Strutture dati elementari - Esercizi

Pietro Di Lena

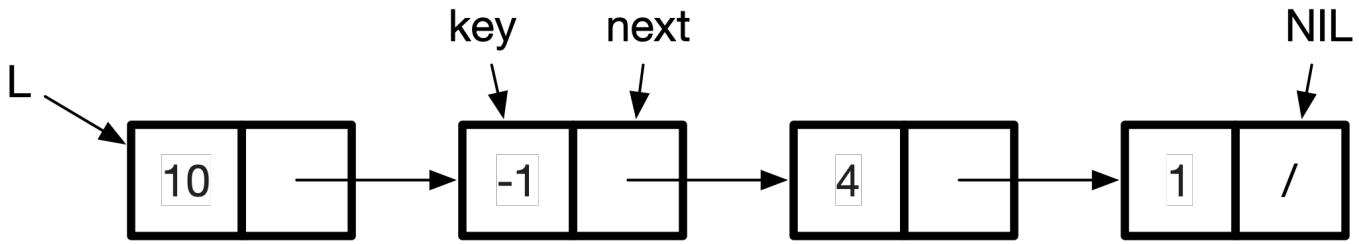
Dipartimento di Informatica – Scienza e Ingegneria
Università di Bologna

Algoritmi e Strutture di Dati
Anno Accademico 2021/2022

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

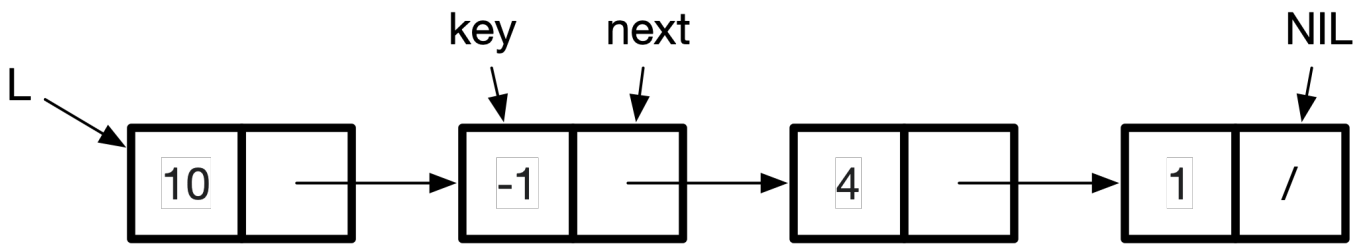
```
1:  function LASTK(List L, Int k) → List
2:      tmp = L                                  ▷ Temporary pointer
3:      n = 0                                    ▷ Number of nodes
4:      while tmp ≠ NIL do                       ▷ Count the number of nodes
5:          n = n + 1
6:          tmp = tmp.next
7:      if n < k then
8:          return NIL
9:      else
10:         tmp = L                              ▷ 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

```
 1: function LastK(List L, Int k) → List
 2:     tmp1 = L                                  ▷ Temporary pointer
 3:     tmp2 = L                                  ▷ Temporary pointer
 4:     i = 0                                     ▷ Counter
 5:     while tmp1 ≠ NIL and i < k do             ▷ Move tmp2 on the k-th node
 6:         i = i + 1
 7:         tmp1 = tmp1.next
 8:     if i < k then                             ▷ There are less than k nodes
 9:         return NIL
10:     else
11:                                               ▷ The distance between tmp1 and tmp2 is exactly k
12:         while tmp1 ≠ NIL do
13:             tmp1 = tmp1.next
14:             tmp2 = tmp2.next
15:         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

```
1: function deleteEven(List L) → List
2:     if L == NIL then
3:         return NIL
4:     else if L.key mod 2 == 0 then                ▷ Even node, remove
5:         return deleteEven(L.next)
6:     else                                         ▷ Odd node, no remove
7:         return L.next = deleteEven(L.next)
8:         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

```
1:  function deleteAndDuplicate(List L, int nPrec) → List
2:      if L == NIL then
3:          return NIL
4:      else if L.key mod 2 == 0 then                ▷ Even node, remove
5:          return deleteAndDuplicate(L.next, nPrec + 1)
6:      else                                         ▷ Odd node, duplicate
7:          L.next = deleteAndDuplicate(L.next, nPrec)
8:          while nPrec > 0 do                       ▷ Duplication with nPrec head insert
9:              tmp = new List(L.key)
10:             tmp.next = L
11:             L = tmp
12:             nPrec = nPrec − 1
13:         return L
```

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

Esercizio 4 - Soluzione 2

