Laboratorio I
a.a. 2025/2026

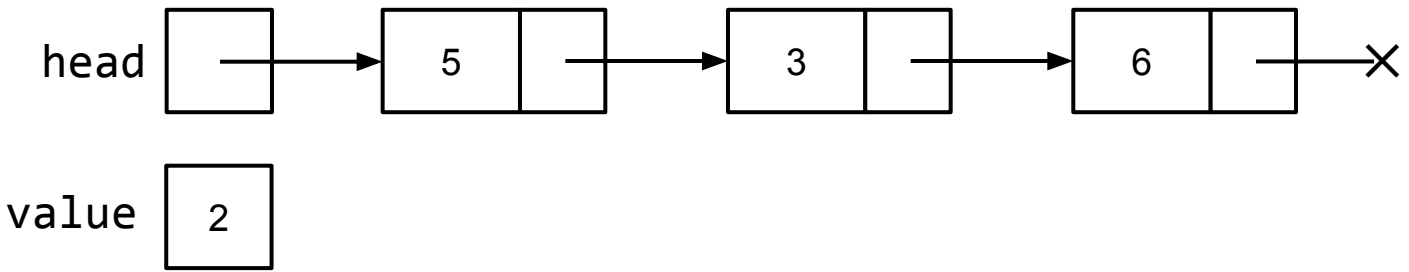
Ripasso ed esercizi su liste concatenate

Contenuti

- Ripasso operazioni di base su liste concatenate
- Esercizi su liste concatenate e ricorsione

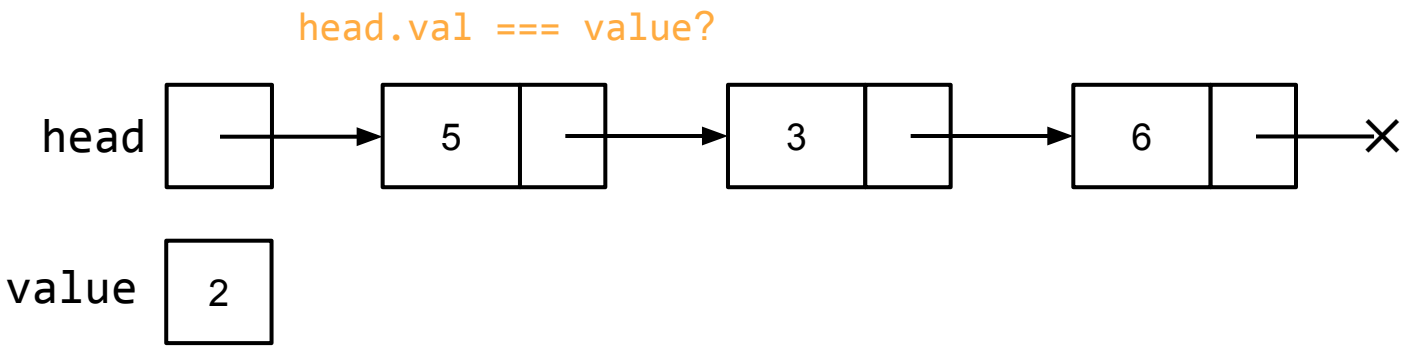
listFind(head, value): Restituisce il primo nodo che contiene value, oppure null se non trovato

```
1.  function listFind(head, value) {
2.      if (!head)
3.          return null
4.      if (head.val === value)
5.          return head
6.      return listFind(head.next, value)
7.  }
```



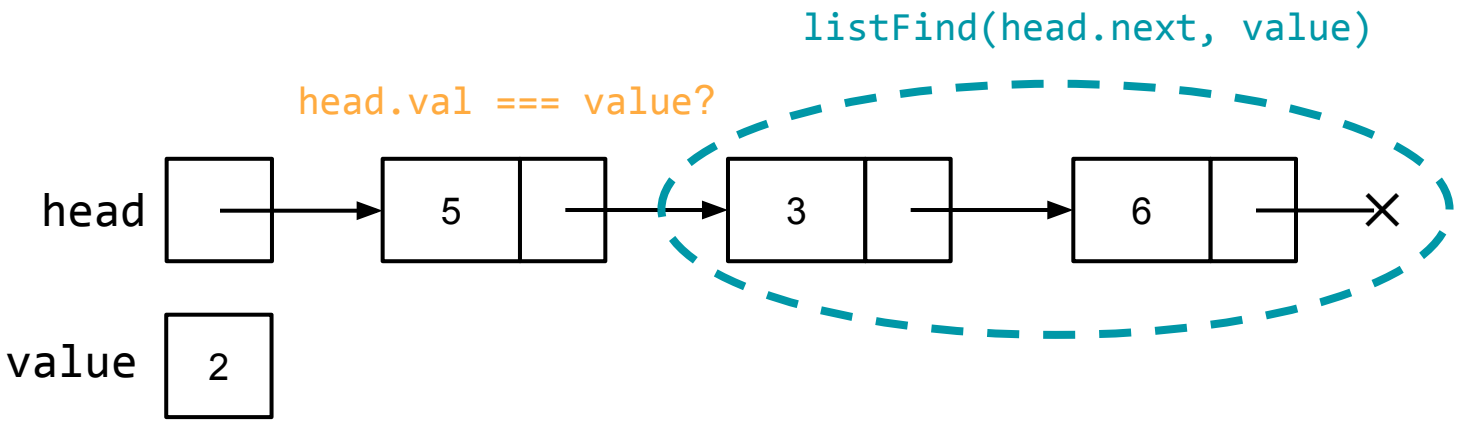
listFind(head, value): Restituisce il primo nodo che contiene value, oppure null se non trovato

```
1.  function listFind(head, value) {
2.      if (!head)
3.          return null
4.      if (head.val === value)
5.          return head
6.      return listFind(head.next, value)
7.  }
```



listFind(head, value): Restituisce il primo nodo che contiene value, oppure null se non trovato

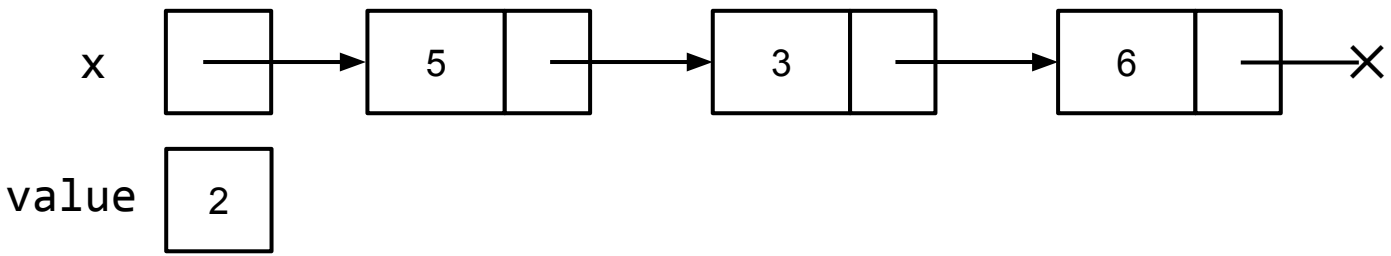
```
1.  function listFind(head, value) {
2.      if (!head)
3.          return null
4.      if (head.val === value)
5.          return head
6.      return listFind(head.next, value)
7.  }
```



listInsert

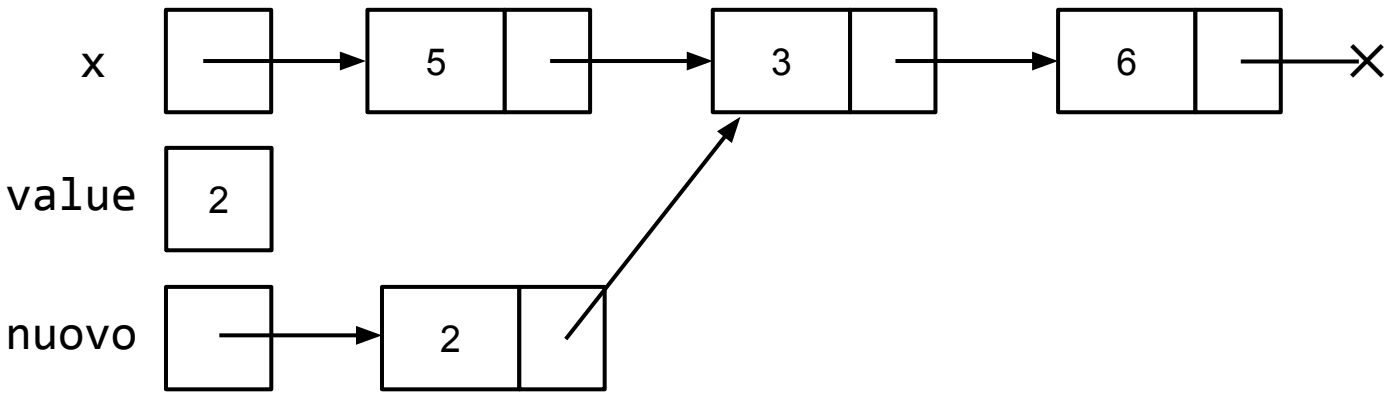
listInsert(x, value): Inserisce un nuovo nodo con valore value dopo il nodo x

```
1.  function listInsert(x, value) {
2.     if (!x) return
3.     let nuovo = { val: value, next: x.next }
4.     x.next = nuovo
5.  }
```



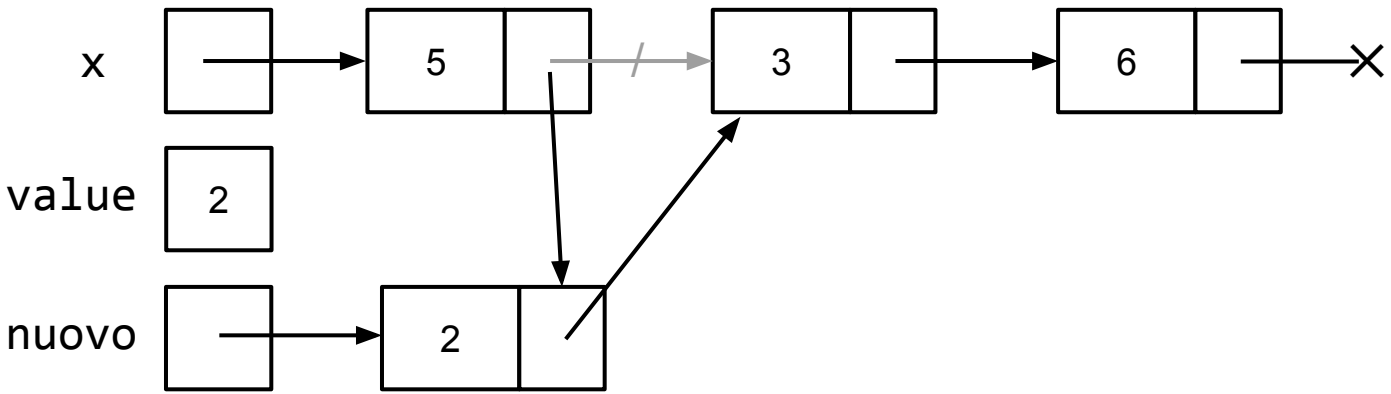
listInsert(x, value): Inserisce un nuovo nodo con valore value dopo il nodo x

```
1.  function listInsert(x, value) {
2.      if (!x) return
3.      let nuovo = { val: value, next: x.next }
4.      x.next = nuovo
5.  }
```



listInsert(x, value): Inserisce un nuovo nodo con valore value dopo il nodo x

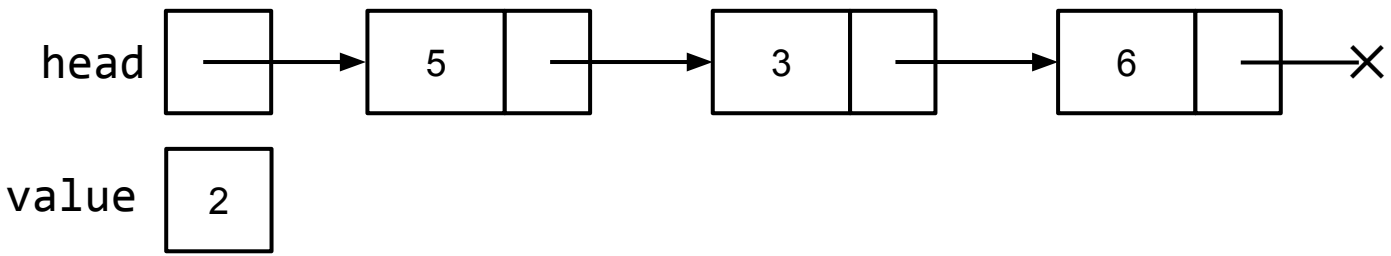
```
1.  function listInsert(x, value) {
2.      if (!x) return
3.      let nuovo = { val: value, next: x.next }
4.      x.next = nuovo
5.  }
```



