISTE E ADT STACK CON ARRAY DINAMICI

OTTIMIZZAZIONE DELLE LISTE



OTTIMIZZAZIONE DEGLI OPERATORI IN LIST

- È possibile utilizzare gli operatori di:
  - Rimozione dalla testa
  - Aggiunta in coda
- Per motivi di efficienza, conviene avere accesso sia al primo elemento sia all'ultimo. Occorre modificare il tipo lista come un puntatore ad una struct che contiene
  - Un intero numelem che indica il numero di elementi della coda
  - Un puntatore head ad uno struct nodo
  - Un puntatore tail ad uno struct nodo



OTTIMIZZAZIONE DEGLI OPERATORI IN LIST

- Dobbiamo innanzitutto aggiungere il puntatore **tail**
- Poi bisogna modificare gli operatori principali:
  - ***RemoveHead***
    - Deve eventualmente aggiornare entrambi i puntatori head e tail.
  - ***addListTail***, grazie alla presenza del puntatore tail,
    - Non deve più scorrere gli elementi della lista fino all'ultimo
    - Deve eventualmente aggiornare entrambi i puntatori head e tail.

MODIFICA REMOVEHEAD (ADT LIST)

- Bisogna prima salvare il puntatore al nodo da eliminare (quello puntato da head)
- Head dovrà quindi puntare al successivo
- A questo punto si può deallocare la memoria del nodo da rimuovere
- Se la coda aveva un solo elemento, ora è vuota, per cui bisogna porre anche il puntatore tail a NULL

```
 9  struct list {
10      int size;
11      struct node *head;
12      struct node *tail;
13  };
```

MODIFICARE LA STRUCT LIST

```
22  List newList(){
23
24      List list = malloc(sizeof(struct list));
25      list->size = 0;
26      list->head = NULL;
27      list->tail = NULL;
28      return list;
29  }
```

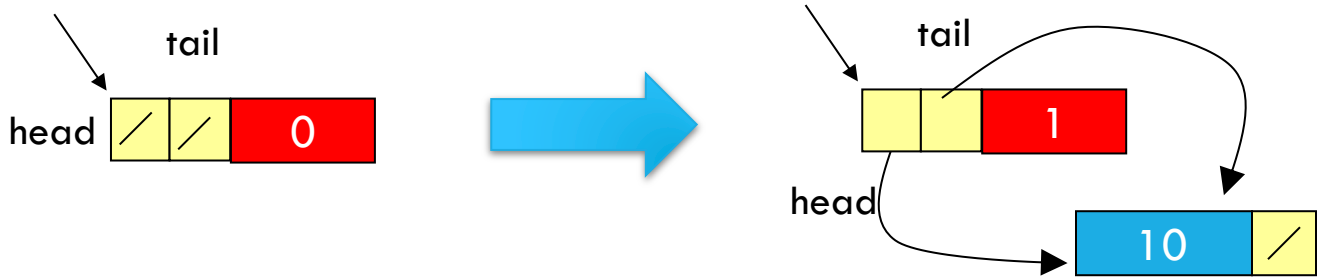
NEWLIST

REMOVEHEAD

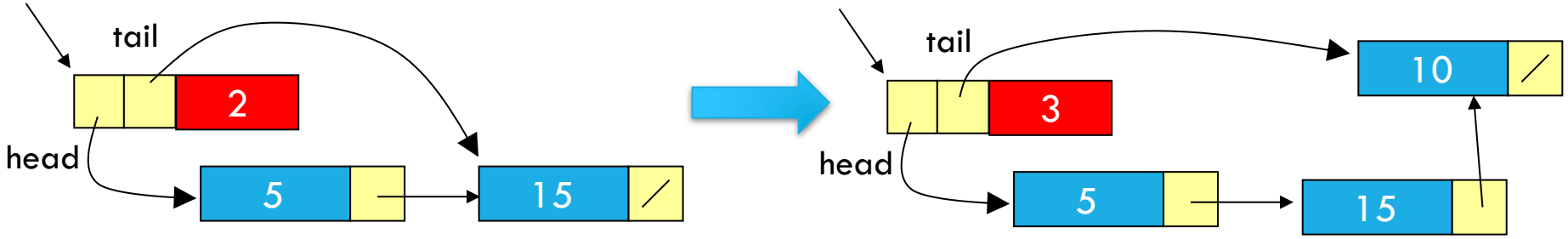
```
35  Item removeHead(List list){
36      Item app;
37      if(isEmpty(list)==1){
38          fprintf(stderr,"Lista vuota");
39          return NULL;
40      }
41      struct node *temp = list->head;
42      list->head = temp->next;
43      app=temp->item;
44      free(temp);
45      list->size--;
46      if(list->size == 0)
47          list->tail = NULL;
48
49      return app;
50  }
```

