Laboratorio di algoritmi e strutture dati

Esercizi su alberi*

Docente: Violetta Lonati

1 Alberi binari - implementazione con puntatori ai figli destro e sinistro

In questo esercizio scriveremo un programma che

1. genera a caso una sequenza di interi (di lunghezza massima fissata con una opportuna macro) e la memorizza in una slice;
2. costruisce un albero binario a partire dalla slice (come descritto in seguito);
3. stampa l'albero nella reppresentazione *a sommario* (come descritta in seguito);
4. stampa i nodi dell'albero negli ordini determinati rispettivamente dalle visite in preordine, inordine e postordine.

Seguite le indicazioni che seguono; osservate che prima vi suggerisco di scrivere alcune funzioni preliminari utili per stampare gli alberi e per fare testing.

1.1 Funzioni preliminari per elaborare alberi binari

Innanzitutto, definite i tipi:

```
type bitreeNode struct {
  left  *bitreeNode
  right *bitreeNode
  val   int
}

type bitree struct {
  root *bitreeNode
}
```

Quindi scrivete (e testate!) le funzioni per le visite in ordine simmetrico, anticipato, posticipato:

*Ultimo aggiornamento: 16 novembre 2022 - 09:13:15

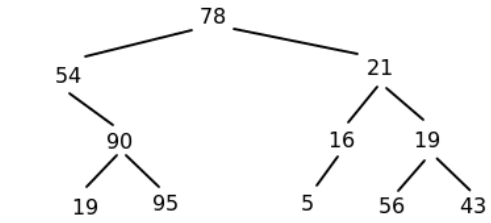
```
inorder(node *bitreeNode)
preorder(node *bitreeNode)
postorde(node *bitreeNode)
```

Per testare le funzioni, costruite degli alberi di test manualmente, ad esempio iniziando così:

```
t := &bitree{nil}
t.root = &bitreeNode{nil, nil, 50}
t.root.left = newNode(20)
t.root.right = newNode(80)
t.root.left.left = newNode(10)
t.root.left.left.right = newNode(15)
...
```

1.2 Stampa di alberi *a sommario*

Per visualizzare alberi binari su standar output, è utile usare la rappresentazione usata nei sommari dei libri, oppure nei browser di file, come nel seguente esempio:




```
*78
  *54
    *
    *90
      *19
      *95
  *21
    *16
      *5
      *
    *19
      *56
      *43
```

Notate che, nel caso in cui un nodo abbia un solo figlio, per poter distinguere tra figlio destro e figlio sinistro, è necessario evidenziare con una riga vuota l'assenza del figlio mancante.

Per realizzare questa funzione, si può usare una funzione di visita dell'albero: qual è quella appropriata? La stampa dei nodi deve essere fatta indentando appropriatamente; il numero di spazi può essere dato come parametro della funzione, che quindi avrà segnatura

```
stampaAlberoASommario(node *bitreeNode, spaces int)
```

e andrà chiamata nel modo seguente:

```
stampaAlberoASommario( root, 0 );
```

La seguente implementazione presenta dei problemi: correggetela!

```
func stampaAlberoASommario(node *bitreeNode, spaces int) {
  for i := 0; i < spaces; i++ {
    fmt.Print("  ")
  }
  fmt.Print("*")

  fmt.Println(node.val)

  if node.left != nil || node.right != nil {
    stampaAlberoASommario(node.right, spaces+1)
    stampaAlberoASommario(node.left, spaces+1)
  }
}
```

1.3 Una stampa alternativa

Vediamo un'altra funzione per la stampa di alberi binari: in questo caso l'albero viene stampato su una sola riga.

1. provate a tracciare la funzione su carta su un paio di alberi di test.
2. Verificate poi la vostra previsione eseguendo effettivamente la funzione.
3. Descrivete a parole il formato con cui la funzione rappresenta un albero binario.

```
func stampaAlbero(node *bitreeNode) {
  if node == nil {
    return
  }

  fmt.Print(node.val, "")
  if node.left == nil && node.right == nil {
    return
  }

  fmt.Print(" [")
  if node.left != nil {
    stampaAlbero(node.left)
  } else {
    fmt.Print("-")
  }
  fmt.Print(", ")
  if node.right != nil {
    stampaAlbero(node.right)
  } else {
    fmt.Print("-")
  }
  fmt.Print("]")
}
```

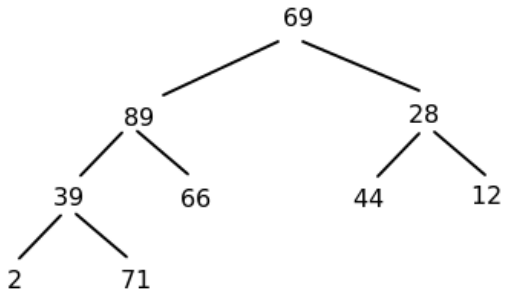
1.4 Dal vettore all'albero

Scriviamo una funzione ricorsiva

```
arr2tree(a []int, i int) (root *bitreeNode)
```

che, data una slice `a` ed un indice `i`, costruisca ricorsivamente l'albero binario, con radice contenente il valore `a[0]`, e tale che valga la seguente proprietà: se un nodo è etichettato con `a[i]`, allora il suo figlio sinistro è etichettato con `a[2*i+1]` e il suo figlio destro è etichettato con `a[2*i+2]`. Questo significa che la slice contiene gli elementi del vettore letti livello per livello, a partire dal livello della radice e scendendo man mano di livello.

