1) Implementa una funzione:

```
BST* crea_bst_da_array(int* arr, int size);
```

che, dato un array ordinato arr di dimensione size, crei un albero binario di ricerca bilanciato contenente tutti gli elementi dell'array. La funzione deve restituire il puntatore alla radice del nuovo albero.

2) Data la seguente struttura ricorsiva:

```
struct BST {
    int value;
    struct BST* left;
    struct BST* right;
    int height;
};
```

E la seguente funzione:

```
int updateHeight(struct BST* List) {
    if (List == NULL) return -1;
    int leftHeight = updateHeight(List->left);
    int rightHeight = updateHeight(List->right);
    List->height = 1 + (leftHeight > rightHeight ? leftHeight :
rightHeight);
    return List->height;
}
```

3) Consideriamo la seguente struttura e funzione:

```
typedef struct List {
    int data;
    struct List* next;
} List;


void f(List** head) {
    List *slow = *head, *fast = *head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
```

```
    }
    // Operazione con slow
}
```

a) Cosa fa questa funzione? b) In quale scenario questa funzione potrebbe essere utile? c) Qual è la complessità temporale di questa funzione?

4) Considera il seguente frammento di codice:

```
int x = 5;
int *p = &x;
int **q = &p;
(*q)++;
printf("%d", x);
```

a) Cosa viene stampato e perché? b) Disegna un diagramma che mostri la relazione tra x, p e q in memoria. c) Come cambierebbe il risultato se sostituissimo `(*q)++;` con `(**q)++;`?

5) Data la seguente struttura e dichiarazione:

```
struct S {
    char a;
    int b;
    char c;
};

struct S arr[10];
```

a) Quanto spazio occupa in memoria `arr`? Spiega il tuo ragionamento. b) Come cambierebbe l'occupazione di memoria se riordinassimo i campi della struttura? Perché? c) Scrivi una versione della struttura che minimizzi l'occupazione di memoria.

6) Implementa una funzione che unisca due liste ordinate in una nuova lista ordinata:

```
List* merge_sorted_lists(List* l1, List* l2);
```

7) Considera il seguente codice:

```
void func(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

int main() {
    int a[2] = {1, 2};
    func(a, a+1);
    printf("%d %d", a[0], a[1]);
    return 0;
}
```

a) Cosa viene stampato e perché? b) Cosa succede se chiamiamo `func(&a[0], &a[0])`? c) Come modificheresti `func` per evitare problemi nel caso b?

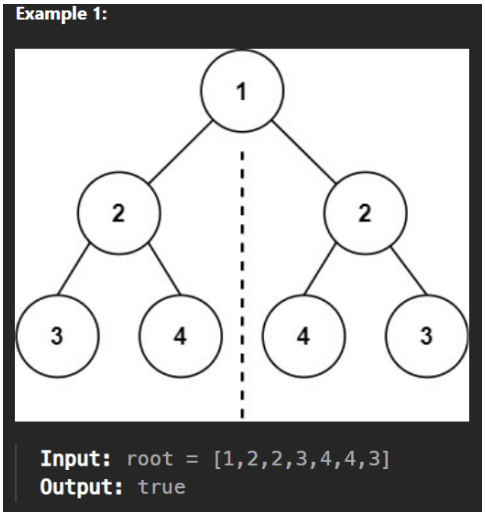
8) Implementa una funzione che trovi tutti i cammini da radice a foglia in un albero binario che sommano a un valore target:

```
void pathSum(BST* root, int targetSum, int* currentPath, int pathSize,
int** result, int* resultSize);
```

9) Implementa una funzione che verifichi se un albero binario è simmetrico:

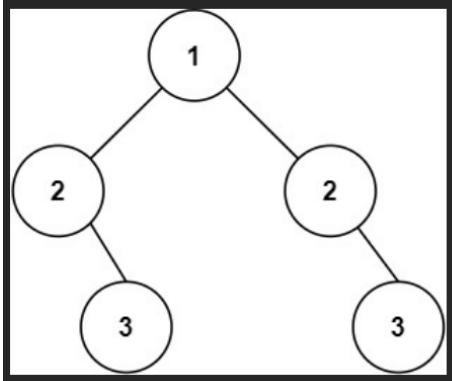
```
bool isSymmetric(BST* root);
```

Example 1:



Input: root = [1,2,2,3,4,4,3]
Output: true

Example 2:

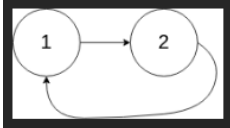


Input: root = [1,2,2,null,3,null,3]
Output: false

10) Scrivi una funzione che rilevi se c'è un ciclo in una lista concatenata:

```
bool hasCycle(List* head);
```

Example 2:



Input: head = [1,2], pos = 0
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.

Example 3:

Input: head = [1], pos = -1
Output: false
Explanation: There is no cycle in the linked list.

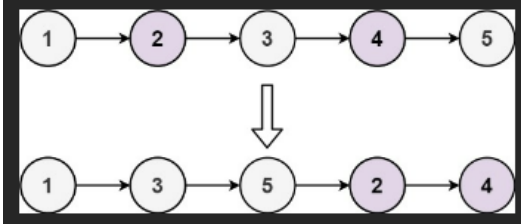
11) Data la testa di una linked list, raggruppa tutti i nodi con indici dispari seguiti dai nodi con indici pari e restituisci la lista riordinato.

Il primo nodo è considerato dispari, il secondo è pari e così via.

Si noti che l'ordine relativo all'interno dei gruppi pari e dispari deve rimanere quello dell'input.

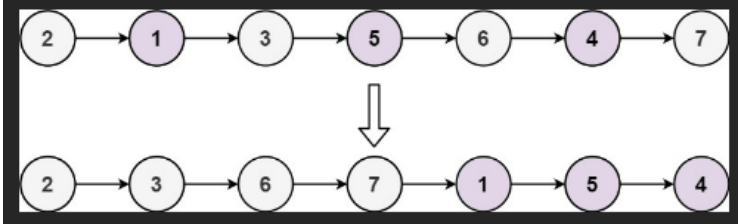
```
List* oddEvenList(List* head)
```

Example 1:



Input: head = [1,2,3,4,5]
Output: [1,3,5,2,4]

Example 2:



Input: head = [2,1,3,5,6,4,7]
Output: [2,3,6,7,1,5,4]