MODIFICA ADDLISTTAIL (ADT LIST)

- Dobbiamo innanzitutto creare un nuovo nodo a cui dovrà puntare il puntatore tail
- Poi bisogna distinguere il caso in cui la coda di input è vuota e il caso in cui non è vuota
  - Coda vuota: il puntatore head dovrà puntare al nuovo nodo



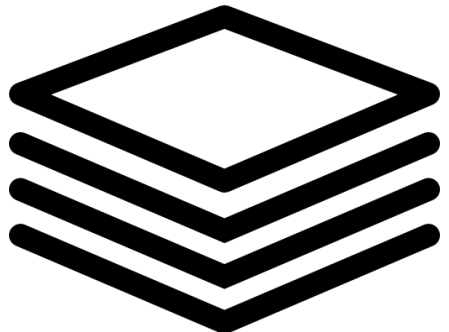
  - Coda non vuota: il puntatore next dell'ultimo nodo dovrà puntare a nuovo



ADDLISTTAIL

```
63  int addListTail(List list, Item item){
64
65      struct node *new = malloc(sizeof(struct node));
66      new -> next = NULL;
67      new -> item = item;
68      if(list->size == 0)
69          list->head = new;
70      else
71          list->tail->next = new;
72      list->tail = new;
73
74      list -> size++;
75      return 1;
76
77  }
```

STACK CON ARRAY DINAMICI



STACK: IMPLEMENTAZIONE CON ARRAY DINAMICI

- Come facciamo ad evitare che lo stack abbia una capienza massima ?
  - Bisogna usare l'allocazione dinamica della memoria e due costanti
  - La prima START_DIM definisce la dimensione iniziale dello stack
  - La seconda ADD_DIM definisce di quanto allargare lo stack nel caso in cui si riempia
  - Questo significa che ci occorre anche una variabile dim che ci dica quanti elementi può contenere lo stack in ogni momento

```
6   #define START_DIM 10
7   #define ADD_DIM 5
8
9   struct stack
10  {
11    Item *elements;
12    int top;
13    int dim;
14  };
```

SCTRUCT STACK

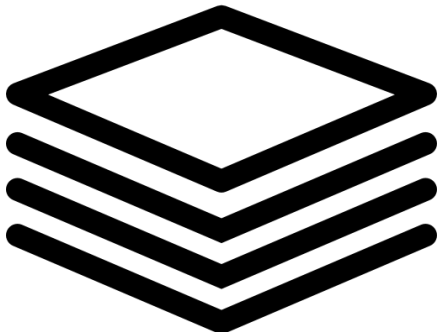
NEWSTACK

```
16  Stack newStack()
17  {
18    Stack s=malloc(sizeof(struct stack));
19
20    if(s==NULL)
21      return NULL;
22    s->top=0;
23
24    s->elements=malloc(sizeof(Item)*START_DIM);
25
26    if(s->elements==NULL)
27      return NULL;
28    s->dim=START_DIM;
29
30    return s;
31  }
```

PUSH

```
41  int push(Stack s, Item i)
42  {
43    Item *temp;
44    if(s->top==s->dim){
45      temp=realloc(s->elements, sizeof(Item)*(s->dim+ADD_DIM));
46      if(temp==NULL)
47        return 0;
48      else{
49        s->elements=temp;
50        s->dim+=ADD_DIM;
51        printf("Spazio esteso a %d\n", s->dim);
52      }
53    }
54    s->elements[s->top]=i;
55    s->top++;
56    return 1;
57
58  }
```