Ad esempio, dato `a = {69, 89, 28, 39, 66, 44, 12, 2, 71}`, la funzione deve costruire l'albero



Il primo elemento della slice è la radice, il secondo e il terzo sono al livello 1, gli elementi dal quarto al settimo sono al livello 2, e così via.

Notate che si otterrà sempre un albero binario completo (in cui cioè ogni nodo, tranne le foglie, ha due figli).

Completate opportunamente la funzione qui sotto, poi testatela.

```
func arr2tree(a []int, i int) (root *bitreeNode) {
  if i >= len(a) {
    return nil
  }

  root = ...
  root.left = ...
  root.right = ...
  return root
}
```

2 Alberi - implementazione con puntatori al primo figlio e al prossimo fratello

Rivedete l'implementazione degli alberi binari svolta negli esercizi precedenti, e valutate cosa può essere adattato facilmente al caso di alberi generici, implementati con un puntatore al primo

figlio e un puntatore al prossimo fratello. Ci sono funzioni che presentano delle criticità? A cosa sono dovute?

3 Alberi - implementazione con tabella dei padri

Se non ci interessa scendere dalla radice verso le foglie, ma importa di più la relazione da figlio a padre, si può implementare l'albero tramite una "tabella dei padri": per ciascun elemento, si memorizza il padre. Questa implementazione è molto semplice da realizzare, ma non sempre è sufficiente: dipende dalla situazione che vogliamo modellare e dal problema che stiamo affrontando.

Qual è il modo più appropriato di realizzare una ""tabella dei padri in Go?

3.1 Mappa delle orbite

La sfida di Advent of Code, 2019, day 6, intitolata *Universal Orbit Map*, può essere affrontata modellando la situazione con un albero, e implementando l'albero con una tabella dei padri. La prima parte della sfida è riassunta qui sotto. Chi lo preferisce, può fare riferimento alla versione originale inglese, disponibile a questo indirizzo: `https://adventofcode.com/2019/day/6`

Per entrambe le parti della sfida, prima di iniziare a scrivere codice, rispondete a queste domande:

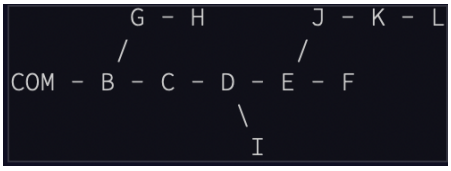
1. Modellate la situazione con un albero: cosa rappresentano i nodi dell'albero? cosa rappresenta la relazione padre/figlio?
2. Riformulate i problemi usando la terminologia degli alberi:
   a) Parte 1 - Cosa sono le orbite dirette? E le orbite indirette? Come può essere descritto, in termini di alberi, il numero di orbite indirette di un oggetto?
   b) Parte 2 - Come può essere descritta, in termini di alberi, La distanza tra gli oggetti attorno a cui orbitano YOU e SAN?
3. Progettate una soluzione per calcolare:
   a) Parte 1 - il numero di orbite indirette
   b) Parte 2 - il numero di trasferimenti di orbita necessari

   Ragionare in termini di alberi vi aiuterà a impostare gli algoritmi risolutivi!

Descrizione del problema

Sei venuto in possesso della Mappa delle Orbite dell'Universo. La Mappa mostra che ogni oggetto dello spazio (tranne il COM - *Center of Mass*) ruota attorno a *esattamente* un altro oggetto. Questa relazione è indicata in questa forma: `AAA)BBB` significa che `BBB` orbita attorno ad `AAA`.

Per esempio, data questa situazione



è rappresentata nella mappa così:

```
COM)B
B)C
C)D
D)E
E)F
B)G
G)H
D)I
E)J
J)K
K)L
```

Parte 1. Se A orbita attorno a B e B orbita attorno a C, si dice che A orbita indirettamente attorno a C. Devi calcolare il numero di tutte le orbite dirette e indirette presenti nella mappa.

Ad esempio:

- D orbita direttamente attorno a C e indirettamente attorno a COM (3 orbite in totale)
- L orbita direttamente attorno a K e indirettamente attorno a J, E, D, C, B e COM (7 orbite in totale)
- COM non orbita attorno a nulla

Il numero totale di orbite dirette e indirette è 42.

Testate la vostra soluzione con l'input che trovate a questo URL: `https://adventofcode.com/2019/day/6/input`

Parte 2. Potete trovare la seconda parte della sfida sul sito di Advent of Code, una volta completata con successo la prima!