Strutture dati elementari - Esercizi

Pietro Di Lena

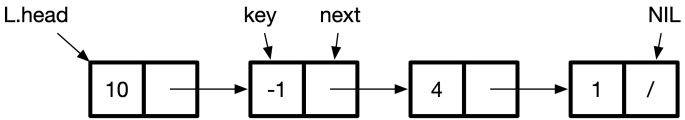
Dipartimento di Informatica – Scienza e Ingegneria
Università di Bologna

Algoritmi e Strutture di Dati
Anno Accademico 2022/2023

Esercizio 1

- Scrivere un algoritmo che, dati in input un intero positivo k ed una lista concatenata semplice, restituisca il k-esimo elemento a partire dalla fine

- Esempio, se $k = 1$ l'algoritmo deve restituire l'ultimo elemento; se $k = 2$ il penultimo, e cosi via



Esercizio 1 - Soluzione

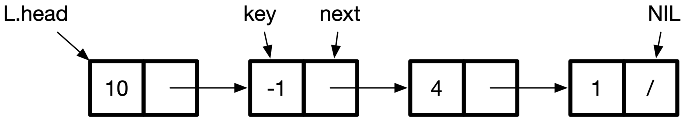
```
1: function LASTK(List L, Int k) → Node
2:     tmp = L.head                          ▷ Temporary pointer
3:     n = 0                                 ▷ Number of nodes
4:     while tmp ≠ NIL do                    ▷ Count the number of nodes
5:         n = n + 1
6:         tmp = tmp.next
7:     if n < k then
8:         return NIL
9:     else
10:        tmp = L.head                      ▷ Reset tmp
11:        n = n − k + 1                     ▷ Index of last-k node
12:        i = 1
13:        while i < n do                    ▷ Search the last-k node
14:            tmp = tmp.next
15:            i = i + 1
16:        return tmp
```

- Costo computazionale: $\Theta(n)$ (visitiamo sempre tutta L: linee 4-6)
- $n =$ numero di nodi nella lista

Esercizio 2

- Scrivere un algoritmo che, dati in input un intero positivo k ed una lista concatenata semplice restituisca il k-esimo elemento a partire dalla fine senza visitare per due volte la lista (per memorizzarne la lunghezza)

- Esempio, se $k = 1$ l'algoritmo deve restituire l'ultimo elemento; se $k = 2$ il penultimo, e cosi via



Esercizio 2 - Soluzione

```
function lastk(List L, Int k) → Node
    tmp1 = L.head                                   ▷ Temporary pointer
    tmp2 = L.head                                   ▷ Temporary pointer
    i = 0                                           ▷ Counter
    while tmp1 ≠ NIL and i < k do                   ▷ Move tmp2 on the k-th node
        i = i + 1
        tmp1 = tmp1.next
    if i < k then                                   ▷ There are less than k nodes
        return NIL
    else
                        ▷ The distance between tmp1 and tmp2 is exactly k
        while tmp1 ≠ NIL do
            tmp1 = tmp1.next
            tmp2 = tmp2.next
        return tmp2
```

- Costo computazionale: $\Theta(n)$ (*tmp1* visita sempre tutti i nodi di L)
- n = numero di nodi nella lista

Esercizio 3

- Scrivere un algoritmo ricorsivo che, data una lista concatenata semplice, la modifichi eliminando ogni elemento pari
- Esempio, se $L = [4, 6, 7, 3, 2, 5]$ al termine dell'esecuzione $L = [7, 3, 5]$

Esercizio 3 - Soluzione

```
function deleteEven(Node L) → Node
    if L == NIL then
        return NIL
    else if L.key mod 2 == 0 then                ▷ Even node, remove
        return deleteEven(L.next)
    else                                         ▷ Odd node, no remove
        return L.next = deleteEven(L.next)
        return L
```

- Costo computazionale: $\Theta(n)$ (visita sempre tutti i nodi della lista)