```
1:  function deleteAndDuplicate(List L, int nPrec, int nDup) → List
2:      if L == NIL then
3:          return NIL
4:      else if L.key mod 2 == 0 then                    ▷ Even node, remove
5:          return deleteAndDuplicate(L.next, nPrec + 1, nPrec + 1)
6:      else if nPrec > 0 then                           ▷ Odd node, duplicate
7:          tmp = new List(L.key)
8:          tmp.next = deleteAndDuplicate(L, nPrec, nPrec − 1)
9:          return tmp
10:     else                                             ▷ Odd node, no duplicate
11:         L.next = deleteAndDuplicate(L.next, nPrec, nPrec)
12:         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 effettivo: 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, 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 5 - Soluzione

```
1: function removeLeaves(Tree T) → Tree
2:     if T == NIL or isLeaf(T) then
3:         delete(T)
4:         return NIL
5:     else
6:         T.left = removeLeaves(T.left)
7:         T.right = removeLeaves(T.right)
8:         return T
```

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

Esercizio 6

- 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 6 - Soluzione ricorsiva

```
1: function SUMLEAVES(TREE T) → INT
2:     if T == NIL then
3:         return 0
4:     else if ISLEAF(T) then
5:         return T.val
6:     else
7:         return SUMLEAVES(T.left)+SUMLEAVES(T.right)
```

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

Esercizio 6 - Soluzione iterativa

```
1:  function CountLeaves(Tree T) → int
2:      n = 0
3:      Let Q be a Queue
4:      if T ≠ NIL then
5:          enqueue(Q, T)
6:      while Q.size ≠ 0 do
7:          x = dequeue(Q)
8:          if isLeaf(x) then
9:              n = n + x.val
10:         if x.left ≠ NIL then
11:             enqueue(Q, x.left)
12:         if x.right ≠ NIL then
13:             enqueue(Q, x.right)
14:     return n
```

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

Esercizio 7

- 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 7 - Soluzione

```
1: function DELLEAVES(TREE T)
2:     if T ≠ NIL then
3:         if T.left ≠ NIL and ISLEAF(T.left) and T.left.val == T.val then
4:             DELETE(T.left)
5:             T.left =NIL
6:         DELLEAVES(T.left)
7:         DELLEAVES(T.right)
```

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

Esercizio 8

- 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 8 - Soluzione in profondità

```
1: function HEIGHT(TREE T) → INT
2:     if T == NIL then
3:         return −1
4:     else if ISLEAF(T) then
5:         return 0
6:     else
7:         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

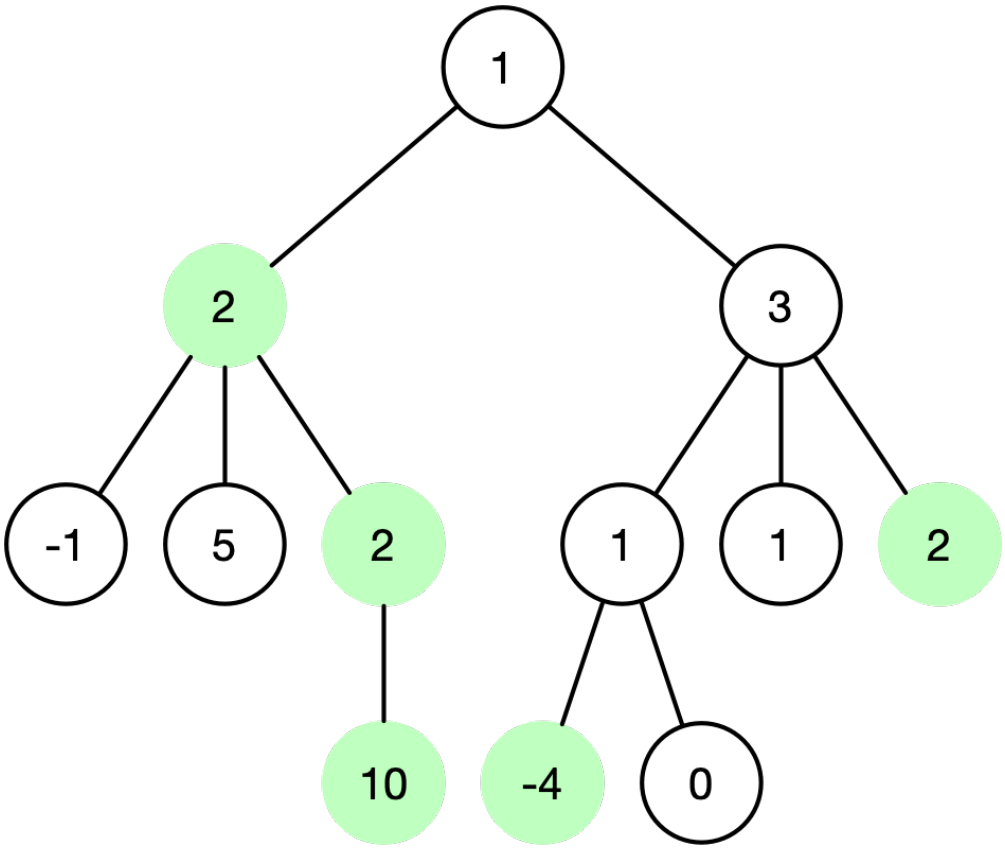
Esercizio 8 - Soluzione in ampiezza