listPush

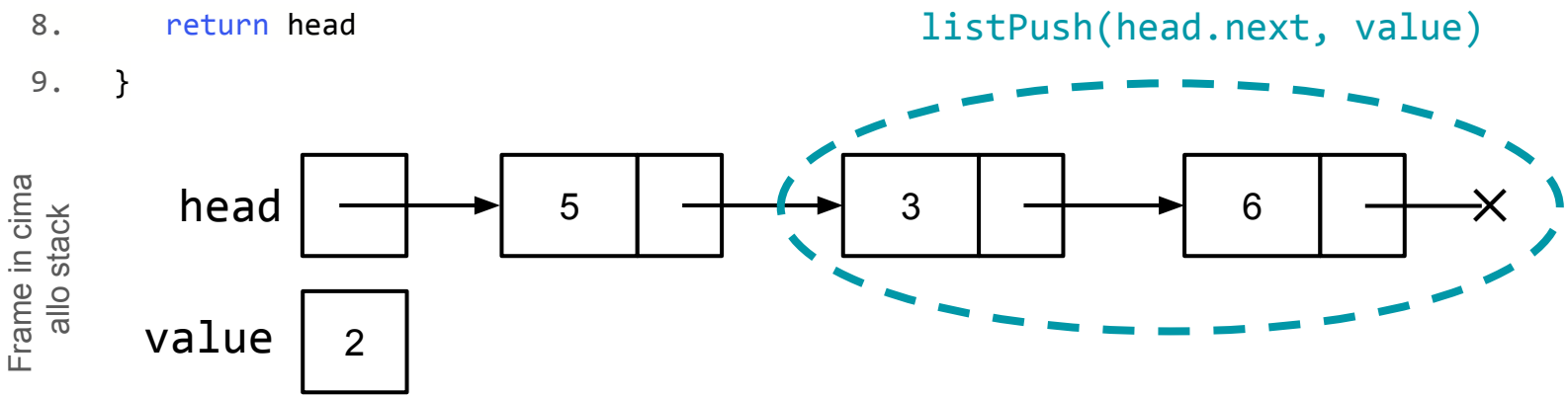
listPush(head, value): Aggiunge un nodo in coda e restituisce la testa aggiornata

```
1.  function listPush(head, value) {
2.      if (!head)
3.          return { val: value, next: null }
4.      if (head.next)
5.          listPush(head.next, value)
6.      else
7.          listInsert(head, value)
8.      return head
9.  }
```



listPush(head, value): Aggiunge un nodo in coda e restituisce la testa aggiornata

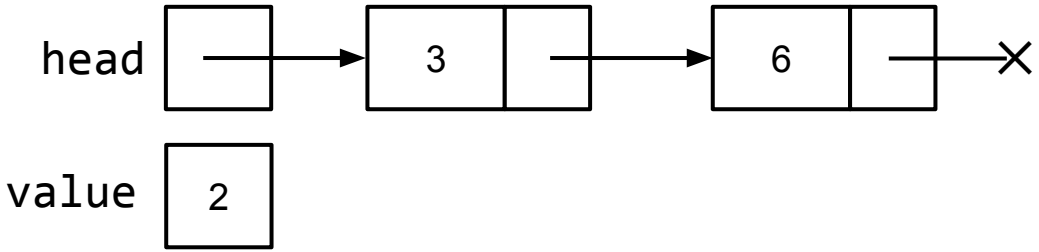
```
1.  function listPush(head, value) {
2.      if (!head)
3.          return { val: value, next: null }
4.      if (head.next)
5.          listPush(head.next, value)
6.      else
7.          listInsert(head, value)
8.      return head
9.  }
```



listPush(head, value): Aggiunge un nodo in coda e restituisce la testa aggiornata

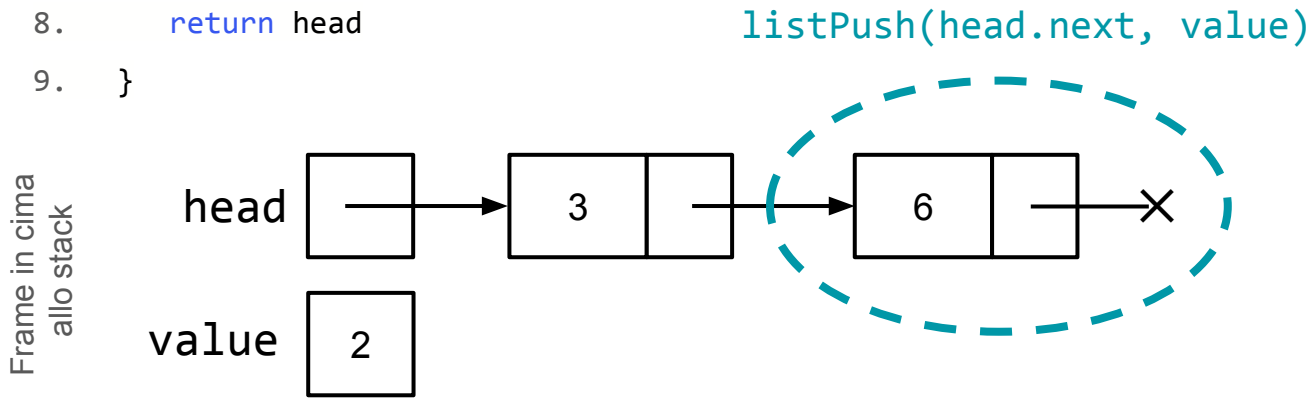
```
1.  function listPush(head, value) {
2.      if (!head)
3.          return { val: value, next: null }
4.      if (head.next)
5.          listPush(head.next, value)
6.      else
7.          listInsert(head, value)
8.      return head
9.  }
```

Frame in cima allo stack



listPush(head, value): Aggiunge un nodo in coda e restituisce la testa aggiornata

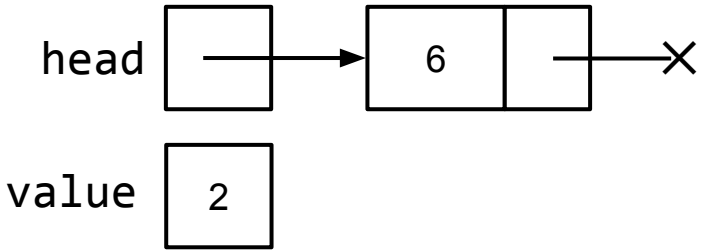
```
1.  function listPush(head, value) {
2.      if (!head)
3.          return { val: value, next: null }
4.      if (head.next)
5.          listPush(head.next, value)
6.      else
7.          listInsert(head, value)
8.      return head
9.  }
```



listPush(head, value): Aggiunge un nodo in coda e restituisce la testa aggiornata

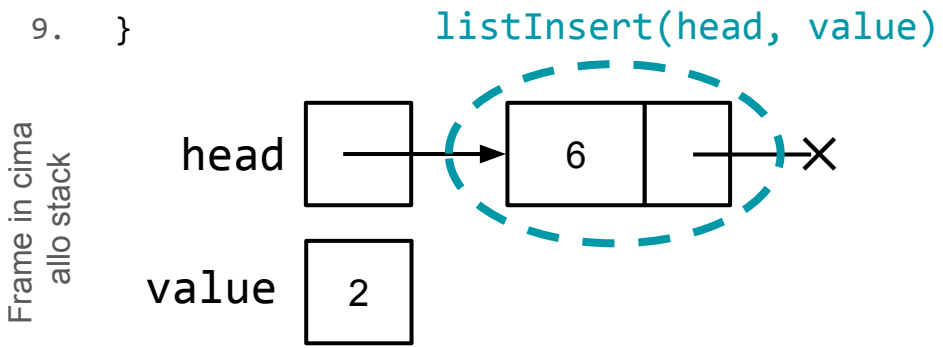
```
1.  function listPush(head, value) {
2.      if (!head)
3.          return { val: value, next: null }
4.      if (head.next)
5.          listPush(head.next, value)
6.      else
7.          listInsert(head, value)
8.      return head
9.  }
```

Frame in cima allo stack



listPush(head, value): Aggiunge un nodo in coda e restituisce la testa aggiornata

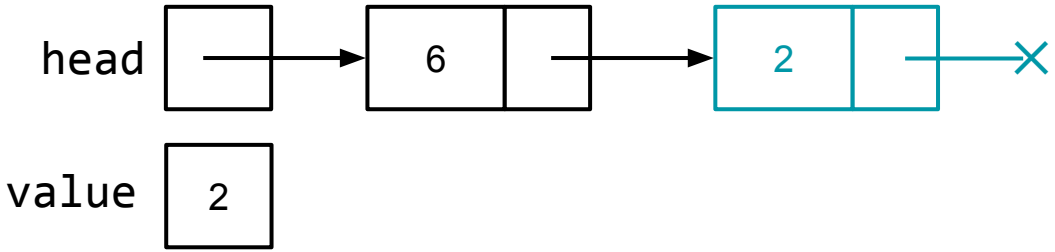
```
1.  function listPush(head, value) {
2.      if (!head)
3.          return { val: value, next: null }
4.      if (head.next)
5.          listPush(head.next, value)
6.      else
7.          listInsert(head, value)
8.      return head
9.  }
```



listPush(head, value): Aggiunge un nodo in coda e restituisce la testa aggiornata

```
1.  function listPush(head, value) {
2.      if (!head)
3.          return { val: value, next: null }
4.      if (head.next)
5.          listPush(head.next, value)
6.      else
7.          listInsert(head, value)
8.      return head
9.  }
```

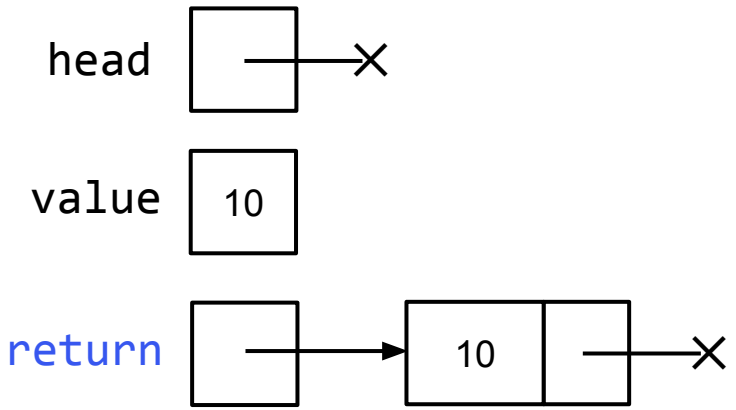
Frame in cima allo stack



listPush(head, value): Aggiunge un nodo in coda e restituisce la testa aggiornata

```
1.  function listPush(head, value) {
2.      if (!head)
3.          return { val: value, next: null }
4.      if (head.next)
5.          listPush(head.next, value)
6.      else
7.          listInsert(head, value)
8.      return head
9.  }
```

Ultimo caso: lista vuota



listPush(head, value): Aggiunge un nodo in coda e restituisce la testa aggiornata

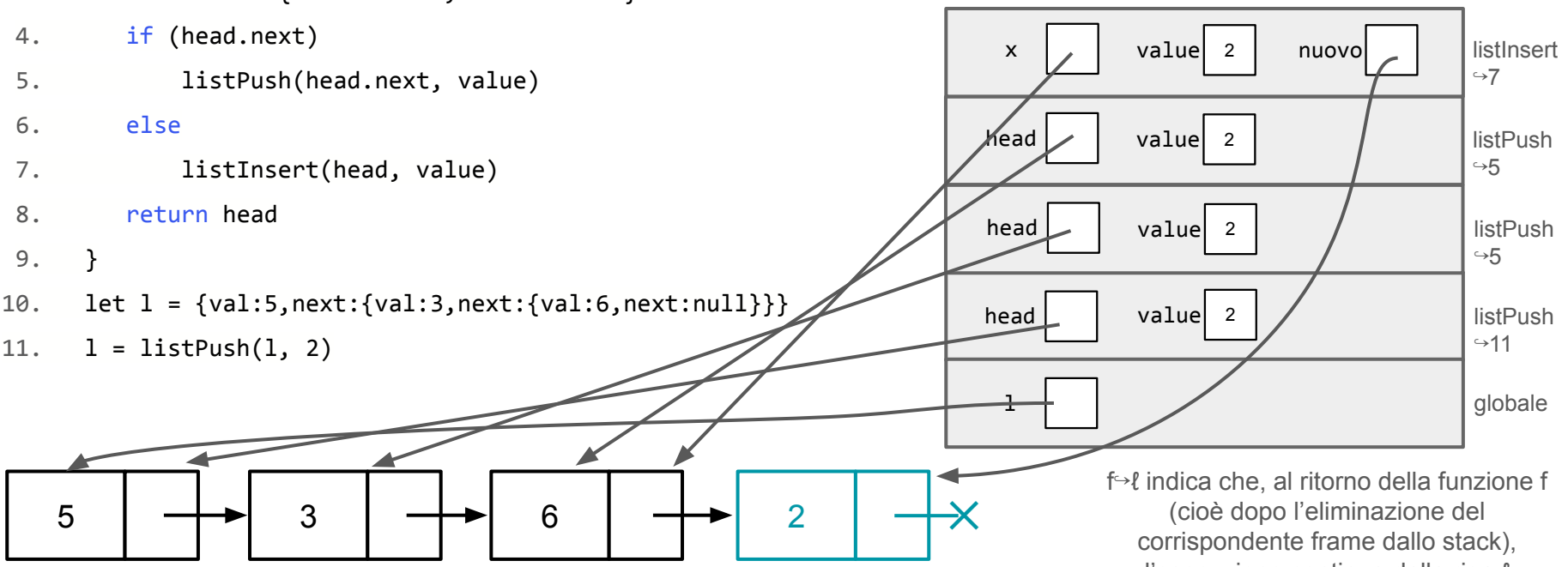
```
1.  function listPush(head, value) {
2.      if (!head)
3.          return { val: value, next: null }
4.      if (head.next)
5.          listPush(head.next, value)
6.      else
7.          listInsert(head, value)
8.      return head
9.  }
10. let l = {val:5,next:{val:3,next:{val:6,next:null}}}
11. l = listPush(l, 2)
```

Qual è il contenuto dell'intero stack quando la ricorsione tocca il fondo?

listPush(head, value): Aggiunge un nodo in coda e restituisce la testa aggiornata

```
1.  function listPush(head, value) {
2.      if (!head)
3.          return { val: value, next: null }
4.      if (head.next)
5.          listPush(head.next, value)
6.      else
7.          listInsert(head, value)
8.      return head
9.  }
10. let l = {val:5,next:{val:3,next:{val:6,next:null}}}
11. l = listPush(l, 2)
```

Qual è il contenuto dell'intero stack quando la ricorsione tocca il fondo?