STACK : ESERCITAZIONE



ESERCIZIO SULL'USO DI STACK
ESPRESSIONI CON PARENTESI BILANCIATE

- **Verificare se una data espressione aritmetica è ben bilanciata rispetto a tre tipi di parentesi: ( ) , [ ] , { }**

(4 + a) * {[1 – (2/x)] * (8 – a)} è ben bilanciata

[x – (4y + 3] * (1 – x)) non è ben bilanciata

N.B.: per semplicità supponiamo che non esista un ordine di priorità fra i tre tipi di parentesi

(a + {b -1}) / [b + 2] è ammessa come valida

PARENTESI BILANCIATE
ANALISI DEL PROBLEMA (1 DI 2)

- **Vogliamo solo verificare se una data espressione aritmetica è ben bilanciata rispetto alle parentesi**
  - **non ci interessa sapere se gli operatori in essa contenuti sono corretti e se hanno il giusto numero di operandi**
- **Possiamo estrarre dall'espressione solo le parentesi, cancellando tutto il resto**
  - **se l'espressione (4 + a) * {[1 – (2/x)] * (8 – a)} è ben bilanciata, lo valutiamo dalla sua versione semplificata:**

( ) { [( )] ( ) }

PARENTESI BILANCIATE
ANALISI DEL PROBLEMA (2 DI 2)

- Dati di ingresso: **Una stringa exp**
  - Precondizione: **True**
- Dati di uscita: **un valore booleano ris**
  - Postcondizione:
    - se la stringa exp è vuota, allora ris = true
    - se la stringa exp non è vuota, allora ris=true se le parentesi in essa presenti sono bilanciate, ris=false se non lo sono

Dizionario dei dati

Identificatore	Tipo	Descrizione
exp	stringa	espressione in input
ris	booleano	risultato della valutazione in output

PARENTESI BILANCIATE PROGETTAZIONE

Step 1: prendere in input una stringa **exp**

Step 2: se **exp** è vuota, dare in output true,

Step 3: sia **S** uno stack di caratteri inizialmente vuoto

Step 4: per ogni carattere **car** della stringa

- se **car** == '(' or car=='[' or **car**=='{'

  inserisci **car** in **S**

  se **car** == ')' or **car**==']' or **car**=='}' allora…

  1) se **S** è vuoto dare in output **false**

  2) altrimenti estrarre il top da **S** e metterlo in una variabile **t**

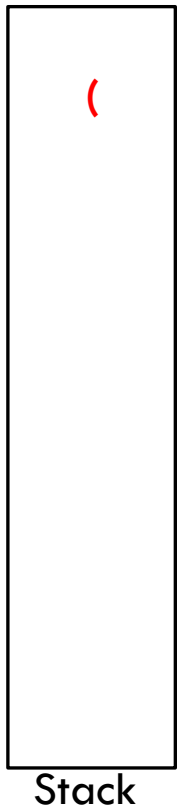
  -) se **car** non corrisponde a **t** dare in output **false**