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + 1 & n > 0 \end{cases}$$

- n = numero di nodi nella lista

Esercizio 4

- Scrivere un algoritmo ricorsivo che, data una lista concatenata semplice, la modifichi eliminando ogni elemento pari e replicando ogni elemento dispari tante volte quanti sono gli elementi pari che lo precedono
- Esempio, se $L = [4, 6, 7, 3, 2, 5]$ al termine dell'esecuzione abbiamo che $L = [7, 7, 7, 3, 3, 3, 5, 5, 5, 5]$

Esercizio 4 - Soluzione 1

```
function deleteAndDuplicate(Node L, int nPrec) → Node
    if L == NIL then
        return NIL
    else if L.key mod 2 == 0 then                        ▷ Even node, remove
        return deleteAndDuplicate(L.next, nPrec + 1)
    else                                                  ▷ Odd node, duplicate
        L.next = deleteAndDuplicate(L.next, nPrec)
        while nPrec > 0 do                 ▷ Duplication with nPrec head insert
            tmp = new List(L.key)
            tmp.next = L
            L = tmp
            nPrec = nPrec − 1
        return L
```

- Soluzione mista ricorsiva/iterativa
- Prima esecuzione: deleteAndDuplicate$(L, 0)$

Esercizio 4 - Soluzione 2

```
function deleteAndDuplicate(Node L, int nPrec, int nDup) → Node
    if L == NIL then
        return NIL
    else if L.key mod 2 == 0 then                          ▷ Even node, remove
        return deleteAndDuplicate(L.next,nPrec + 1,nPrec + 1)
    else if nPrec > 0 then                                 ▷ Odd node, duplicate
        tmp = new List(L.key)
        tmp.next = deleteAndDuplicate(L,nPrec,nPrec − 1)
        return tmp
    else                                                   ▷ Odd node, no duplicate
        L.next = deleteAndDuplicate(L.next,nPrec,nPrec)
        return L
```

- Soluzione interamente ricorsiva
- Prima esecuzione: deleteAndDuplicate$(L, 0, 0)$

Esercizio 4 - Analisi del costo computazionale

- Vale sia per la versione mista che per la versione solo ricorsiva
- Assumiamo $n =$ numero di nodi nella lista
- Costo ottimo: $\Theta(n)$ (ci sono solo nodi pari o solo nodi dispari)
- Caso pessimo (irrealistico): ogni nodo viene duplicato tante volte quanti sono i nodi che lo precedono

$$T(n) \leq \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2 - n}{2} = O(n^2)$$

  Quindi il costo pessimo T(n) è limitato superiormente da $O(n^2)$

- Caso reale: un nodo pari ed un nodo dispari, alternati

$$T'(n) = \sum_{i=1}^{n/2} i = \frac{n/2(n/2+1)}{2} = \frac{n^2}{8} + \frac{n}{4} = \Theta(n^2)$$

- Poiché $T'(n) = O(T(n))$ concludiamo che il costo pessimo è $\Theta(n^2)$

Esercizio 5

- Scrivere un algoritmo che, presi in input una lista concatenata semplice con chiavi intere ed un intero x, calcoli il numero di nodi con chiave x
- Scrivere un algoritmo ricorsivo ed uno iterativo

Esercizio 5 - Soluzione

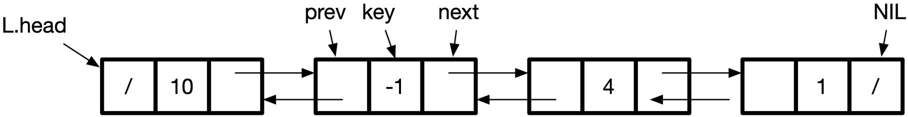
```
function COUNT(LIST L, INT x) → INT            ▷ Soluzione iterativa
    tmp = L.head, n = 0
    while tmp ≠ NIL do
        if tmp.key == x then
            n = n + 1
        tmp = tmp.next
    return n
```

```
function COUNT(NODE L, INT x) → INT            ▷ Soluzione ricorsiva
    if L == NIL then
        return 0
    else if L.key == x then
        return 1 + COUNT(L.next)
    else
        return COUNT(L.next)
```