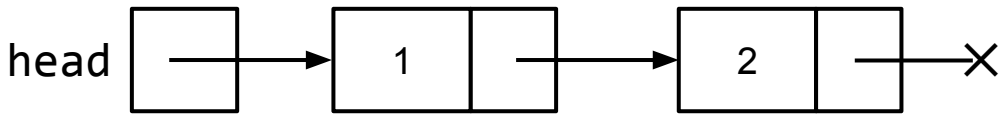


f↪ℓ indica che, al ritorno della funzione f (cioè dopo l'eliminazione del corrispondente frame dallo stack), l'esecuzione continua dalla riga ℓ

listPop

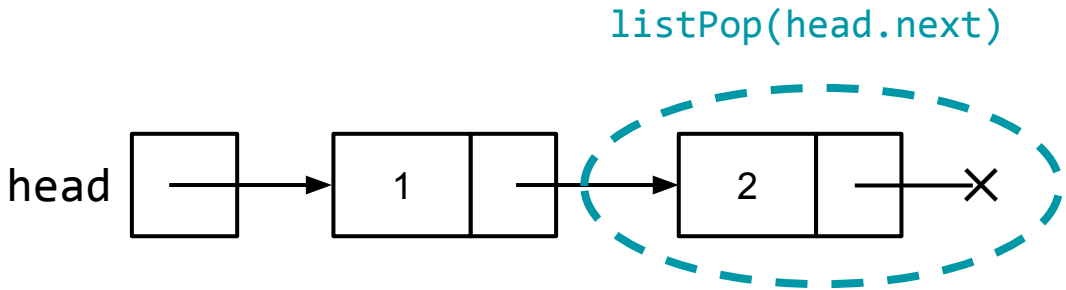
listPop(head): Rimuove il nodo in coda e restituisce una coppia [testa aggiornata, valore rimosso]

```
1.  function listPop(head) {
2.      if (!head)
3.          return [null, undefined]
4.      if (!head.next)
5.          return [null, head.val]
6.      let [newNext, removedVal] = listPop(head.next)
7.      head.next = newNext
8.      return [head, removedVal]
9.  }
```



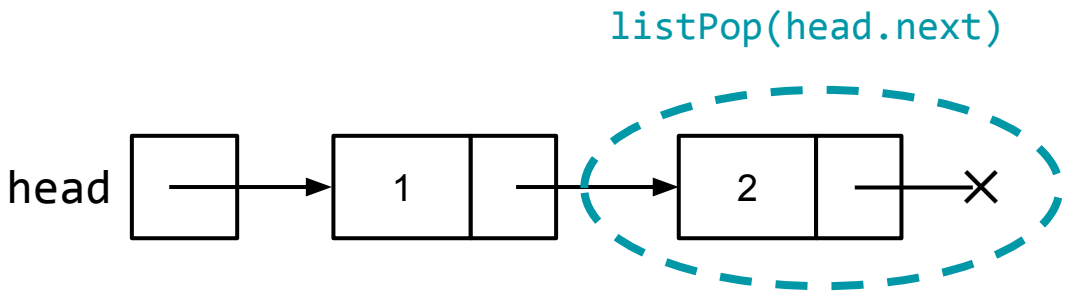
listPop(head): Rimuove il nodo in coda e restituisce una coppia [testa aggiornata, valore rimosso]

```
1.  function listPop(head) {
2.      if (!head)
3.          return [null, undefined]
4.      if (!head.next)
5.          return [null, head.val]
6.      let [newNext, removedVal] = listPop(head.next)
7.      head.next = newNext
8.      return [head, removedVal]
9.  }
```



listPop(head): Rimuove il nodo in coda e restituisce una coppia [testa aggiornata, valore rimosso]

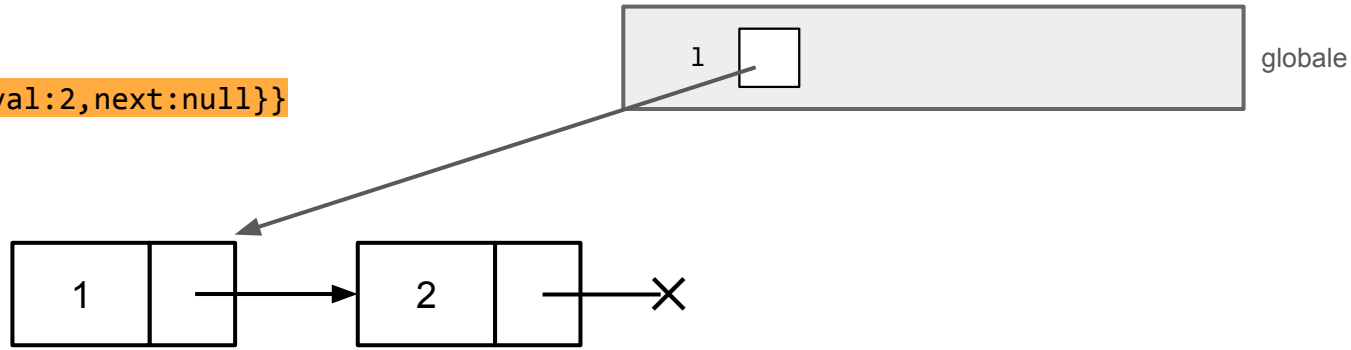
```
1.  function listPop(head) {
2.      if (!head)
3.          return [null, undefined]
4.      if (!head.next)
5.          return [null, head.val]
6.      let [newNext, removedVal] = listPop(head.next)
7.      head.next = newNext
8.      return [head, removedVal]
9.  }
```



Vediamo cosa succede nello stack passo passo…

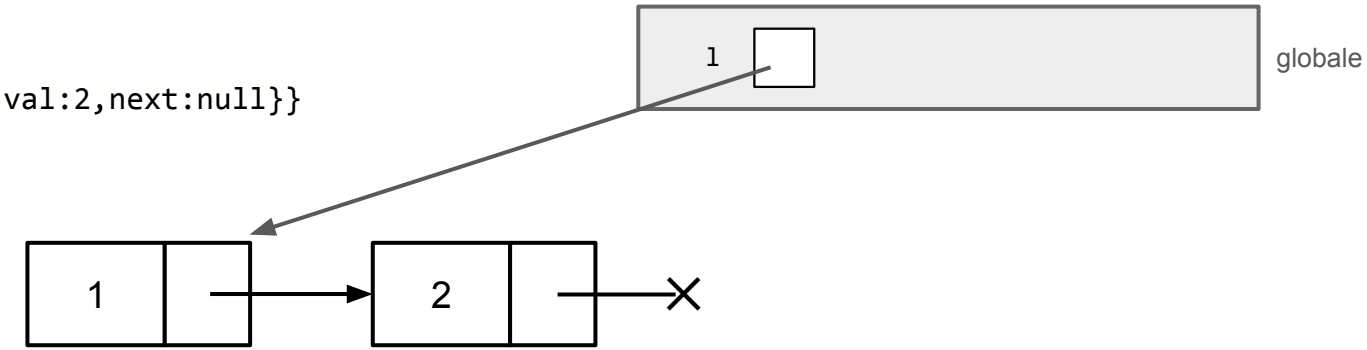
listPop(head): Rimuove il nodo in coda e restituisce una coppia [testa aggiornata, valore rimosso]

```
1.  function listPop(head) {
2.      if (!head)
3.          return [null, undefined]
4.      if (!head.next)
5.          return [null, head.val]
6.      let [newNext, removedVal] = listPop(head.next)
7.      head.next = newNext
8.      return [head, removedVal]
9.  }
10. let l = {val:1,next:{val:2,next:null}}
11. l = listPop(l)[0]
```



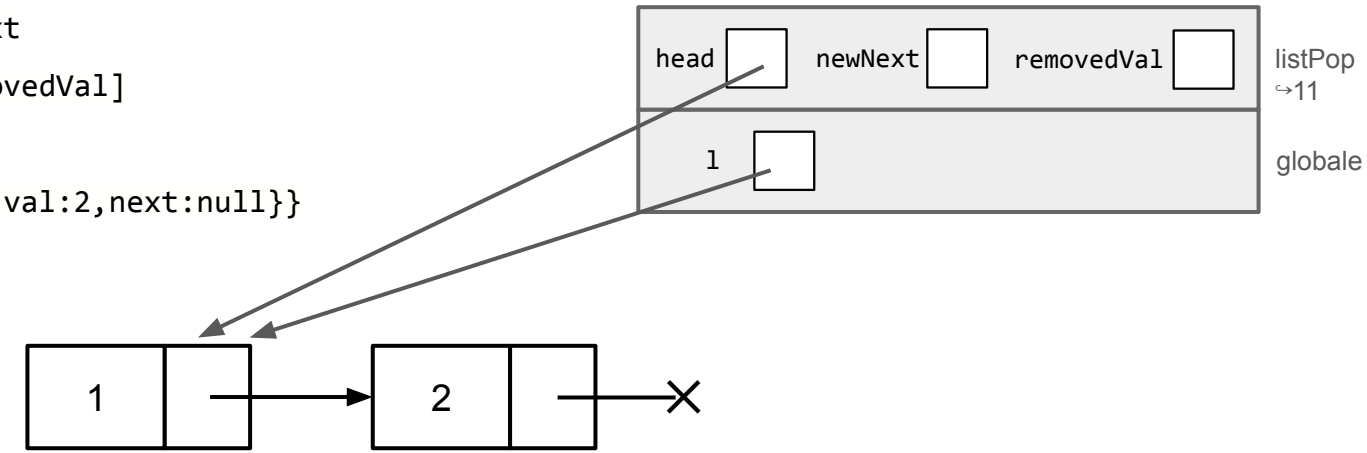
listPop(head): Rimuove il nodo in coda e restituisce una coppia [testa aggiornata, valore rimosso]

```
1.  function listPop(head) {
2.      if (!head)
3.          return [null, undefined]
4.      if (!head.next)
5.          return [null, head.val]
6.      let [newNext, removedVal] = listPop(head.next)
7.      head.next = newNext
8.      return [head, removedVal]
9.  }
10. let l = {val:1,next:{val:2,next:null}}
11. l = listPop(l)[0]
```



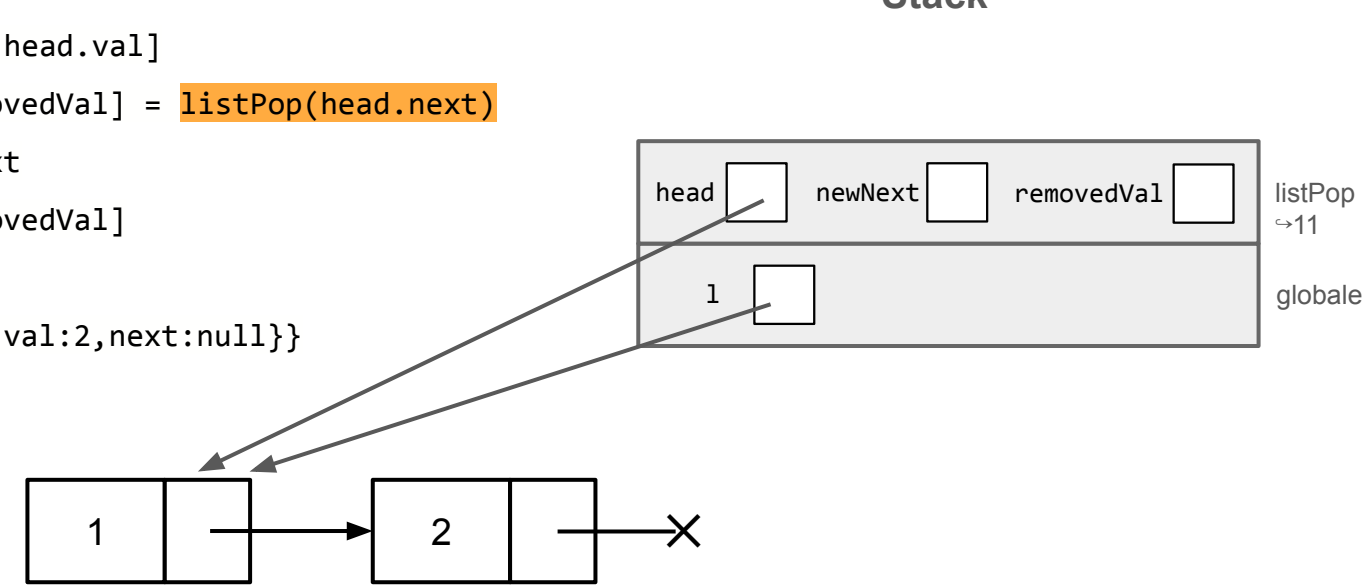
listPop(head): Rimuove il nodo in coda e restituisce una coppia [testa aggiornata, valore rimosso]

```
1.  function listPop(head) {
2.      if (!head)
3.          return [null, undefined]
4.      if (!head.next)
5.          return [null, head.val]
6.      let [newNext, removedVal] = listPop(head.next)
7.      head.next = newNext
8.      return [head, removedVal]
9.  }
10. let l = {val:1,next:{val:2,next:null}}
11. l = listPop(l)[0]
```



