Laboratorio di Fondamenti di Informatica LAB_2015.12.22

Anno accademico 2015/2016

1. Bilanciamento di simboli

Si scriva un programma che usa una pila (come spiegata a lezione, che dovrà contiene "caratteri", non esattamente "interi") per il controllo del bilanciamento delle parentesi (si considerino le coppie ( ) [ ] { } < >) in una stringa.

Si può partire dal programma "BilanciamentoSimboli.c" pubblicato inseme a questa specifica, al quale manca solo una funzione (che però fa quasi tutto il lavoro ☺)

- Per ogni simbolo di "apertura" si impili un segnaposto, che sarà rimosso quando si incontra il simbolo duale di "chiusura"
  - Se il simbolo non corrisponde al segnaposto sulla pila, allora la sequenza non è bilanciata
  - se avanzano simboli (nella sequenza letta o sulla pila) la sequenza non è bilanciata
- Esempi: {[][{}[](){}]()}{[()]()}[{}] →ok {[}{] ()} → ko

2.a I miei amici alberi

Scriviamo semplici funzioni per familiarizzare con gli amici alberi.

Si usino, per fare il test delle funzioni, i quattro alberi già contenuti nel programma "Alberi.c", rappresentati nella slide successiva.

```
void print( tree t )
```

stampa tutti i valori contenuti nell'albero

```
void printleaves( tree t )
```

stampa tutti i valori contenuti nelle foglie

```
int contafoglie( tree t )
```

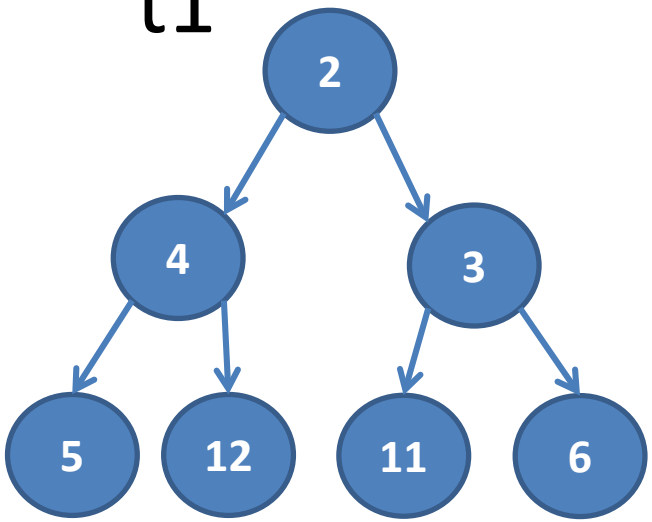
restituisce il numero di foglie dell'albero

```
int containterni( tree t )
```

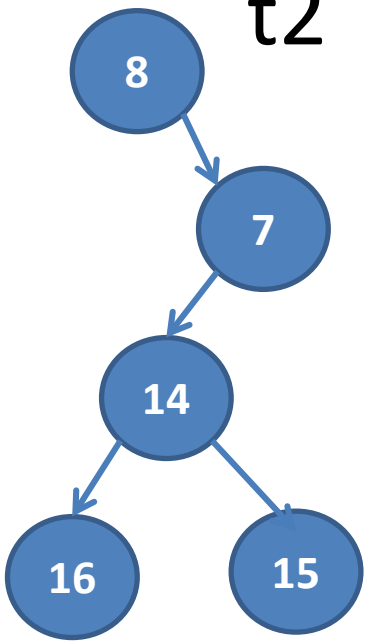
restituisce il numero di nodi interni dell'albero