- Costo computazionale: $\Theta(n)$, $n =$ numero di nodi nella lista
- Entrambe le implementazioni devono visitare tutti i nodi nella lista

Esercizio 6

- Scrivere l'algoritmo INSERTIONSORT per liste doppiamente concatenate



Esercizio 6 - Soluzione

```
 1: function INSERTIONSORT(DLList L)
 2:     tmp = L.head
 3:     while tmp ≠ NIL do
 4:         p = tmp
 5:         while p.prev ≠ NIL and  p.key < p.prev.key do
 6:             SWAP(p,p.prev)
 7:             p = p.prev
 8:         tmp = tmp.next
 9:
10: function SWAP(DLList p, DLList q)
11:     if p ≠ NIL and q ≠ NIL then
12:         tmp = p.key, p.key = q.key, q.key = tmp
```

- Il ciclo while a riga 3 viene eseguito n volte
- Caso ottimo (ciclo while a riga 4 mai eseguito): $\Theta(n)$
- Caso pessimo (ciclo while a riga 4 eseguito interamente):

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$$

Esercizio 7

Consideriamo la seguente sequenza di caratteri

E A S * Y * Q U E * * * S T * * * I O * N * * *

1. Partendo da una pila vuota S, un carattere c indica l'operazione *push(S,c)* mentre un asterisco * indica l'operazione *pop(S)*. Mostrare la sequenza di caratteri ritornata dalle operazioni *pop*.

2. Partendo da una coda vuota Q, un carattere c indica l'operazione *enqueue(Q,c)* mentre un asterisco * indica l'operazione *dequeue(Q)*. Mostrare la sequenza di caratteri ritornata dalle operazioni *dequeue*.

Esercizio 7 - Soluzione

1. Sequenza di caratteri ritornata dalle operazioni *pop*

   S Y E U Q T S A O N I

2. Sequenza di caratteri ritornata dalle operazioni *dequeue*

   E A S Y Q U E S T I O N

Esercizio 8

- Consideriamo una pila S che supporta le seguenti operazioni in $O(1)$
  - $\textsc{top}(S)$ ritorna il valore in cima alla pila S (senza rimuoverlo)
  - $\textsc{isEmpty}(S)$ true se S è vuota, falso altrimenti
  - $\textsc{push}(S, x)$ inserisce x in cima alla pila S
  - $\textsc{pop}(S)$ rimuove e ritorna il valore in cima alla pila S
    - Termina in errore se S è vuota (ex., lancia un'eccezione)
- Vogliamo mantenere traccia del valore minimo in una pila S_1
- Usiamo una seconda pila S_2 che ha sempre in cima il valore attualmente minimo in S_1
- Ideare le seguenti due operazioni, con costo $O(1)$
  1. $\textsc{push}2(S_1, S_2, x)$ inserisce x in S_1 e aggiorna (se necessario) la cima di S_2 in modo che contenga il valore minimo in S_1
  2. $\textsc{pop}2(S_1, S_2)$ rimuove e ritorna il valore in cima ad S_1 e aggiorna (se necessario) S_2

Esercizio 8 - Soluzione

```
function PUSH2(STACK S1, STACK S2, INT x)
    PUSH(S1,X)
    if ISEMPTY(S2) or TOP(S2) ≥ x then
        PUSH(S2,X)
```

```
function POP2(STACK S1, STACK S2) → INT
    if not ISEMPTY(S1) and TOP(S1) == TOP(S2) then
        POP(S2)
    return POP(S1)
```

- PUSH2: se il nuovo valore x è più piccolo o uguale al valore in cima ad S_2 allora inserisce x in cima ad S_2
- POP2: tre casi possibili:
  1. il valore top in S_1 è uguale a quello in S_2 $\Rightarrow$ pop su S_2
  2. il valore top in S_1 è più grande di quello in S_2 $\Rightarrow$ nessuna azione
  3. il valore top in S_1 è più piccolo di quello in S_2 $\Rightarrow$ impossibile (il valore in cima ad S_2 è il minimo in S_1)
- Entrambe le funzioni usano solo operazioni $O(1)$ $\Rightarrow$ costo $O(1)$