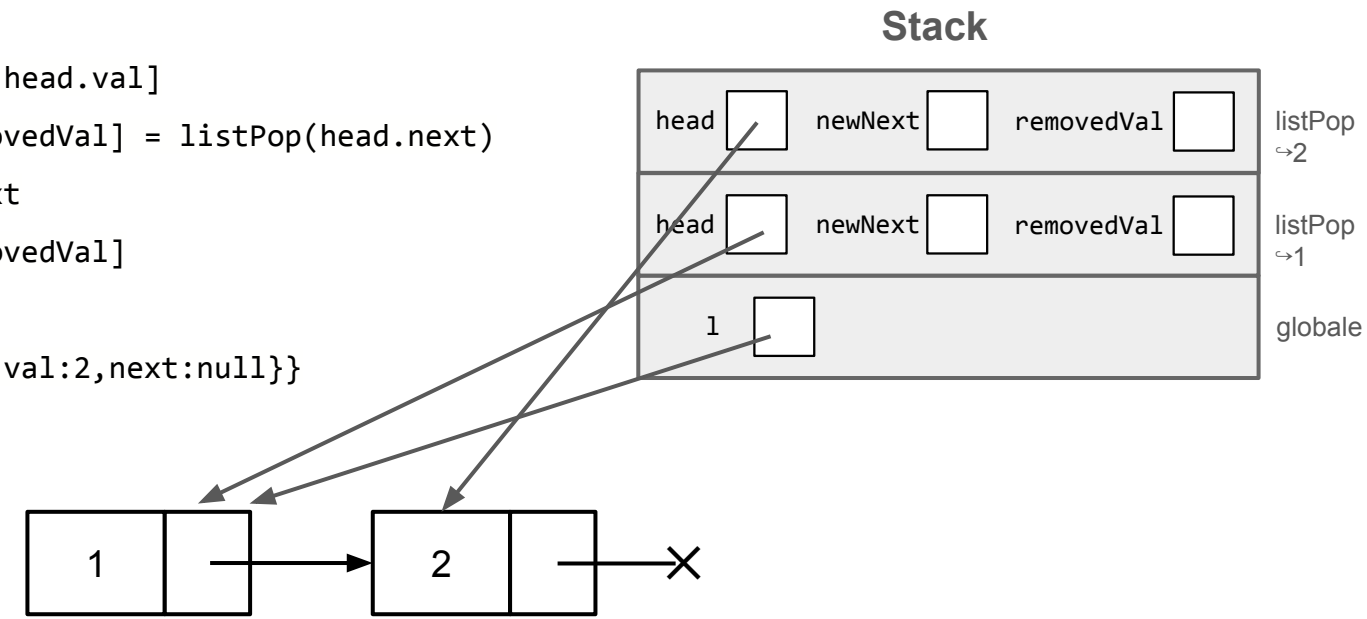
listPop(head): Rimuove il nodo in coda e restituisce una coppia [testa aggiornata, valore rimosso]

```
1.  function listPop(head) {
2.      if (!head)
3.          return [null, undefined]
4.      if (!head.next)
5.          return [null, head.val]
6.      let [newNext, removedVal] = listPop(head.next)
7.      head.next = newNext
8.      return [head, removedVal]
9.  }
10. let l = {val:1,next:{val:2,next:null}}
11. l = listPop(l)[0]
```



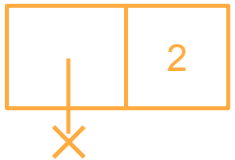
listPop(head): Rimuove il nodo in coda e restituisce una coppia [testa aggiornata, valore rimosso]

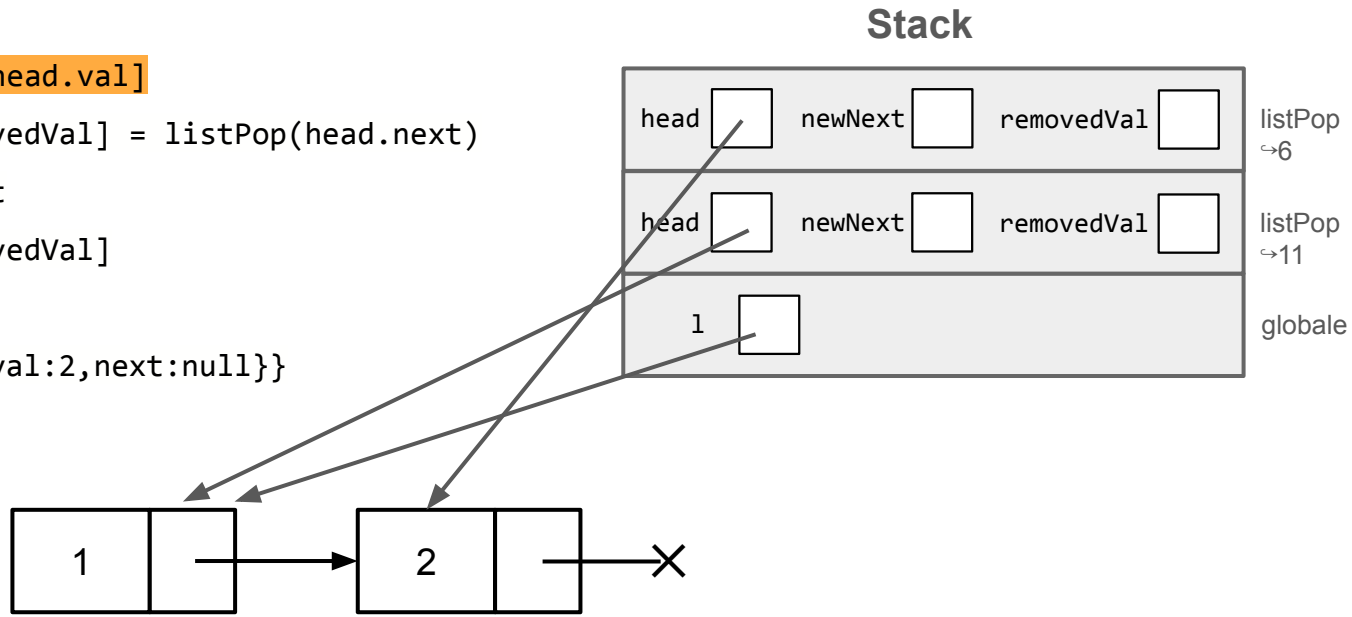
```
1.  function listPop(head) {
2.      if (!head)
3.          return [null, undefined]
4.      if (!head.next)
5.          return [null, head.val]
6.      let [newNext, removedVal] = listPop(head.next)
7.      head.next = newNext
8.      return [head, removedVal]
9.  }
10. let l = {val:1,next:{val:2,next:null}}
11. l = listPop(l)[0]
```



listPop(head): Rimuove il nodo in coda e restituisce una coppia [testa aggiornata, valore rimosso]

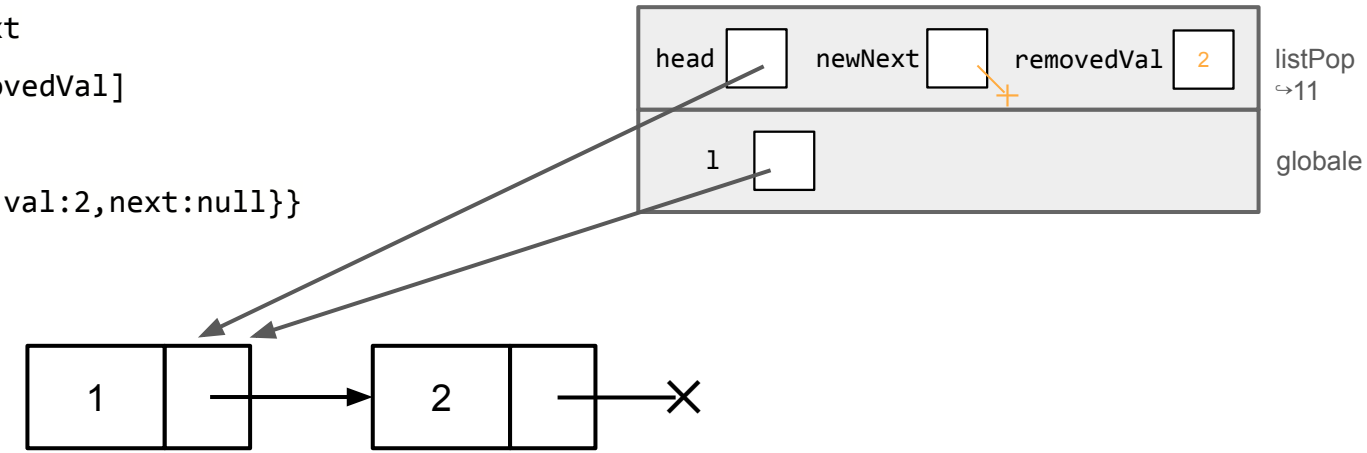
```
1.  function listPop(head) {
2.      if (!head)
3.          return [null, undefined]
4.      if (!head.next)
5.          return [null, head.val]
6.      let [newNext, removedVal] = listPop(head.next)
7.      head.next = newNext
8.      return [head, removedVal]
9.  }
10. let l = {val:1,next:{val:2,next:null}}
11. l = listPop(l)[0]
```





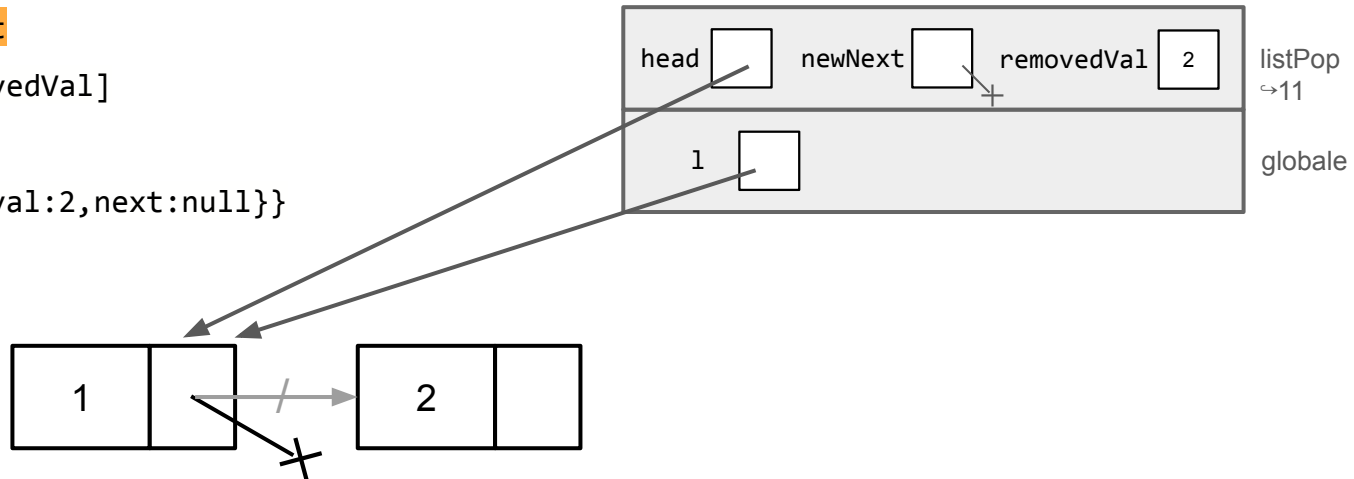
listPop(head): Rimuove il nodo in coda e restituisce una coppia [testa aggiornata, valore rimosso]

```
1.  function listPop(head) {
2.      if (!head)
3.          return [null, undefined]
4.      if (!head.next)
5.          return [null, head.val]
6.      let [newNext, removedVal] = listPop(head.next)
7.      head.next = newNext
8.      return [head, removedVal]
9.  }
10. let l = {val:1,next:{val:2,next:null}}
11. l = listPop(l)[0]
```



