Complementi di Programmazione

Esercitazione 12

- Si considerino alberi binari con valori di tipo intero, **positivi**.
- Scrivere dei test nel main per verificare che le funzioni scritte siano corrette.
- Per tutti gli esercizi che richiedono la modifica dell'albero dato in input, svolgere anche le varianti funzionali, in cui viene ritornata la versione modificata dell'albero senza modificare l'albero in input.
- Svolgere gli esercizi sia accedendo solo tramite le funzione del tipo astratto (ove possibile, vedere lista.h, albero.h), sia accedendo alla rappresentazione interna.

Esercizio 12.1

Implementare le funzioni

```
void rimuovi_minimo(TipoAlbero *a);
```

che, dato un albero binario, lo modifichi rimuovendo il nodo (e i rispettivi sottoalberi) che contiene il valore minimo tra tutti i nodi dell'albero originale. Se l'albero è vuoto, non va modificato.

Versione funzionale: `TipoAlbero rimuovi_minimo(TipoAlbero a);`

Esercizio 12.2

Implementare la funzione:

```
void duplica_foglie(TipoAlbero *a);
```

che, dato un albero binario, aggiunga ad ogni nodo foglia `f` un sottoalbero sinistro con un singolo nodo di valore uguale al valore di `f`.

Versione funzionale: `TipoAlbero duplica_foglie(TipoAlbero a);`

Esercizio 12.3

Implementare la funzione:

```
void rimuovi_foglie_profonde(TipoAlbero *a, int n);
```

che, dato un albero binario, rimuova tutti i suoi nodi foglia che hanno almeno `n` antenati.

Versione funzionale:
`TipoAlbero rimuovi_foglie_profonde(TipoAlbero a, int n);`

Esercizio 12.4

Implementare la funzione:

```
void scambia_sottoalberi_simili(TipoAlbero *a);
```

che, dato un albero binario, lo modifica scambiando i sottoalberi di ogni nodo che ha entrambi i sottoalberi, ma solo se i due sottoalberi contengono lo stesso numero di nodi.

Versione funzionale:
`TipoAlbero scambia_sottoalberi_simili(TipoAlbero a);`

Esercizio 12.5

Implementare la funzione

```
void limita_livello(TipoAlbero *a, int livello);
```

che, dati un albero binario `a` e un intero `livello`, rimuova tutti i nodi (e i relativi sottoalberi) che si trovano a un livello maggiore (strettamente) di `livello`. Il nodo radice è da considerarsi al livello 0. Se l'albero è vuoto o non contiene nodi a livelli maggiori di `livello`, non va modificato.

Versione funzionale:
`TipoAlbero limita_livello(TipoAlbero a, int livello);`

Esercizio 12.6

Implementare la funzione:

```
void rimuovi_max_livello(TipoAlbero *a, int livello);
```

che, dato un albero binario e un livello, rimuove il nodo (e il relativo sottoalbero) che ha valore massimo tra i nodi di quel livello.

Versione funzionale:
`TipoAlbero rimuovi_max_livello(TipoAlbero a, int livello);`

Esercizio 12.7

Implementare la funzione:

```
void comprimi_sottoalberi(TipoAlbero *a);
```

che, dato un albero binario, ne modifica la struttura per ogni nodo che ha esattamente un sottoalbero, rimuovendo il sottoalbero stesso. Inoltre, al valore del nodo da cui è stato rimosso il sottoalbero va assegnato il valore della somma dei valori dei nodi contenuti nel sottoalbero rimosso.

Versione funzionale: `TipoAlbero comprimi_sottoalberi(TipoAlbero a);`

Esercizio 12.8

Implementare la funzione:

```
void sx_pari_dx_dispari(TipoAlbero *a);
```

Che, dato un albero binario, modifichi la struttura per ogni nodo `n` nel modo seguente:

- se il valore di `n` è dispari:
    - rimuovere il suo sottoalbero destro, se esiste
    - se `n` non ha un sottoalbero sinistro, aggiungere un sottoalbero sinistro composto da un singolo nodo con lo stesso valore di `n`
- se il valore di `n` è pari:
    - rimuovere il suo sottoalbero sinistro, se esiste
    - se `n` non ha un sottoalbero destro, aggiungere un sottoalbero sinistro composto da un singolo nodo con lo stesso valore di `n`

Versione funzionale: `TipoAlbero sx_pari_dx_dispari(TipoAlbero a);`

Esercizio 12.9

Implementare la seguente funzione

```
TipoAlbero albero_percorso_lungo(TipoAlbero *a);
```

Che, dato un albero binario, ne identifichi il percorso più lungo e ritorni un nuovo albero che contenga tutti i nodi contenuti nel percorso più lungo e i loro nodi figli (ma non il resto dei relativi sottoalberi). Ogni nodo del nuovo albero deve essere allo stesso livello al quale era il nodo corrispondente nell'albero originale.

Esercizio 12.10

Implementare la funzione:

```
void scambia_uguali_livello(TipoAlbero *a, int livello);
```

che, dato un albero binario, scambi ogni nodo `n` (e i relativi sottoalberi) che si trova al livello `livello` con un nodo che 1) sia allo stesso livello, 2) abbia lo stesso valore e 3) sia più a destra del nodo `n`. Se vi è più di un nodo che rispetta tutte e 3 le condizioni, va scelto quello più a sinistra. Se non vi è nessun nodo che rispetta le 3 condizioni, il nodo non va scambiato.

Versione funzionale:
`TipoAlbero scambia_uguali_livello(TipoAlbero a, int livello);`

Esercizio 12.11

Implementare la seguente funzione:

```
TipoAlbero albero_ricerca(TipoLista l);
```

che, data una lista di valori ordinata in modo decrescente, restituisca un nuovo albero che sia l'albero di ricerca corrispondente a `l`.

Esercizio 12.12

Implementare la seguente funzione

```
void binric_rimuovi(TipoAlbero *a, int v);
```

che, dato un albero binario di ricerca `a` e un valore `v`, modifichi l'albero in maniera che venga rimosso il nodo (ma non i suoi sottoalberi) di valore `v`, e l'albero sia ancora un albero binario di ricerca.

Versione funzionale: `TipoAlbero binric_rimuovi(TipoAlbero a, int v);`