Esercizio 9

- Scrivere lo pseudocodice di un algoritmo MERGESTACKS(S_1,S_2,S_3) che, presi in input due pile ordinate S_1 and S_2 (elemento minimo in cima), costruisce S_3, inizialmente vuota, con tutti gli elementi in S_1 e S_2 in ordine (dal più grande al più piccolo)

- Usare solo le operazioni della pila ISEMPTY, TOP, PUSH, POP

- Il costo pessimo di MERGESTACKS deve essere $O(n_1 + n_2)$ dove
  - n_1 = numero di elementi in S_1
  - n_2 = numero di elementi in S_2

Esercizio 9 - Soluzione

```
1: function MERGESTACKS(Stack S1, Stack S2, Stack S3)
2:     if not ISEMPTY(S1) or not ISEMPTY(S2) then
3:         if ISEMPTY(S1) or TOP(S1) ≥ TOP(S2) then
4:             x = POP(S2)
5:         else if ISEMPTY(S2) or TOP(S2) > TOP(S1) then
6:             x = POP(S1)
7:         MERGESTACKS(S1,S2,S3)
8:         PUSH(S3,x)
```

- Costo computazionale $\Theta(n_1 + n_2)$
  - La ricorsione termina quando sia S_1 che S_2 sono vuote
  - In ogni chiamata ricorsiva un solo valore viene rimosso o da S_1 (line 4) o da S_2 (line 6)
- Notiamo che la chiamata ricorsiva a riga 7, eseguita prima di PUSH a riga 8, assicura che gli elementi siano inseriti in S_3 partendo dal più grande al più piccolo
  - Invertendo le righe 7 e 8 otteniamo S_3 ordinato in modo inverso

Esercizio 10

- Scrivere un algoritmo che, preso in input un albero binario, ne cancelli tutte le foglie

- L'algoritmo ritorna il puntatore alla radice dell'albero (che potrebbe essere NIL se l'albero consiste di una sola foglia)

Esercizio 10 - Soluzione

```
function removeLeaves(Node T) → Node
    if T == NIL or isLeaf(T) then
        delete(T)
        return NIL
    else
        T.left = removeLeaves(T.left)
        T.right = removeLeaves(T.right)
        return T
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- n = numero di nodi nell'albero

Esercizio 11

- Scrivere un algoritmo che, preso in input un albero binario, ritorni la somma dei valori contenuti nelle foglie
- Se l'albero è vuoto, l'algoritmo ritorna 0
- Risolvere l'esercizio sia con un algoritmo ricorsivo che iterativo

Esercizio 11 - Soluzione ricorsiva

```
function sumleaves(Node T) → int
    if T == NIL then
        return 0
    else if isLeaf(T) then
        return T.val
    else
        return sumleaves(T.left)+sumleaves(T.right)
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- n = numero di nodi nell'albero

Esercizio 11 - Soluzione iterativa

```
function COUNTLEAVES(TREE T) → INT
    n = 0
    LET Q BE A QUEUE
    if T.root ≠ NIL then
        ENQUEUE(Q, T.root)
    while Q.size ≠ 0 do
        x = DEQUEUE(Q)
        if ISLEAF(x) then
            n = n + x.val
        if x.left ≠ NIL then
            ENQUEUE(Q, x.left)
        if x.right ≠ NIL then
            ENQUEUE(Q, x.right)
    return n
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- n = numero di nodi nell'albero

Esercizio 12

- Scrivere un algoritmo che, preso in input un albero binario, elimini tutte le foglie che sono figli sinistri e con contengono lo stesso valore del nodo padre

- L'algoritmo non ritorna nessun valore (la radice non viene mai modificata dato che non ha un padre)

Esercizio 12 - Soluzione