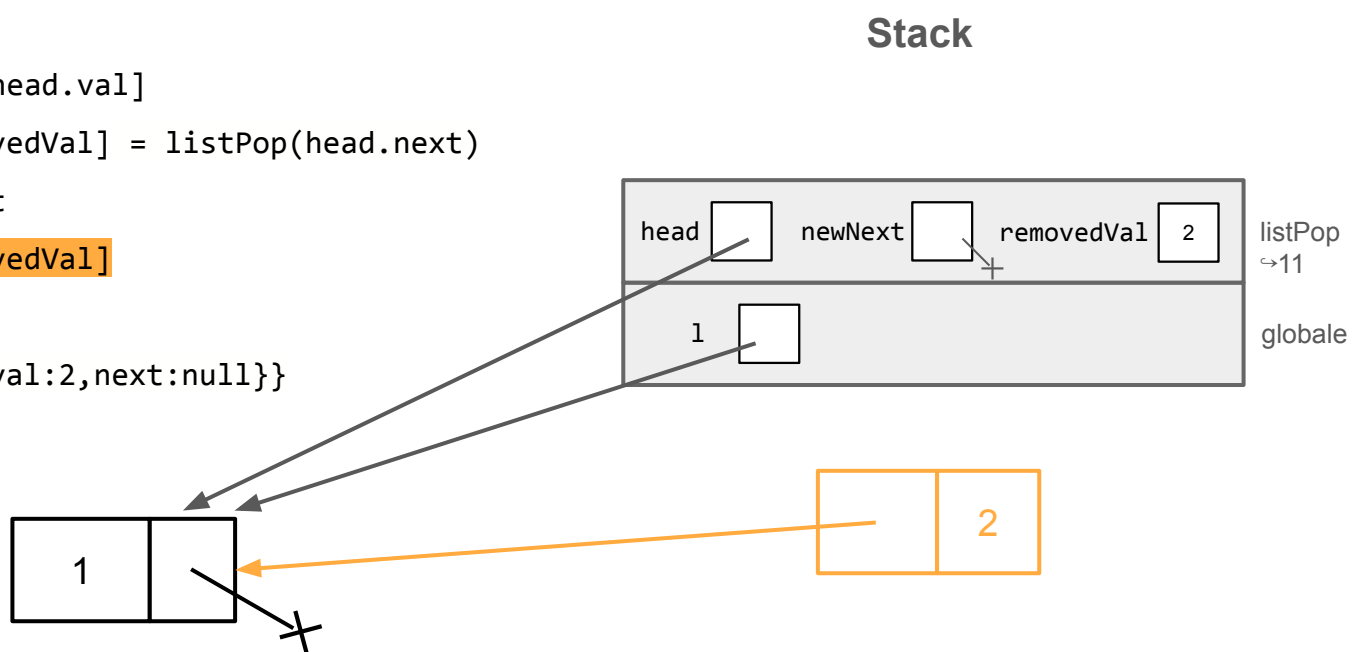
listPop(head): Rimuove il nodo in coda e restituisce una coppia [testa aggiornata, valore rimosso]

```
1.  function listPop(head) {
2.      if (!head)
3.          return [null, undefined]
4.      if (!head.next)
5.          return [null, head.val]
6.      let [newNext, removedVal] = listPop(head.next)
7.      head.next = newNext
8.      return [head, removedVal]
9.  }
10. let l = {val:1,next:{val:2,next:null}}
11. l = listPop(l)[0]
```



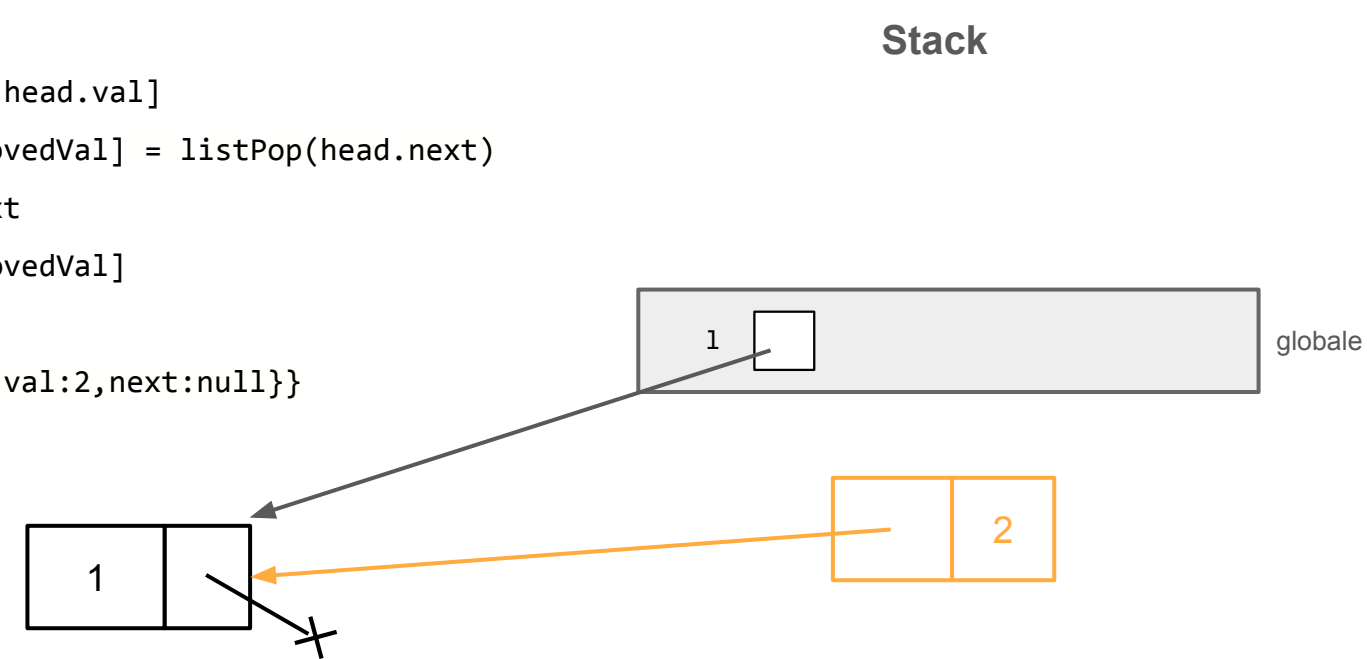
listPop(head): Rimuove il nodo in coda e restituisce una coppia [testa aggiornata, valore rimosso]

```
1.  function listPop(head) {
2.      if (!head)
3.          return [null, undefined]
4.      if (!head.next)
5.          return [null, head.val]
6.      let [newNext, removedVal] = listPop(head.next)
7.      head.next = newNext
8.      return [head, removedVal]
9.  }
10. let l = {val:1,next:{val:2,next:null}}
11. l = listPop(l)[0]
```



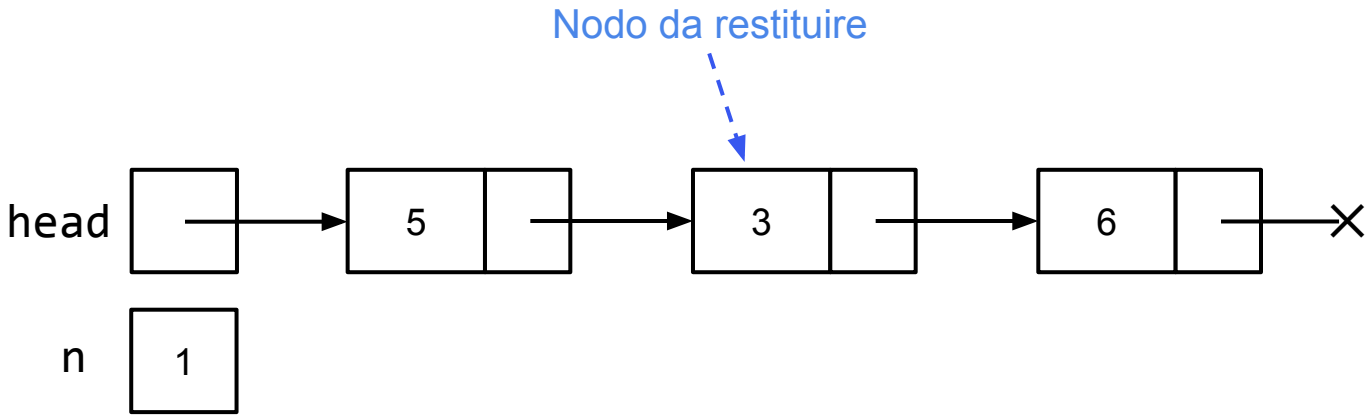
listPop(head): Rimuove il nodo in coda e restituisce una coppia [testa aggiornata, valore rimosso]

```
1.  function listPop(head) {
2.      if (!head)
3.          return [null, undefined]
4.      if (!head.next)
5.          return [null, head.val]
6.      let [newNext, removedVal] = listPop(head.next)
7.      head.next = newNext
8.      return [head, removedVal]
9.  }
10. let l = {val:1,next:{val:2,next:null}}
11. l = listPop(l)[0]
```



Esercizio 1

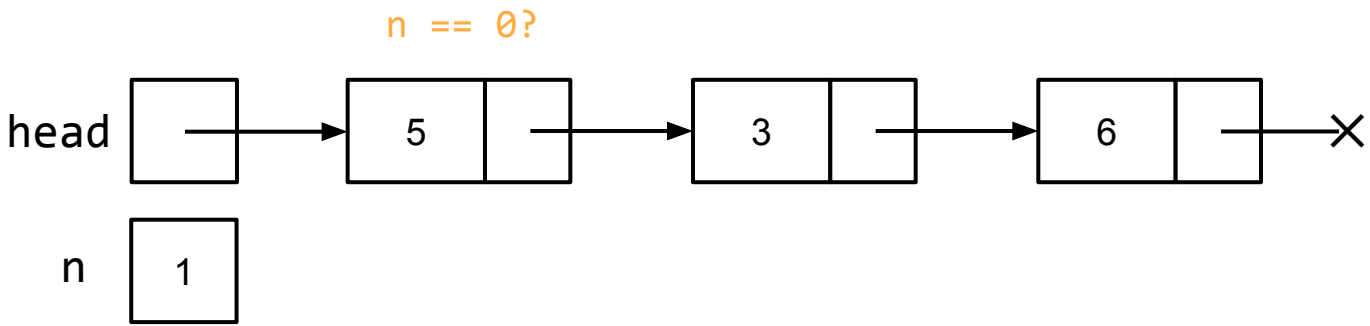
Scrivere una funzione ricorsiva `listNth(head, n)` che restituisce il nodo alla posizione `n`. Si assuma che il primo nodo abbia indice 0.



Esercizio 1 - Soluzione

Scrivere una funzione ricorsiva `listNth(head, n)` che restituisce il nodo alla posizione `n`. Si assuma che il primo nodo abbia indice 0.

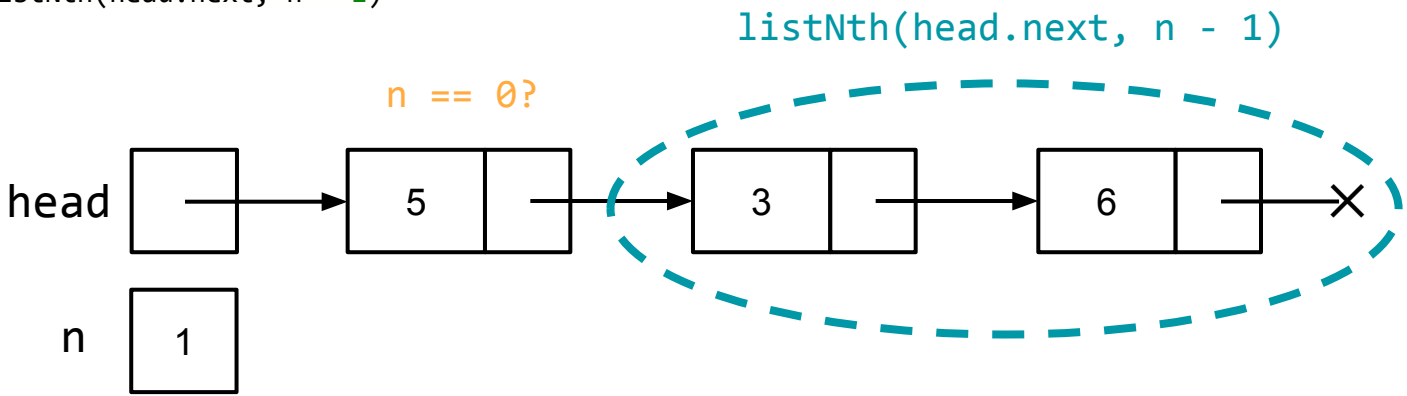
```
1.  function listNth(head, n) {
2.      if (!head)
3.          return null
4.      if (n == 0)
5.          return head
6.      return listNth(head.next, n - 1)
7.  }
```



Esercizio 1 - Soluzione

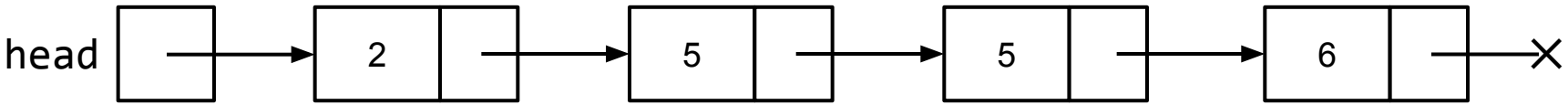
Scrivere una funzione ricorsiva `listNth(head, n)` che restituisce il nodo alla posizione `n`. Si assuma che il primo nodo abbia indice 0.

```
1.  function listNth(head, n) {
2.      if (!head)
3.          return null
4.      if (n == 0)
5.          return head
6.      return listNth(head.next, n - 1)
7.  }
```



Esercizio 2

Scrivere una funzione ricorsiva `listIsSorted(head)` che restituisce `true` se e solo se la lista è ordinata in modo non decrescente