```
 1: function HEIGHT(TREE T) → INT
 2:     n = −1
 3:     if T ≠ NIL then
 4:         LET Q BE A NEW QUEUE
 5:         ENQUEUE(Q, [T, 0])
 6:         while Q.size ≠ 0 do
 7:             [x, n] = DEQUEUE(Q)
 8:             if x.left ≠ NIL then
 9:                 ENQUEUE(Q, [x.left, n + 1])
10:             if x.right ≠ NIL then
11:                 ENQUEUE(Q, [x.right, n + 1])
12:     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 9

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



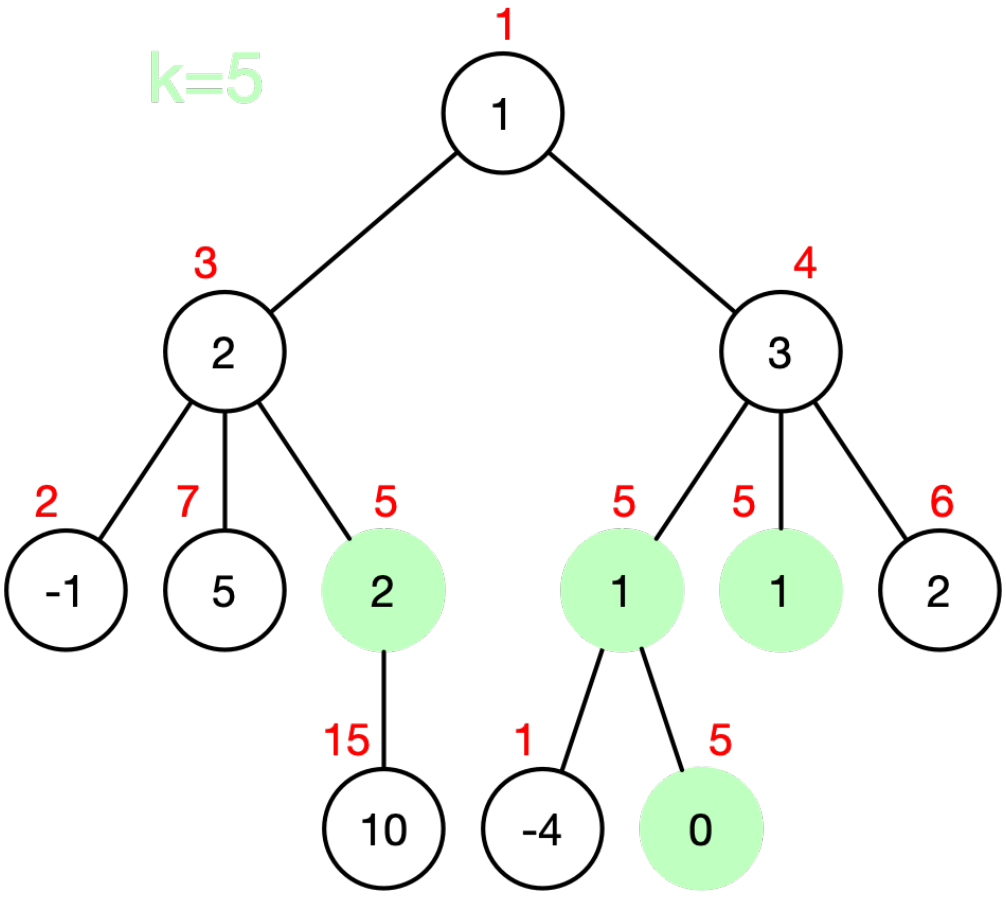
Esercizio 9 - Soluzione

```
1: function CountEven(Tree T) → int
2:     if T == NIL then
3:         return 0
4:     else if T.val mod 2 == 0 then
5:         return 1+CountEven(T.first)+CountEven(T.next)
6:     else
7:         return CountEven(T.first)+CountEven(T.next)
```

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

Esercizio 10

- 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 10 - Soluzione

```
1: function CountK(Tree T, int k) → int
2:     if T == NIL then
3:         return 0
4:     else
5:         if T.val == k then
6:             return 1+CountK(T.first,k − T.val)+CountK(T.next,k)
7:         else
8:             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