Quattro alberelli

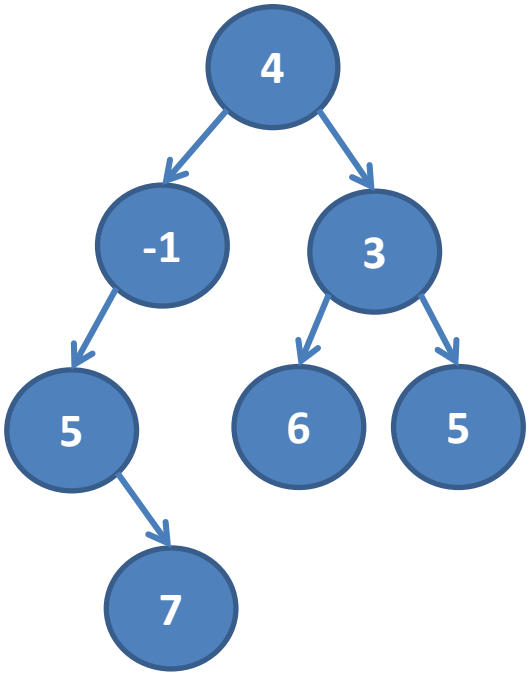
t1



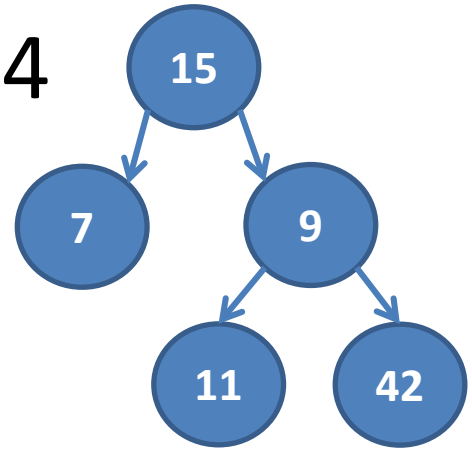
t2



t3



t4



2.b Litighiamo con gli alberi

Ma sono proprio nostri amici, gli alberi?

```
int tuttipositivi( tree t )
```
restituisce 1 se tutti i valori dell'albero sono positivi, 0 altrimenti

```
float average( tree t )
```
restituisce la media dei valori contenuti nell'albero

```
int pienoebilanciato( tree t )
```
restituisce 1 se l'albero è pieno e bilanciato, 0 altrimenti (solo t1 lo è)

```
void stampalivello( tree t, int liv )
```
stampa solo i valori contenuti a profondità liv

2.c Datemi una motosega

Si codifichino le funzioni:

```
int tuttidiversi( tree t )
```

che restituisce 1 se i valori nell'albero sono tutti diversi tra loro, 0 se vi sono duplicati

```
int isobato( tree t )
```

che restituisce 1 se l'albero gode della seguente proprietà: tutti i cammini dalla radice alle foglie hanno la stessa lunghezza (t1 e t2 sono isobati, t3 e t4 no)

```
void stampacomesideve( tree t )
```

che stampa a video un albero…come si deve!

…così si vede bene quali nodi sono figli di quali nodi…

... ma come si fa?

3. Per fare un albero (ordinato) ci vuole…

Si scriva un semplice programma che chiede di inserire (o legge da un vettore opportunamente inizializzato) una sequenza di interi positivi, i quali vengono progressivamente inseriti, tramite una funzione

… `ins_ord(…)` ,

in un albero binario inizialmente vuoto che rimane sempre ordinato. In caso di valori duplicati, si aggiunga sempre un nuovo nodo per il nuovo valore aggiunto.

Si badi che la forma dell'albero risultante dipende dall'ordine di inserimento dei valori: ad esempio inserendo nell'ordine 8 4 12 2 6 10 14 1 3 5 7 9 11 13 15 si ottiene un albero pieno e bilanciato di profondità 4 che contiene i primi 15 numeri naturali, mentre inserendoli in ordine crescente si ottiene un albero di profondità 15, costituito da un solo lunghissimo ramo.

Si codifichino/riciclino poi le funzioni `print()` e `printleaves()` per stampare i valori dell'intero albero e delle sole foglie (in due versioni, una «standard» per stamparli in ordine crescente, una per stamparli in ordine descrescente).

4. We can work it out… with a little help from my friends

Riordiniamo il canzoniere! Si codifichino opportuni programmi e funzioni per:

- inizializzare correttamente una versione "in memoria" dell'indice dei brani del canzniere, secondo la struttura dati (definita nel frammento di codice distribuito):

```
#define MAX_BRANI 200
#define MAX_RIGA  100
typedef struct { char titolo[MAX_RIGA];
        int linea_inizio, lunghezza_testo_in_righe; } Brano ;
typedef struct { char filename[50];  int num_brani;
        Brano brani[MAX_BRANI]; } Canzoniere;
```

- generare un file "canzoniere2.txt" che contiene le stesse canzoni, ma in ordine alfabetico di titolo
  - Può essere utile partire da una versione "in memoria" dell'indice, che conserva i riferimenti alla posizione delle canzoni in un file, e riordinare tale struttura dati prima di procedere alla costruzione del nuovo file ordinato alfabeticamente
  - Può essere utile "riavvolgere" il file con la funzione **`void rewind( FILE * fp )`**
  - Può risultare comodo estendere la struttura dati "Brano" con un campo **`int byte_inizio;`** che indica a quale byte del file inizia ogni brano, in modo da poter posizionare la testina all'inizio dell'i-esimo brano con l'istruzione fseek(fpin, ***c.brani[i]***.byte_inizio, SEEK_SET) [dove ***c*** indica il canzoniere]

5. L'appetito vien mangiando (…la frutta) Alberi n-ari

Un albero *binario* si chiama ***bi***nario perché ogni nodo ha al più *2* sottoalberi.

In un albero n-ario, ogni nodo ha, invece, un numero arbitrario di sottoalberi (a priori illimitato!!)

In ogni nodo dell'albero, quindi, per non sprecare memoria (nei nodi che hanno pochi sottoalberi), si possono rappresentare i sottoalberi tramite… una ***lista*** (semplicemente concatenata) di… alberi n-ari!

Ad esempio la struttura dati potrebbe essere dichiarata come segue, dove la lista di sottoalberi di ogni nodo è realizzata tramite la struttura **Ramo**. (ogni nodo, cioè, ha uan lista di rami, e da ogni ramo «spunta» un sottoalbero).

```
typedef   struct  N  {  int  dato;
                        struct  R  * rami;  }  Nodo;

typedef Nodo * Albero;

typedef   struct  R  {  Albero  sottoalbero;
                        struct R * next; }  Ramo;
```

Si codifichino per gli alberi n-ari le funzioni contanodi(…) , profondità(…), verificapresenza(…) , e… tutte quelle che volete!