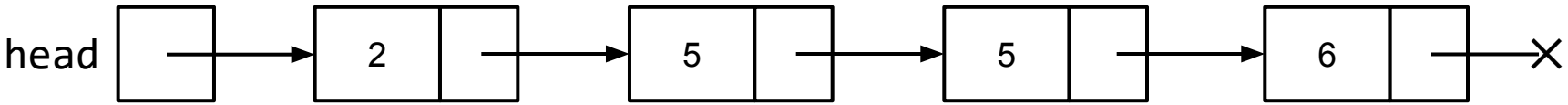


Risultato atteso: `true`

Esercizio 2 - Soluzione

Scrivere una funzione ricorsiva `listIsSorted(head)` che restituisce `true` se e solo se la lista è ordinata in modo non decrescente

```
1.  function listIsSorted(head) {
2.      if (!head || !head.next)
3.          return true
4.      return head.val <= head.next.val && listIsSorted(head.next)
5.  }
```

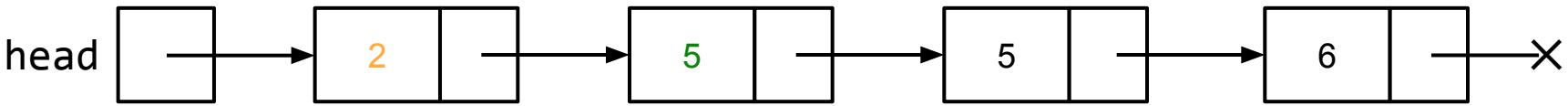


Esercizio 2 - Soluzione

Scrivere una funzione ricorsiva `listIsSorted(head)` che restituisce `true` se e solo se la lista è ordinata in modo non decrescente

```
1.  function listIsSorted(head) {
2.      if (!head || !head.next)
3.          return true
4.      return head.val <= head.next.val && listIsSorted(head.next)
5.  }
```

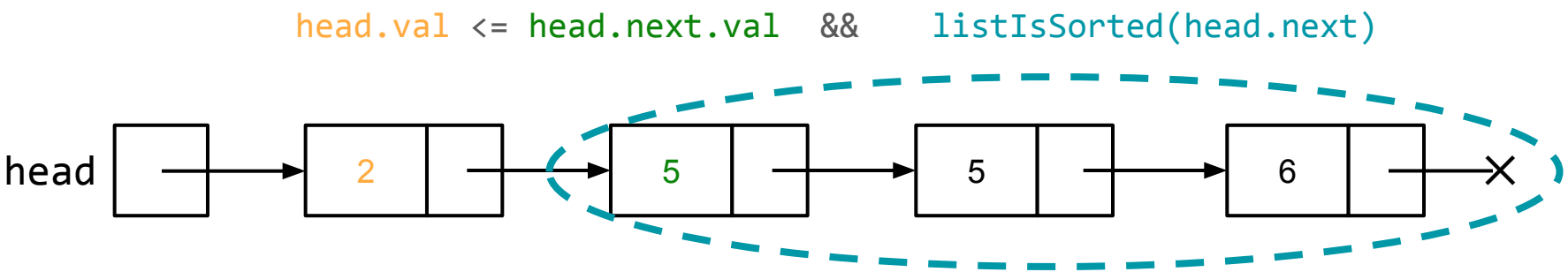
`head.val <= head.next.val`?



Esercizio 2 - Soluzione

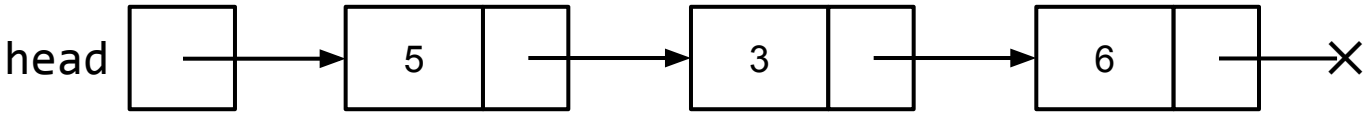
Scrivere una funzione ricorsiva `listIsSorted(head)` che restituisce `true` se e solo se la lista è ordinata in modo non decrescente

```
1.  function listIsSorted(head) {
2.      if (!head || !head.next)
3.          return true
4.      return head.val <= head.next.val && listIsSorted(head.next)
5.  }
```



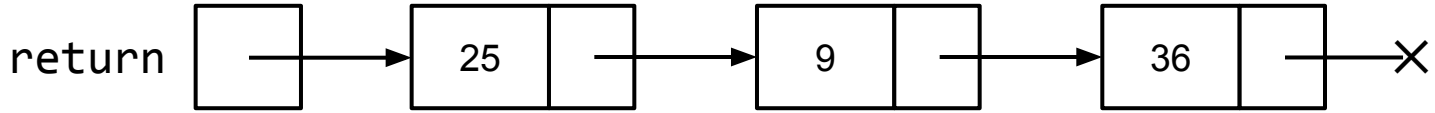
Esercizio 3

Scrivere una funzione ricorsiva `listMap(head, f)` che restituisce una *nuova* lista i cui valori sono il risultato dell'applicazione di `f` ai valori della lista originale




```
f = x => x ** 2
```

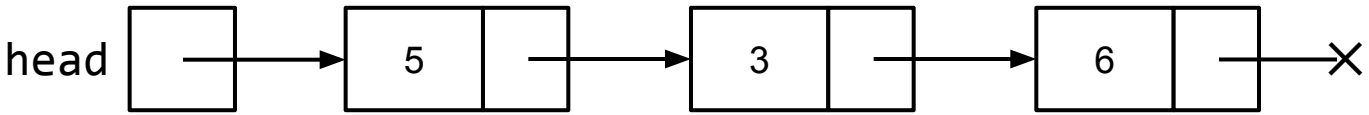
Risultato atteso:



Esercizio 3 - Soluzione

Scrivere una funzione ricorsiva `listMap(head, f)` che restituisce una *nuova* lista i cui valori sono il risultato dell'applicazione di `f` ai valori della lista originale

```
1.  function listMap(head, f) {
2.      if (!head)
3.          return null
4.      return { val: f(head.val), next: listMap(head.next, f) }
5.  }
```

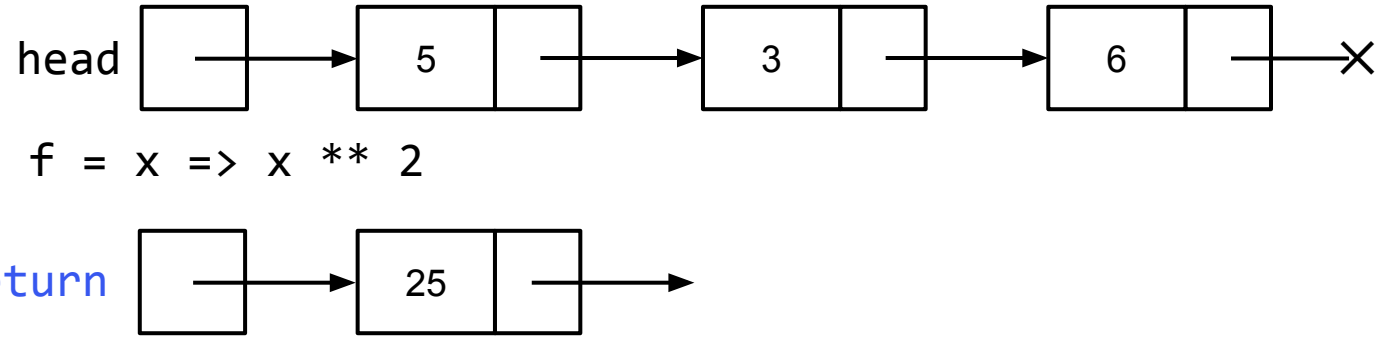



```
f = x => x ** 2
```

Esercizio 3 - Soluzione

Scrivere una funzione ricorsiva `listMap(head, f)` che restituisce una *nuova* lista i cui valori sono il risultato dell'applicazione di `f` ai valori della lista originale

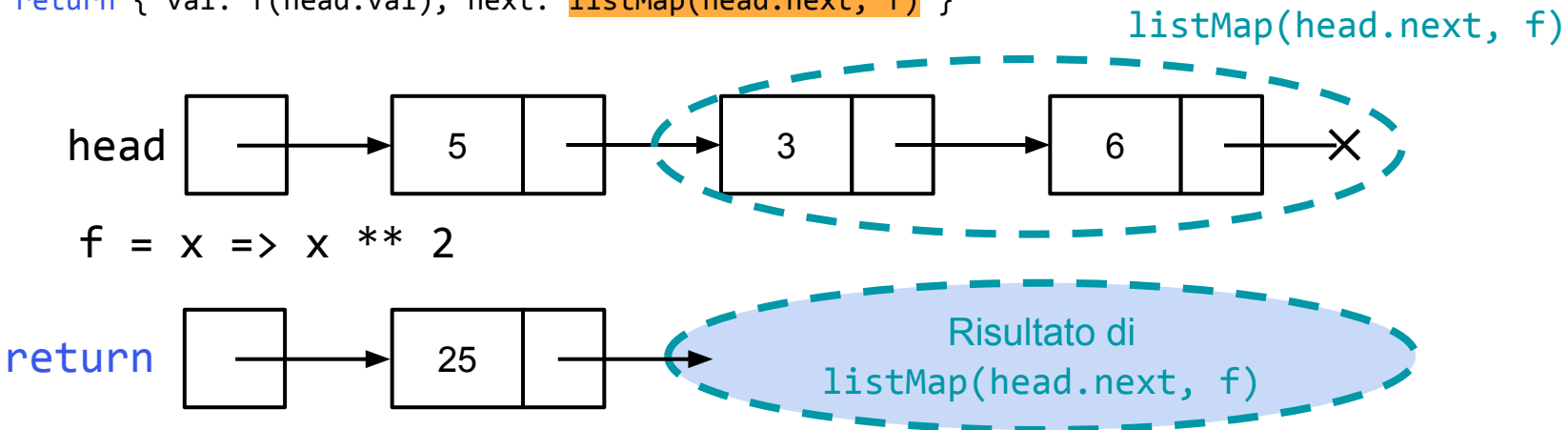
```
1.  function listMap(head, f) {
2.      if (!head)
3.          return null
4.      return { val: f(head.val), next: listMap(head.next, f) }
5.  }
```



Esercizio 3 - Soluzione

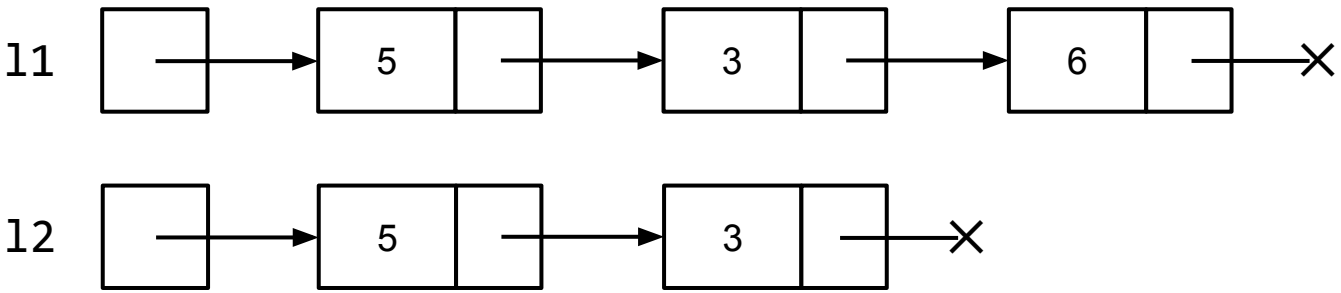
Scrivere una funzione ricorsiva `listMap(head, f)` che restituisce una *nuova* lista i cui valori sono il risultato dell'applicazione di `f` ai valori della lista originale

```
1.  function listMap(head, f) {
2.      if (!head)
3.          return null
4.      return { val: f(head.val), next: listMap(head.next, f) }
5.  }
```



Esercizio 4

Scrivere una funzione ricorsiva `listEquals(l1, l2)` che restituisce true se e solo se le due liste sono uguali, ossia hanno gli stessi valori nello stesso ordine



Risultato atteso: `false`

Esercizio 4 - Soluzione

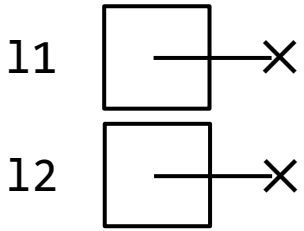
Scrivere una funzione ricorsiva `listEquals(l1, l2)` che restituisce true se e solo se le due liste sono uguali, ossia hanno gli stessi valori nello stesso ordine

```
1.  function listEquals(l1, l2) {
2.      if (!l1 && !l2)
3.          return true
4.      if (!l1 || !l2)
5.          return false
6.      return l1.val === l2.val && listEquals(l1.next, l2.next)
7.  }
```

Esercizio 4 - Soluzione

Scrivere una funzione ricorsiva `listEquals(l1, l2)` che restituisce true se e solo se le due liste sono uguali, ossia hanno gli stessi valori nello stesso ordine

