Liste concatenate

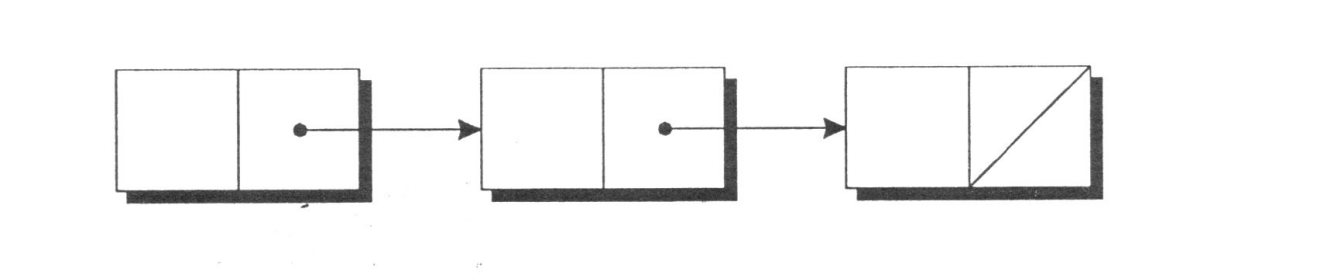
Violetta Lonati

Università degli studi di Milano
Dipartimento di Informatica

Laboratorio di algoritmi e strutture dati
Corso di laurea in Informatica

Liste concatenate

Una lista concatenata consiste di una catena di strutture chiamate nodi. Ogni nodo contiene un puntatore al prossimo nodo della catena. L'ultimo nodo contiene il puntatore nullo (rappresentato da una diagonale).



Struttura nodo

```
type listNode struct {
  item int
  next *listNode
}
```

Bisogna tenere traccia di dove comincia la lista. Possiamo usare un puntatore a `listNode`.

```
var list *listNode
```

Soluzione migliore: definiamo una struttura `linkedList` con un campo di tipo puntatore a `listNode` che punta al nodo di testa.

```
type linkedList struct {
  head *listNode
}
```

Creazione di un nuovo nodo

Serve:

1. allocare memoria per il nuovo nodo;
2. memorizzare il dato nel nuovo nodo;
3. inserire il nodo nella lista.

Allocazione di un nuovo nodo

Allocazione e inizializzazione di un nuovo nodo:

```
var n listNode
n.item = 20
```

Problematico: se riutilizzo `n` per inserire nuovi nodi in una lista, si sovrascrive la struttura che fa da nodo iniziale!

```
var list linkedList
list.head = &n

n.item = 30
n.next = list.head
list.head = &n
```

Invece ogni volta bisogna creare un nuovo nodo (allocare nuovo spazio)!

Creazione di un nuovo nodo

Usiamo una variabile puntatore:

```
var node *listNode
node = new(listNode)
(*node).item = 10
```

In Go, non è necessario dereferenziare i puntatori a struttura, quindi si può scrivere:

```
node = new(listNode)
node.item = 10
```

Ancora più breve:

```
node = &listNode{10, nil}
```

Funzione per creare un nuovo nodo

Funzione che restituisce l'indirizzo di un nuovo nodo inizializzato con il valore passato per argomento:

```
func newNode(val int) *listNode {
  return &listNode{val, nil}
}
```

Variante più estesa

```
func newNode(val int) *listNode {
  node := new(listNode)
  node.item = val
  return node
}
```

NB: `node.item` ha valore `nil` per default.

Inserimento del nodo in testa alla lista

```
var list linkedList
node := newNode(30)
node.next = l.head
list.head = node
```

Inseriamo in una funzione:

```
func addNewNode(l linkedList, val int) {
  node := newNode(val)
  node.next = l.head
  l.head = node
}
```

Problema: si modifica il campo `head` della struttura `l` locale alla funzione, non quello della struttura passata per argomento

Inserimento del nodo in testa alla lista - funzione

Prima soluzione - restituiamo la lista modificata; la funzione chiamante deve fare un assegnamento:

```
func addNewNode(l linkedList, val int) linkedList {
  node := newNode(val)
  node.next = l.head
  l.head = node
  return l
}

func funzioneChiamante( ... ) {
  ...
  list = addNewNode(list, 42)
  ...
}
```

Nota: l'assegnamento `l.head = node` modifica il campo `head` della struttura `l`, non della struttura `list` della funzione chiamante.