Step 5: se **S** è vuoto dare in output **true**

altrimenti dare in output **false**

( ) { [( )] ( ) }

Solo le parentesi che aprono



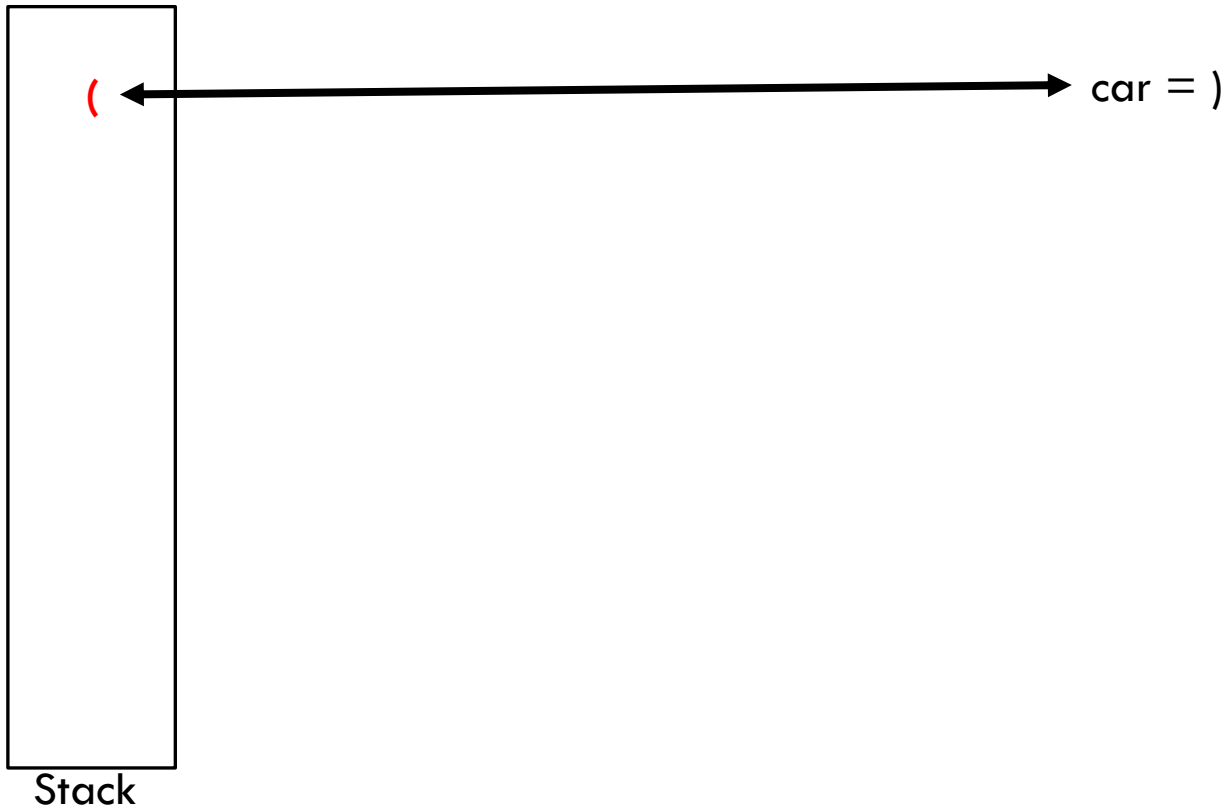
Solo le parentesi che chiudono

car =

( ) { [( )] ( ) }

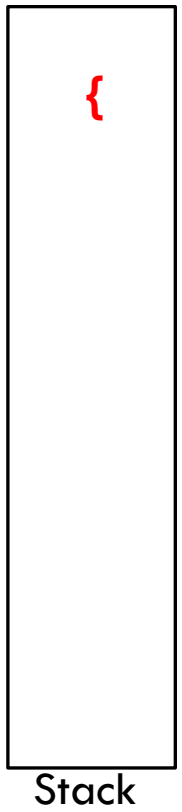
Solo le parentesi che aprono

Solo le parentesi che chiudono



( ) { [( )] ( ) }

Solo le parentesi che aprono



Solo le parentesi che chiudono

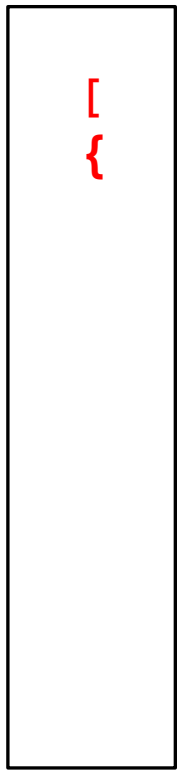
car =