```
function delleaves(Node T)
    if T ≠ NIL then
        if T.left ≠ NIL and isLeaf(T.left) and T.left.val == T.val then
            T.left = NIL
        delleaves(T.left)
        delleaves(T.right)
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- $n =$ numero di nodi nell'albero

Esercizio 13

- Scrivere un algoritmo che calcoli l'altezza di un albero binario
  - Se l'albero contiene solo la radice, l'altezza è 0
  - Se l'albero è vuoto, l'altezza è -1
- Risolvere l'esercizio sia con visita in profondità che in ampiezza

Esercizio 13 - Soluzione in profondità

```
function HEIGHT(NODE T) → INT
    if T == NIL then
        return −1
    else if ISLEAF(T) then
        return 0
    else
        return 1+MAX(HEIGHT(T.left),HEIGHT(T.right))
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- n = numero di nodi nell'albero

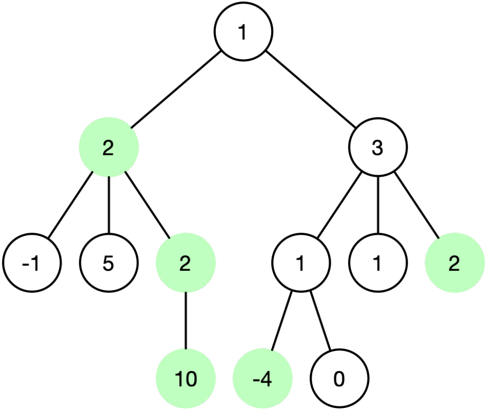
Esercizio 13 - Soluzione in ampiezza

```
function HEIGHT(TREE T) → INT
    n = −1
    if T.root ≠ NIL then
        LET Q BE A NEW QUEUE
        ENQUEUE(Q, [T.root, 0])
        while Q.size ≠ 0 do
            [x, n] = DEQUEUE(Q)
            if x.left ≠ NIL then
                ENQUEUE(Q, [x.left, n + 1])
            if x.right ≠ NIL then
                ENQUEUE(Q, [x.right, n + 1])
    return n                ▷ Last visited node is the deepest one
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- n = numero di nodi nell'albero

Esercizio 14

- Scrivere un algoritmo che conti il numero di nodi con valore pari in un albero generico (non binario)



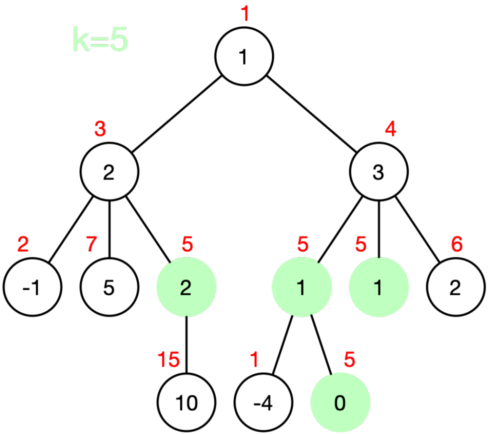
Esercizio 14 - Soluzione

```
function COUNTEVEN(NODE T) → INT
    if T == NIL then
        return 0
    else if T.val mod 2 == 0 then
        return 1+COUNTEVEN(T.first)+COUNTEVEN(T.next)
    else
        return COUNTEVEN(T.first)+COUNTEVEN(T.next)
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- n = numero di nodi nell'albero

Esercizio 15

- Scrivere un algoritmo che, dato un albero generico (non binario) ed un intero *l*, conti il numero di nodi tali per cui la somma dei valori del percorso radice-nodo sia uguale a *k*



Esercizio 15 - Soluzione

```
function countk(Node T, int k) → int
    if T == NIL then
        return 0
    else
        if T.val == k then
            return 1+countk(T.first,k − T.val)+countk(T.next,k)
        else
            return countk(T.first,k − T.val)+countk(T.next,k)
```

- Costo computazionale: $\Theta(n)$ (visita tutti i nodi dell'albero)
- n = numero di nodi nell'albero