Inserimento del nodo nella lista - funzione

Seconda soluzione - usiamo un puntatore e passiamo alla funzione l'indirizzo della struttura

textttlist:

```
func addNewNodePointer(l *linkedList, val int) {
  node := newNode(val)
  node.next = l.head
  l.head = node
}

func funzioneChiamante( ... ) {
  ...
  addNewNodePointer(&list, 42)
  ...
}
```

Nota: l'assegnamento `l.head = node` equivale in questo caso a `(*l).head`, quindi si modifica direttamente la struttura puntata, ovvero la struttura `list` della funzione chiamante.

Stampa di una lista

Scorriamo la lista a partire dalla testa usando un puntatore che ad ogni passo punta al nodo che stiamo visitando.

```
func printList(l linkedList) {
  p := l.head
  for p != nil {
    fmt.Print(p.item, " ")
    p = p.next
  }
  fmt.Println()
```

Ricerca di un nodo

Scorriamo la lista a partire dalla testa cercando il valore desiderato all'interno dei nodi. Usiamo un puntatore che ad ogni passo punta al nodo che stiamo visitando.
Scriviamo una funzione che effettua la ricerca di un elemento in una lista: `list` sia il puntatore al primo nodo della lista e `n` il valore da cercare.

```
func searchList(l linkedList, val int) (bool, *listNode)
  p := l.head
  for p != nil {
    if p.item == val {
      return true, p
    }
    p = p.next
  }
  return false, nil
}
```

Cancellazione di un nodo

Serve:

1. trovare il nodo da eliminare;
2. modificare il nodo precedente in modo che punti al nodo successivo a quello da cancellare.

Non mi basta usare un solo puntatore come per la ricerca, perché una volta trovato, non sappiamo più dov'era il suo predecessore!!

Cancellazione di un nodo

Usiamo due puntatori: `cur` punta al nodo corrente; `prev` punta al suo predecessore.

```
var curr, prev *listNode = l.head, nil
for curr != nil {
  if curr.item == val {
    ....
    return true
  }
  prev = curr
  curr = curr.next
}
```

Cancellazione di un nodo

Serve:

1. Trovare il nodo da eliminare;
2. modificare il nodo precedente in modo che punti al nodo successivo a quello da cancellare;

Per il secondo punto basta l'assegnamento:

```
l.head = l.head.next
```

Bisogna però tenere conto del caso in cui `val` si trova in testa (quindi `prev` è ancora nullo):

```
if curr.item == val {
  if prev == nil {
    l.head = l.head.next
  } else {
    prev.next = curr.next
}
```

Cancellazione di un nodo

```
func deleteItem(l *linkedList, val int) bool {
  var curr, prev *listNode = l.head, nil
  for curr != nil {
    if curr.item == val {
      if prev == nil {
        l.head = l.head.next
      } else {
        prev.next = curr.next
      }
      return true
    }
    prev = curr
    curr = curr.next
  }

  return false
}
```

Liste ordinate

Anziché effettuare l'inserimento in testa, bisogna prima trovare il punto giusto in cui inserire il nuovo nodo. Scorro la lista usando di nuovo due puntatori `cur` e `prev`, fermandomi quando `cur.item` diventa maggiore del valore cercato: questo significa che il nodo va inserito fra `prev` e `cur`.

Liste doppiamente concatenate

Si deve prevedere anche un puntatore al nodo precedente nella struttura `listNode` e un puntatore alle coda nella struttura `doublyLinkedList`,

```
type biListNode struct {
  item int
  next *listNode
  prev *listNode
}

type biLinkedList struct {
  head *listNode
  tail *listNode
}
```