( ) { [( )] ( ) }

Solo le parentesi che aprono

Solo le parentesi che chiudono

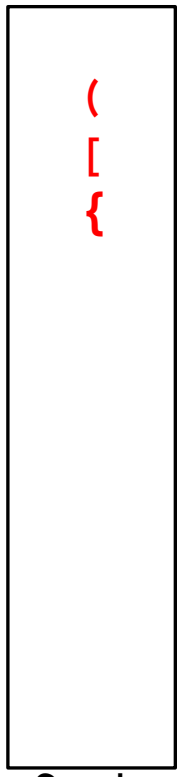
car =



Stack

( ) { [( )] ( ) }

Solo le parentesi che aprono



Stack

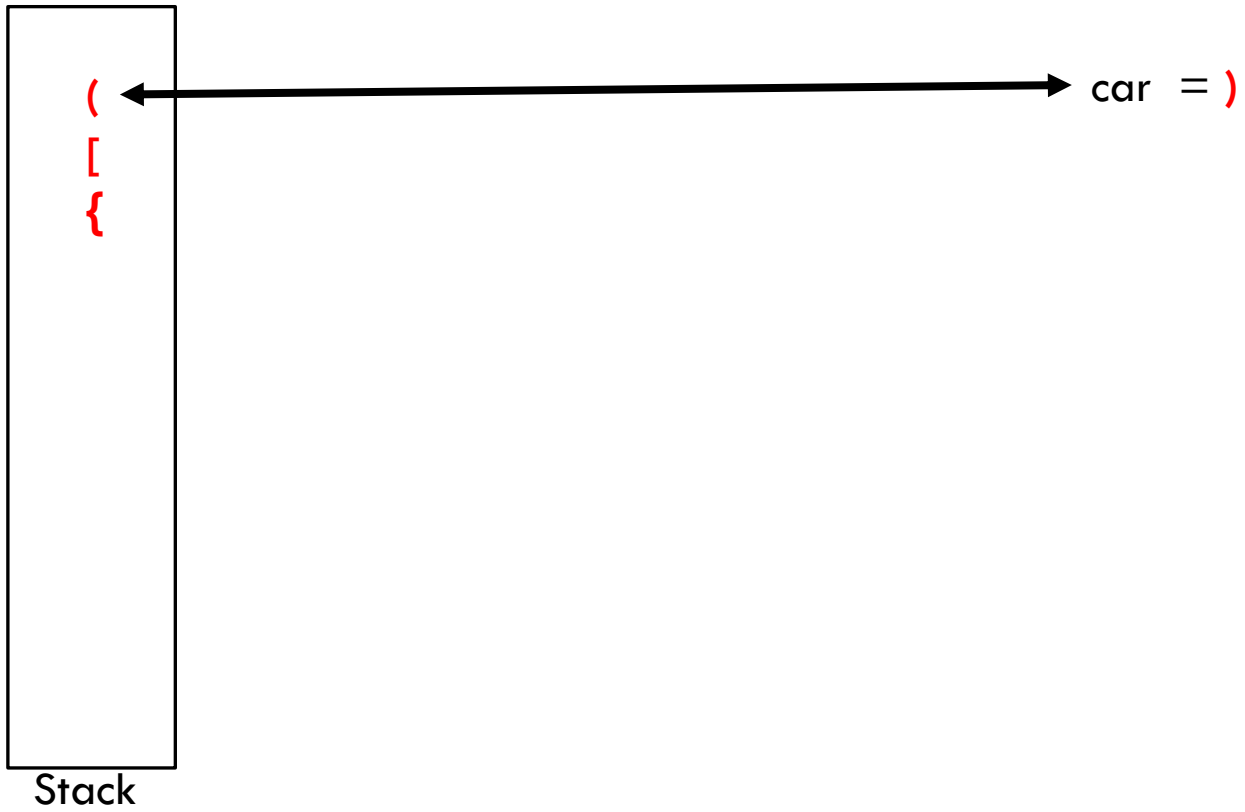
Solo le parentesi che chiudono

car =

( ) { [( )] ( ) }

Solo le parentesi che aprono

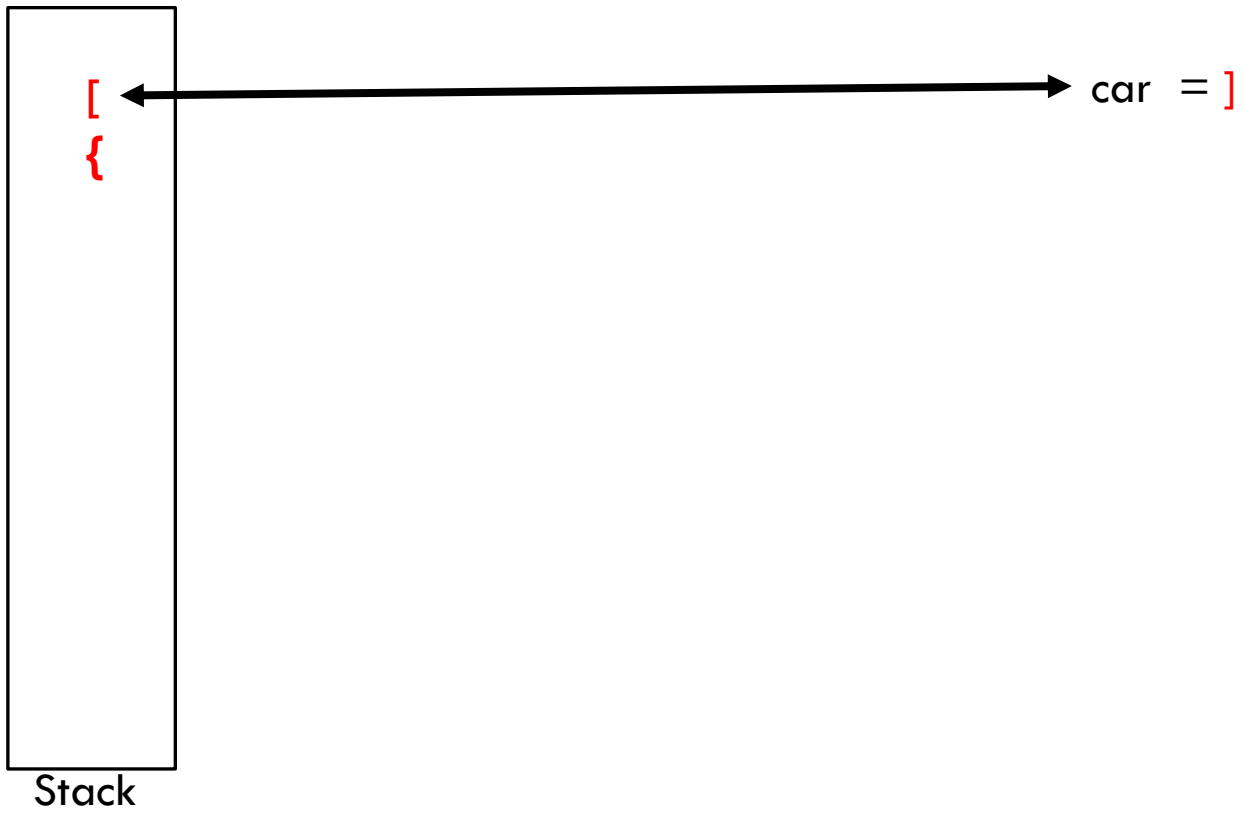
Solo le parentesi che chiudono



( ) { [( )] ( ) }

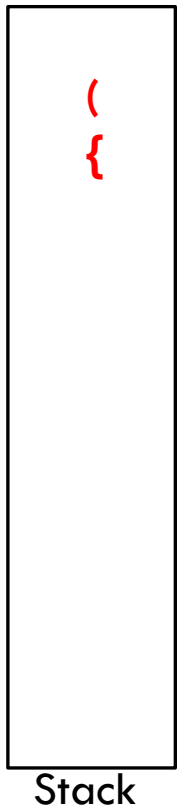