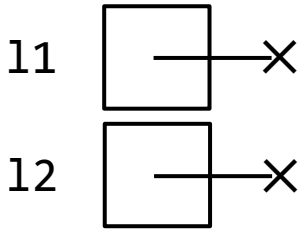
```
1.  function listEquals(l1, l2) {
2.      if (!l1 && !l2)
3.          return true
4.      if (!l1 || !l2)
5.          return false
6.      return l1.val === l2.val && listEquals(l1.next, l2.next)
7.  }
```



Esercizio 4 - Soluzione

Scrivere una funzione ricorsiva `listEquals(l1, l2)` che restituisce true se e solo se le due liste sono uguali, ossia hanno gli stessi valori nello stesso ordine

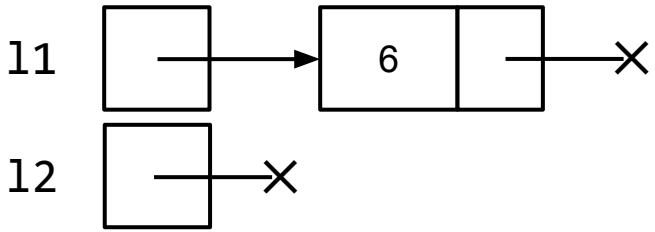
```
1.  function listEquals(l1, l2) {
2.      if (!l1 && !l2)
3.          return true
4.      if (!l1 || !l2)
5.          return false
6.      return l1.val === l2.val && listEquals(l1.next, l2.next)
7.  }
```



Esercizio 4 - Soluzione

Scrivere una funzione ricorsiva `listEquals(l1, l2)` che restituisce true se e solo se le due liste sono uguali, ossia hanno gli stessi valori nello stesso ordine

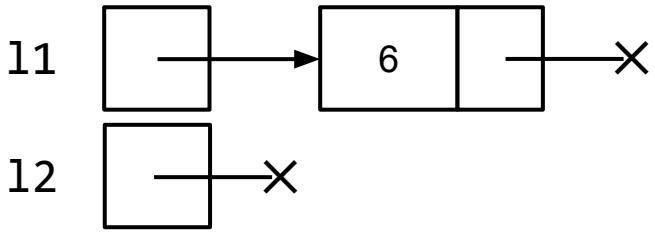
```
1.  function listEquals(l1, l2) {
2.      if (!l1 && !l2)
3.          return true
4.      if (!l1 || !l2)
5.          return false
6.      return l1.val === l2.val && listEquals(l1.next, l2.next)
7.  }
```



Esercizio 4 - Soluzione

Scrivere una funzione ricorsiva `listEquals(l1, l2)` che restituisce true se e solo se le due liste sono uguali, ossia hanno gli stessi valori nello stesso ordine

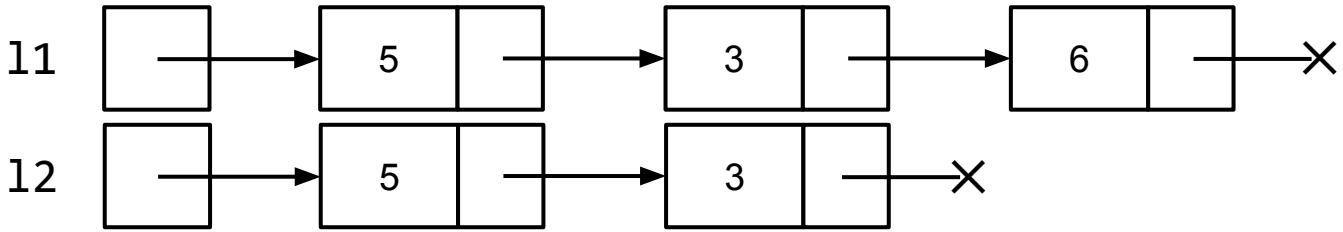
```
1.  function listEquals(l1, l2) {
2.      if (!l1 && !l2)
3.          return true
4.      if (!l1 || !l2)
5.          return false
6.      return l1.val === l2.val && listEquals(l1.next, l2.next)
7.  }
```



Esercizio 4 - Soluzione

Scrivere una funzione ricorsiva `listEquals(l1, l2)` che restituisce true se e solo se le due liste sono uguali, ossia hanno gli stessi valori nello stesso ordine

```
1.  function listEquals(l1, l2) {
2.      if (!l1 && !l2)
3.          return true
4.      if (!l1 || !l2)
5.          return false
6.      return l1.val === l2.val && listEquals(l1.next, l2.next)
7.  }
```

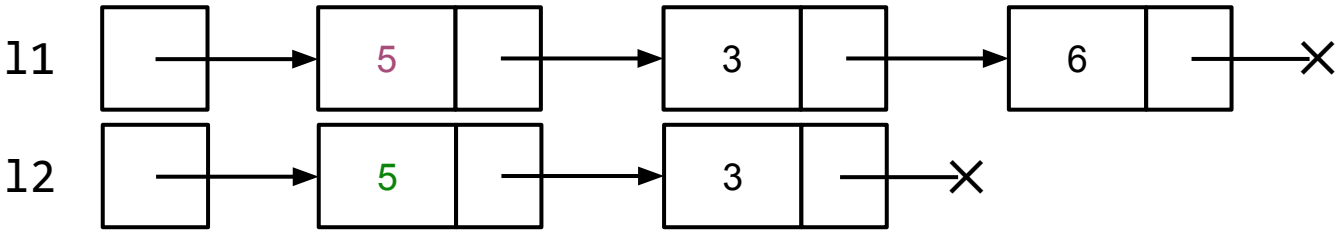


Esercizio 4 - Soluzione

Scrivere una funzione ricorsiva `listEquals(l1, l2)` che restituisce true se e solo se le due liste sono uguali, ossia hanno gli stessi valori nello stesso ordine

```
1.  function listEquals(l1, l2) {
2.      if (!l1 && !l2)
3.          return true
4.      if (!l1 || !l2)
5.          return false
6.      return l1.val === l2.val && listEquals(l1.next, l2.next)
7.  }
```

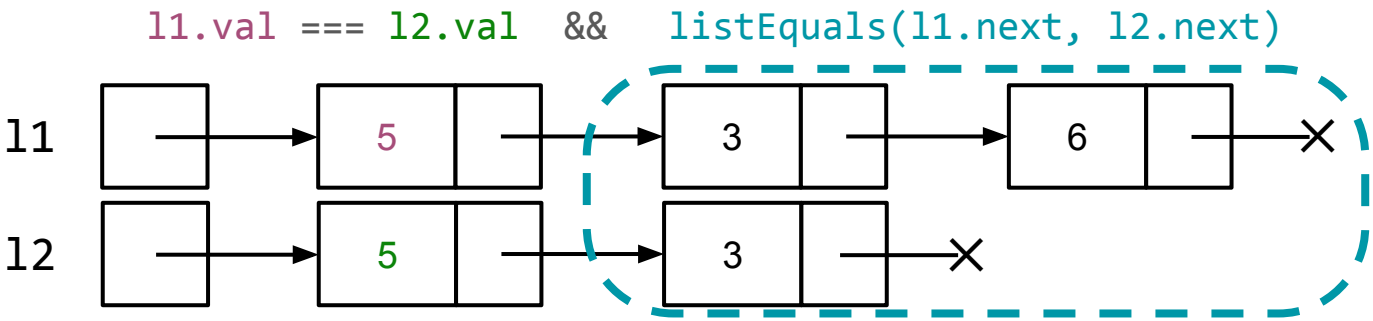
`l1.val === l2.val &&`



Esercizio 4 - Soluzione

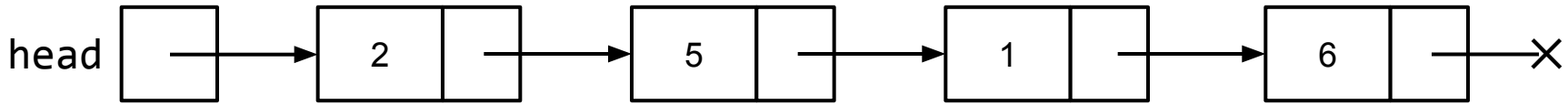
Scrivere una funzione ricorsiva `listEquals(l1, l2)` che restituisce true se e solo se le due liste sono uguali, ossia hanno gli stessi valori nello stesso ordine

```
1.  function listEquals(l1, l2) {
2.      if (!l1 && !l2)
3.          return true
4.      if (!l1 || !l2)
5.          return false
6.      return l1.val === l2.val && listEquals(l1.next, l2.next)
7.  }
```

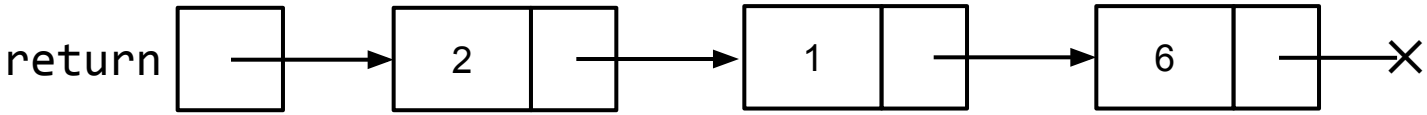


Esercizio 5

Scrivere una funzione ricorsiva `listSplice(head, start, deleteCount)` che restituisce una lista ottenuta rimuovendo `deleteCount` nodi a partire dalla posizione `start`. Si assuma che il primo nodo abbia indice 0



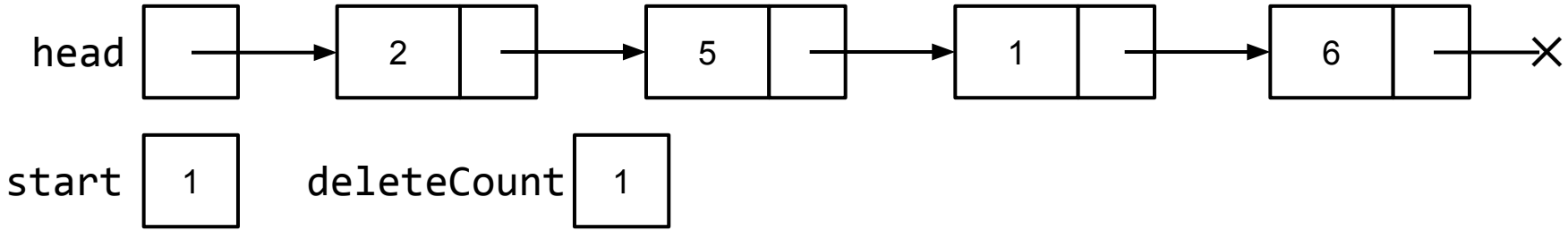
Risultato atteso di listSplice(head, 1, 1):



Esercizio 5 - Soluzione

Scrivere una funzione ricorsiva `listSplice(head, start, deleteCount)` che restituisce una lista ottenuta rimuovendo `deleteCount` nodi a partire dalla posizione `start`. Si assuma che il primo nodo abbia indice 0

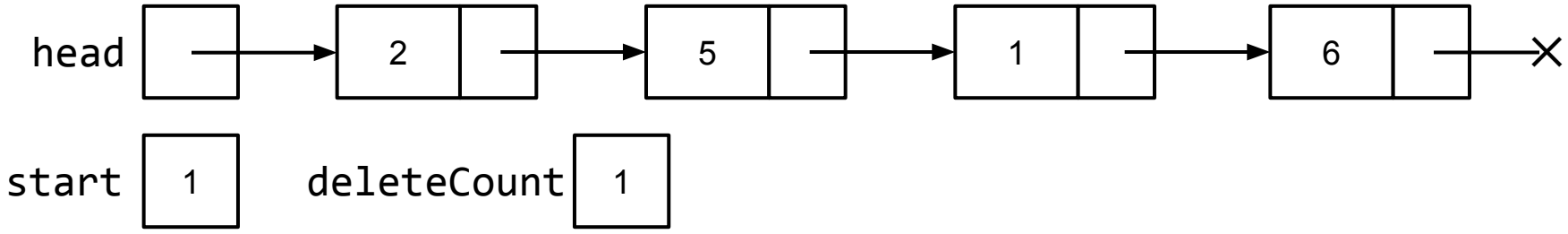
```
1.  function listSplice(head, start, deleteCount) {
2.      if (!head) return null
3.      if (start > 0) // siamo prima del punto di rimozione
4.          return { val: head.val, next: listSplice(head.next, start - 1, deleteCount) }
5.      // siamo nel punto di rimozione
6.      if (deleteCount > 0)
7.          return listSplice(head.next, 0, deleteCount - 1)
8.      return head
9.  }
```



Esercizio 5 - Soluzione

Scrivere una funzione ricorsiva `listSplice(head, start, deleteCount)` che restituisce una lista ottenuta rimuovendo `deleteCount` nodi a partire dalla posizione `start`. Si assuma che il primo nodo abbia indice 0

