Algoritmi e Strutture Dati **Anno Accademico 2022–2023**

Esercizi: Liste

Manuela Montangero

1 Qualche considerazione - DA LEGGERE

NON SALTATE QUESTA SEZIONE

In tutti gli esercizi sulle liste che seguono (a meno che non sia esplicitamente dichiarato diversamente) assumeremo che le liste siano implementate come descritto a lezione. Ovvero, ogni nodo contiene i seguenti campi:

- Un campo val per contenere un valore,
- Un campo $next$ contenente il puntatore al nodo successivo. Il puntatore è NIL se il nodo è l'ultimo della lista,

e la lista è un puntatore al primo nodo. **NON** assumeremo che la lista sia a doppia concatenazione (ovvero con un puntatore $prev$ al nodo precedente).

Per creare un nuovo nodo della lista usaremo la funzione `new_list_node()` che restituisce il puntatore al nodo creato.

Le stesse assunzioni valgono per le prove d'esame.

Sulla natura ricorsiva delle liste. Osserviamo che le liste hanno una natura ricorsiva. Infatti, posso pensare ad una lista L come ad una struttura formata da due parti: (1) il primo nodo (spesso chiamata $head$) e (2) la lista che inizia dal secondo nodo (spesso chiamata $tail$, attenzione a non confondersi con la $tail$ di una coda, che è invece l'ultima posizione della coda). Per identificare il primo nodo basta considerare il puntatore alla lista L, per identificare la tail della lista basta considerare il puntatore nel campo `next` del primo nodo, quindi $L.next$. Osserviamo che quest'ultimo potrebbe essere NIL quando L è composta da un solo nodo, mentre non esiste se L è vuota.

Quindi, una lista L può essere vista come un nodo seguito da una lista che ha dimensione di un nodo più piccola rispetto alla lista L. E questo può essere sfruttato per progettare algoritmi ricorsivi sulle liste. Ovviamente, è necessario anche individuare il caso base della ricorsione. Nella maggior parte dei casi questo è la lista vuota, ma, a seconda del problema, si può pensare di prendere in considerazione anche il caso di una lista formata da un solo nodo (per cui $L.next = NIL$).

Quando abbiamo a che fare con algoritmi ricorsivi, prendete la buona abitudine di scrivere esplicitamente a parole:

il Caso Base, E cosa fare in questo caso

il Caso Ricorsivo, E cosa fare in questo caso

la chiamata principale, specificando il valore dei parametri per risolvere il problema dato

Primitive delle liste. A lezione non siamo entrati nel dettaglio dell'implementazione delle primitive delle liste, ma questo non vuol dire che non dobbiate conoscere le primitive (e sapere cosa fanno) o non possiate usarle nella soluzione

degli esercizi. Quello che non potete fare (né negli esercizi, né nel compito) è inventarvi primitive che non abbiamo elencato a lezione senza darne un'implementazione (e darne un'implementazione equivale a scrivere una funzione/procedura di appoggio per il vostro algoritmo, cosa legittima), così come, se usate le primitive indicate a lezione, non potete cambiare i parametri di input con qualcosa di diverso che vi sia più utile per la vostra soluzione (di nuovo, nel caso ne abbiate bisogno, dovete dare anche l'implementazione alternativa con i parametri stabiliti da voi).

Le primitive che potete usare (anche nel compito) sono quelle che trovate sui lucidi relativi alle liste.

Inoltre, per quello che riguarda questo corso, non ci occupiamo di garbage collection o di deallocazione della memoria (che sono aspetti che dipendono fortemente dal linguaggio di programmazione e dal sistema operativo). Per semplicità assumiamo che la memoria venga deallocata solo una volta terminata l'esecuzione del programma (e quindi è un problema che non riguarda i nostri algoritmi). Quindi, per esempio, quando si elimina un nodo da una lista, il nodo viene solo "scollegato" dagli altri nodi che sono nella lista ma rimane in memoria e, per questo motivo, un puntatore a quel nodo ci permette ancora di accedere alle informazioni contenute nei suoi campi.

2 Esercizi

Esercizio 1 *Scrivere una implementazione delle primitive delle liste che abbiamo elencato a lezione (senza sbirciare le slide!).*

Esercizio 2 *Data una lista L che memorizza interi, si scriva una procedura che stampi tutti i valori che sono memorizzati nella lista, nell'ordine in cui compaiono nella lista. Si dia una versione iterativa e una ricorsiva della procedura.*

Esercizio 3 *Data una lista L che memorizza interi, si scriva una procedura ricorsiva che stampi tutti i valori che sono memorizzati nella lista, nell'ordine* **inverso** *in cui compaiono nella lista.*

Prendere ad esempio una lista con 7 nodi e mostrare l'esecuzione della procedura, facendo vedere le chiamate ricorsive e il momento della stampa.

Suggerimento: *modificare l'esercizio 2 spostando la chiamata ricorsiva nella posizione più opportuna.*

Esercizio 4 *Data una lista L che memorizza numeri naturali, si scriva una funzione che restituisca una* **nuova lista** *che contiene solo i valori pari presenti in L.*

Esercizio 5 *Data una lista L che memorizza numeri naturali, si scriva una procedura che* **modifichi** *L di modo che, alla fine, questa contenga solo numeri dispari.*

Osservazione: *a differenza dell'esercizio 4, in questo caso si chiede di modificare la lista data in input, non di crearne una nuova.*

Esercizio 6 *Data una lista L con n nodi che memorizza interi e un naturale $1 \leq k \leq n$, si scriva una procedura che stampi i valori della lista nel modo seguente: stampa il primo della lista, successivamente quello che si trova dopo k posizioni, successivamente quello che si trova dopo altre k, e così via... Nel caso in cui si arrivi alla fine della lista, si ricomincia a contare da capo. Ogni volta che un valore viene stampato, il nodo corrispondere viene eliminato dalla lista. Ci si ferma quando la lista è vuota.*

Per esempio, se la lista contiene i seguenti valori $L =< 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 >$ e $k = 3$, vengono stampati i seguenti valori, nel seguente ordine: $< 1, 4, 7, 10, 13, 5, 9, 2, 8, 3, 12, 6, 11 >$.

Esercizio 7 *Date due liste L e M che memorizzano dei naturali ordinati in ordine non decrescente, si scriva una funzione (analoga a* `Merge` *di* `MergeSort`*) che restituisce una nuova lista contenente tutti gli elementi di L e di M ordinati in ordine non decrescente.*