Solo le parentesi che aprono

Solo le parentesi che chiudono



( ) { [( )] ( ) }

Solo le parentesi che aprono



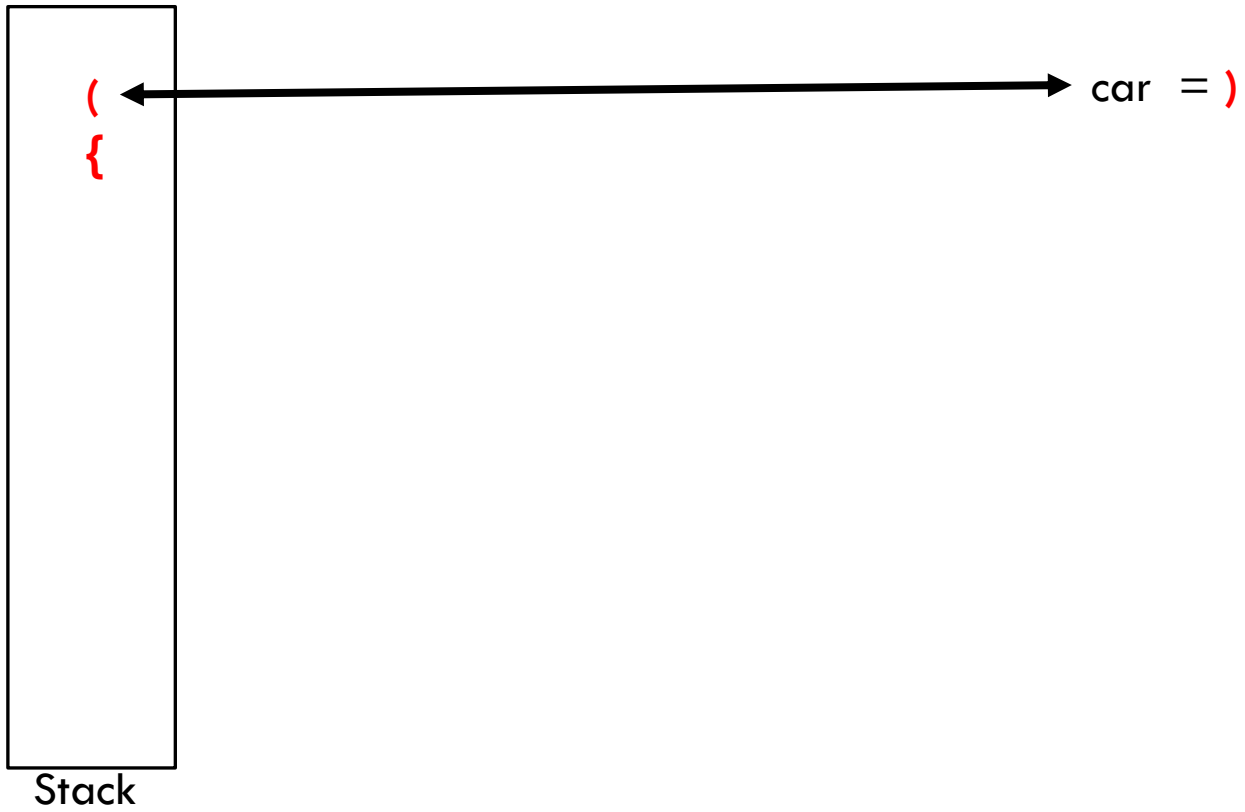
Solo le parentesi che chiudono

car =

( ) { [( )] ( ) }

Solo le parentesi che aprono

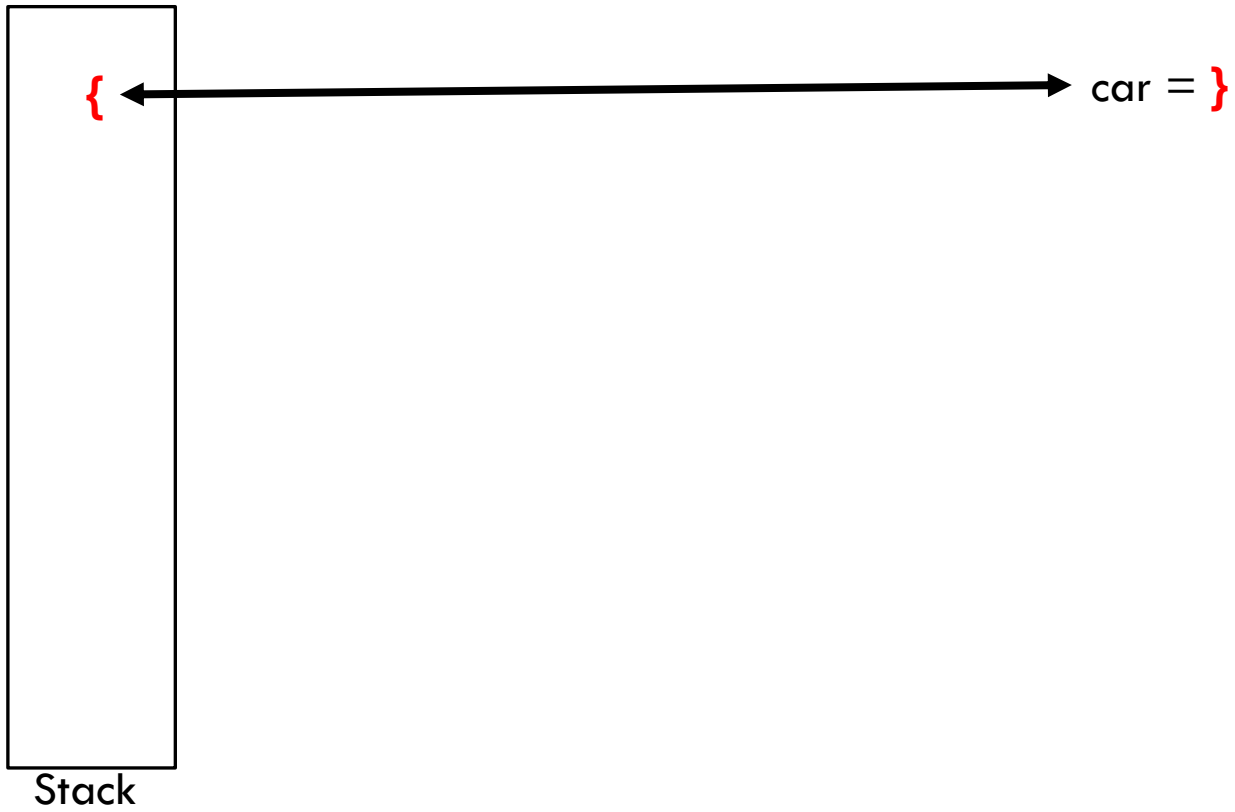
Solo le parentesi che chiudono



( ) { [( )] ( ) }

Solo le parentesi che aprono

Solo le parentesi che chiudono



SUGGERIMENTI

Implementare le funzioni di bilanciamento nel file main o in una libreria apposita.
Non occorre modificare o estendere un precedente ADT

```c
#define N 30
int isOpen(char ch){

    //valuta se il carattere è una parentesi aperta
}

int isClosed(char ch){

    //valuta se il carattere è una parentesi chiusa
}

int isCorresponding(char ch1, char ch2){

    //valuta se ch2 è la parentesi chiusa per ch1
    //  una potenziale soluzione è usare l'aritmetica dei caratteri basata sul codice ASCII
    //  ***TABELLA DEI CARATTERI ASCII***
    //  le parentesi graffe (123,125)
    //  le parentesi quadre (91, 93)
    //  le parentesi tonde  (40, 41)
}

int isBalanced(char *exp){

    //algoritmo che, utilizzando le funzioni precedenti, verifica se l'espressione è bilanciata

}

int main() {
    char exp[N];
    printf("Inserisci l'espressione: ");
    scanf("%[^\n]", exp);
    if (isBalanced(exp))
        printf("L' espressione e' bilanciata");
    else
        printf("L'espressione non e' bilanciata\n");
}
```