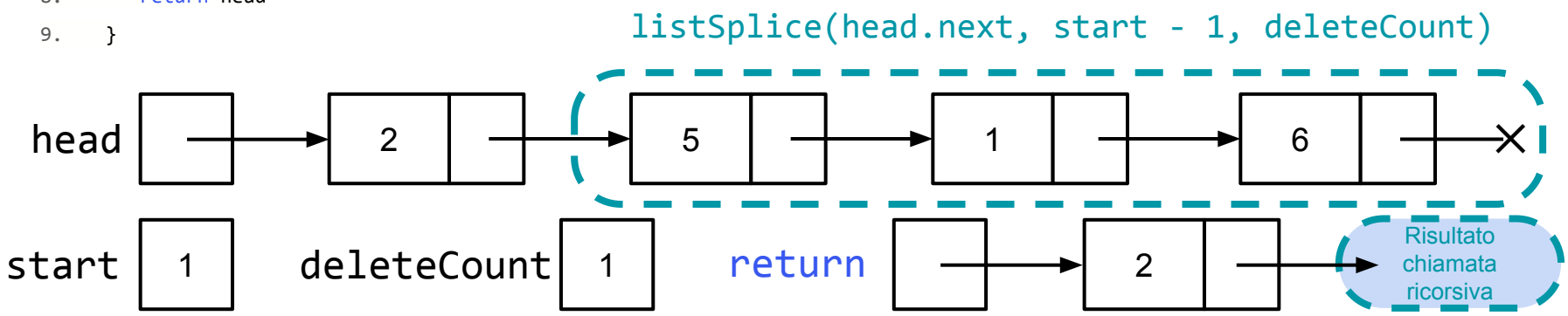
```
1.  function listSplice(head, start, deleteCount) {
2.      if (!head) return null
3.      if (start > 0) // siamo prima del punto di rimozione
4.          return { val: head.val, next: listSplice(head.next, start - 1, deleteCount) }
5.      // siamo nel punto di rimozione
6.      if (deleteCount > 0)
7.          return listSplice(head.next, 0, deleteCount - 1)
8.      return head
9.  }
```



Esercizio 5 - Soluzione

Scrivere una funzione ricorsiva `listSplice(head, start, deleteCount)` che restituisce una lista ottenuta rimuovendo `deleteCount` nodi a partire dalla posizione `start`. Si assuma che il primo nodo abbia indice 0

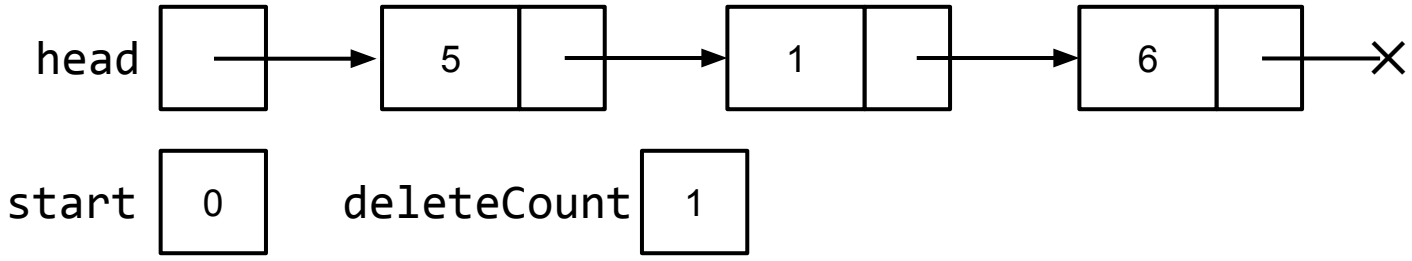
```
1.  function listSplice(head, start, deleteCount) {
2.      if (!head) return null
3.      if (start > 0) // siamo prima del punto di rimozione
4.          return { val: head.val, next: listSplice(head.next, start - 1, deleteCount) }
5.      // siamo nel punto di rimozione
6.      if (deleteCount > 0)
7.          return listSplice(head.next, 0, deleteCount - 1)
8.      return head
9.  }
```



Esercizio 5 - Soluzione

Scrivere una funzione ricorsiva `listSplice(head, start, deleteCount)` che restituisce una lista ottenuta rimuovendo `deleteCount` nodi a partire dalla posizione `start`. Si assuma che il primo nodo abbia indice 0

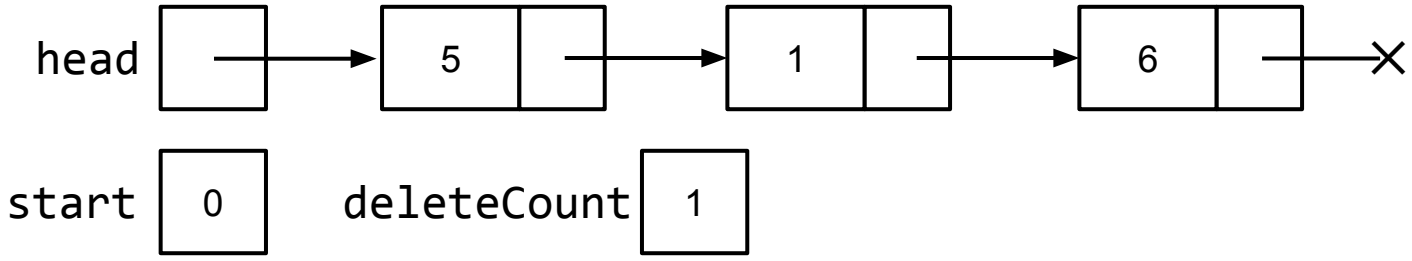
```
1.  function listSplice(head, start, deleteCount) {
2.      if (!head) return null
3.      if (start > 0) // siamo prima del punto di rimozione
4.          return { val: head.val, next: listSplice(head.next, start - 1, deleteCount) }
5.      // siamo nel punto di rimozione
6.      if (deleteCount > 0)
7.          return listSplice(head.next, 0, deleteCount - 1)
8.      return head
9.  }
```



Esercizio 5 - Soluzione

Scrivere una funzione ricorsiva `listSplice(head, start, deleteCount)` che restituisce una lista ottenuta rimuovendo `deleteCount` nodi a partire dalla posizione `start`. Si assuma che il primo nodo abbia indice 0

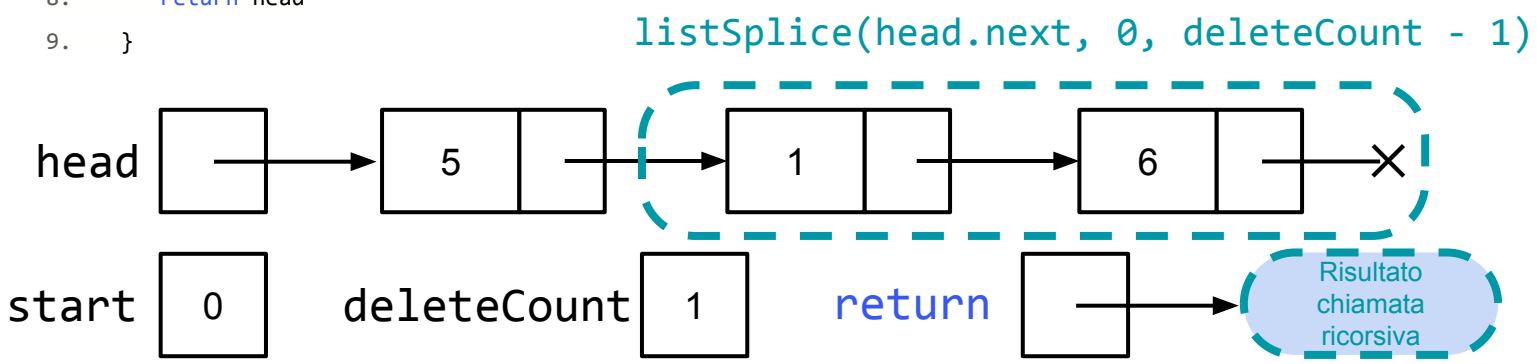
```
1.  function listSplice(head, start, deleteCount) {
2.      if (!head) return null
3.      if (start > 0) // siamo prima del punto di rimozione
4.          return { val: head.val, next: listSplice(head.next, start - 1, deleteCount) }
5.      // siamo nel punto di rimozione
6.      if (deleteCount > 0)
7.          return listSplice(head.next, 0, deleteCount - 1)
8.      return head
9.  }
```



Esercizio 5 - Soluzione

Scrivere una funzione ricorsiva `listSplice(head, start, deleteCount)` che restituisce una lista ottenuta rimuovendo `deleteCount` nodi a partire dalla posizione `start`. Si assuma che il primo nodo abbia indice 0

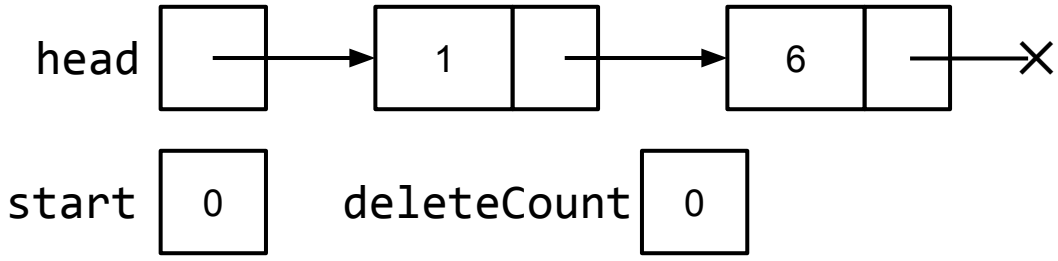
```
1.  function listSplice(head, start, deleteCount) {
2.      if (!head) return null
3.      if (start > 0) // siamo prima del punto di rimozione
4.          return { val: head.val, next: listSplice(head.next, start - 1, deleteCount) }
5.      // siamo nel punto di rimozione
6.      if (deleteCount > 0)
7.          return listSplice(head.next, 0, deleteCount - 1)
8.      return head
9.  }
```



Esercizio 5 - Soluzione

Scrivere una funzione ricorsiva `listSplice(head, start, deleteCount)` che restituisce una lista ottenuta rimuovendo `deleteCount` nodi a partire dalla posizione `start`. Si assuma che il primo nodo abbia indice 0

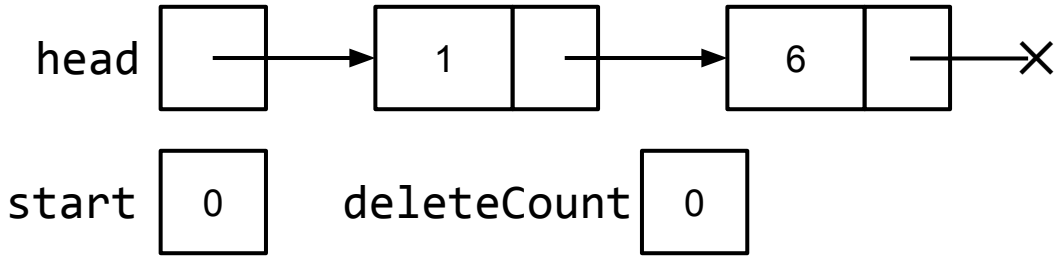
```
1.  function listSplice(head, start, deleteCount) {
2.      if (!head) return null
3.      if (start > 0) // siamo prima del punto di rimozione
4.          return { val: head.val, next: listSplice(head.next, start - 1, deleteCount) }
5.      // siamo nel punto di rimozione
6.      if (deleteCount > 0)
7.          return listSplice(head.next, 0, deleteCount - 1)
8.      return head
9.  }
```



Esercizio 5 - Soluzione

Scrivere una funzione ricorsiva `listSplice(head, start, deleteCount)` che restituisce una lista ottenuta rimuovendo `deleteCount` nodi a partire dalla posizione `start`. Si assuma che il primo nodo abbia indice 0

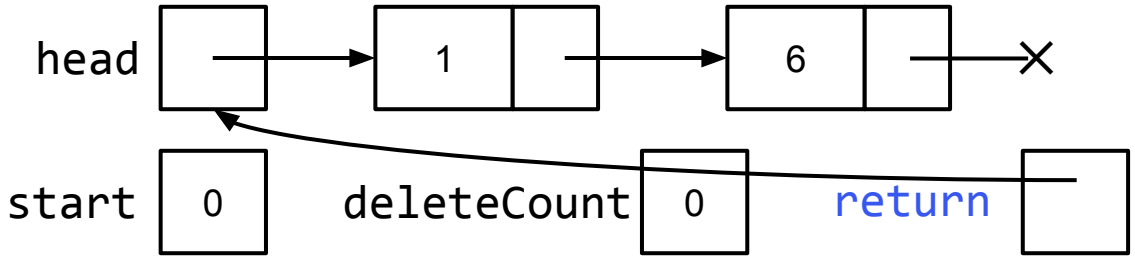
```
1.  function listSplice(head, start, deleteCount) {
2.      if (!head) return null
3.      if (start > 0) // siamo prima del punto di rimozione
4.          return { val: head.val, next: listSplice(head.next, start - 1, deleteCount) }
5.      // siamo nel punto di rimozione
6.      if (deleteCount > 0)
7.          return listSplice(head.next, 0, deleteCount - 1)
8.      return head
9.  }
```



Esercizio 5 - Soluzione

Scrivere una funzione ricorsiva `listSplice(head, start, deleteCount)` che restituisce una lista ottenuta rimuovendo `deleteCount` nodi a partire dalla posizione `start`. Si assuma che il primo nodo abbia indice 0

```
1.  function listSplice(head, start, deleteCount) {
2.      if (!head) return null
3.      if (start > 0) // siamo prima del punto di rimozione
4.          return { val: head.val, next: listSplice(head.next, start - 1, deleteCount) }
5.      // siamo nel punto di rimozione
6.      if (deleteCount > 0)
7.          return listSplice(head.next, 0, deleteCount - 1)
8.      return head
9.  }
```



Q & A