Esercitazioni di Tecniche di Programmazione

Due avvertenze:

1. Le soluzioni agli esercizi, le versioni di programmi dati nel testo delle esercitazioni e quant'altro sono raggiungibili tramite la pagina web del corso. Nel titolo di ogni sezione è specificato tra parentesi il nome del (o dei) file in cui è proposta una soluzione (se disponibile ...).
2. I programmi che scriveremo dovranno essere in accordo con la definizione standard **ANSI C** del linguaggio C; perciò, prima di cominciare vogliamo assicurarci che l'ambiente di programmazione che usiamo "usi" anche lui la medesima definizione.
   2.1. SE si usa il Dev C++, nella versione 4.9.9.2 (lingua inglese) bisogna andare nel menu' "Tools", selezionare "Compiler Options", scegliere "Settings" e poi "C Compiler" (selezionare almeno "Support all ANSI Standard C Prorams")
   2.2. Se si usa il vecchio ed eroico Turbo C++, per essere sicuri di star usando ANSI C, bisogna assegnare opportunamente una certa opzione: aprire il menù OPTIONS, selezionare *Compiler* e poi *Source*. Nella finestra di scelta che appare, selezionate ANSI C.

8. Ottava esercitazione autoguidata: alberi binari

8.1. Costruzione di un albero (directory ALBERI1 e ALBERI1_H)

Scrivere un programma che

- riceve il nome di un file contenente la rappresentazione parentetica di un albero binario di interi;
- ne costruisce l'albero binario in memoria centrale, mediante struct e puntatori;
- stampa l'albero, ottenendo sul video la medesima rappresentazione parentetica di partenza;
- dealloca poi l'albero.

(Nei file `A1.TXT`, `A2.TXT` , `A3.TXT` ci sono le rappresentazioni parentetiche di tre alberi. Provare con queste il programma).

L'esercizio puo' essere svolto secondo due approcci: si puo' scegliere di costruire completamente il programma in un unico file; oppure si puo' decidere di costruire file header in cui siano definite/dichiarate le strutture dati e le funzioni usate nella main, e includere questi file in quello con il programma principale.

I suggerimenti seguenti sono stati sviluppati seguendo il primo, ma sono pienamente leggibili e utili in tutti e due gli approcci.

Il primo suggerimento riguarda l'essenza della forma parentetica, nel caso sia non perfettamente ricordata …

Una proposta di soluzione, nel primo approccio, e' data nella directory `ALBERI1` (file `alberi1.c`).
Una proposta di soluzione sviluppata con il secondo approccio e' data nella directory `ALBERI1_H`.
In entrambe le directory (ci sono opportuni file **`README.TXT`** da leggere)

Suggerimento 1: (forma parentetica)
ecco la forma parentetica per l'albero che ha radice 12 e sottoalberi sin e des vuoti:
`( 12()())`

ecco invece la forma parentetica per l'albero che ha
- radice 12
- sottoalbero sin con radice 24 e sottoalberi sin e des vuoti
- sottoalbero des con radice 36 e sottoalberi sin e des vuoti

`( 12( 24()())( 36()()))`

ecco invece la forma parentetica per l'albero che ha
- radice 12
- sottoalbero sin con radice 36 e sottoalberi sin e des vuoti
- sottoalbero des con
    - radice 24
    - sottoalbero sin avente
        - radice 48
        - sottoalbero sin avente
            - radice 64 e sottoalberi sin e des vuoti
        - sottoalbero des vuoto
    - sottoalbero des vuoto

`( 12( 36()())( 24( 48( 64()())()())()))`

---------- STOP (segue sugg. 2) ----------

Suggerimento 2:
```
/* definizione dei tipi di dato */

typedef int TipoElemAlbero;          /* albero di interi */

struct StructAlbero {
  TipoElemAlbero info;
  struct StructAlbero *des,
                      *sin;
};
typedef struct StructAlbero TipoNodoAlbero;
typedef TipoNodoAlbero * TipoAlbero;
```

******* Il prossimo suggerimento riguarda, se serve, la definizione della funzione di lettura di un albero da file; questo adesso e' un po' prematuro (bisogna pensare a delineare la main() che useremo … ma puo' essere utile per fare mente locale sul processo di costruzione dell'albero che useremo.

Suggerimento 3:
per costruire l'albero si usa la chiamata alla funzione
`TipoAlbero leggiAlberoDaFile(char *nomeFile);`
questa funzione deve

- aprire il file di nome `nomeFile` e associarlo ad una opportuna variabile FILE *
- leggere (per consumarla dall'input) la `'('` di inizio albero
- leggere un altro carattere
    - se questo carattere e' ')'`')'` abbiamo incontrato i caratteri "()" di un albero vuoto e quindi viene restituito NULL
    - altrimenti (e se la forma parentetica e' corretta, cosa che assumiamo sia vera) abbiamo un vero albero, con l'informazione da assegnare alla radice pronta per essere letta dal file, per cui dobbiamo
        - allocare un nodo
        - leggere e assegnare al campo info l'informazione dal file
        - assegnare al campo sin il sottoalbero sinistro (che e' pronto per essere letto dal file in questo momento)
        - assegnare al campo des il sottoalbero destro (che sara' pronto per essere letto dal file appena finita la lettura del sinistro)
        - leggere (per consumarla dall'input) la ')' che termina questo albero

******* se si vuole si puo' fare un abbozzo di questa funzione, da raffinare quando avremo stabilito la forma della funzione main()

******* I prossimi due suggerimenti riguardano:

- la dichiarazione delle funzioni da usare nella `main()`
- la `main()`

Suggerimento 4:

```
/* prototipi delle funzioni usate */

void leggiElemAlberoDaFile(FILE *fin, TipoElemAlbero *pelem);
void stampaElemAlbero(TipoElemAlbero elem);
TipoAlbero leggiAlberoDaFile(FILE *fin);
TipoAlbero leggiSottoAlberoDaFile(FILE *fin);
void stampaAlbero(TipoAlbero alb);
void deallocaAlbero(TipoAlbero *pAlb);
```

---------- STOP (segue sugg.) ----------

Suggerimento 5:
questa potrebbe essere la funzione main() ...

```
int main() {
 TipoAlbero albero;
 char nomeFile[20];   /* per il nome del file */

printf("\n - nome del file con la rappr. par.: ");
 scanf("%s", nomeFile);

 albero = leggiAlberoDaFile(nomeFile);

 printf("\n - ecco l'albero: ");
 stampaAlbero(albero);

 printf("\n - adesso dealloco l'albero: ");
 deallocaAlbero(&albero);

printf("\nFINE\n");
return 0;
}
```

******* A questo punto bisogna affrontare e risolvere la funzione di lettura albero da file
I prossimi 4 suggerimenti riguardano in particolare

- la lettura di un oggetto di tipo TipoElemALBERO, eseguita nella funzione leggiAlberoDaFile(); questa lettura puo' essere fatta "ad hoc" con una scanf, oppure attraverso l'uso di una funzione apposita leggiElemAlberoDaFile; quest'ultima soluzione permette di scrivere la funzione di lettura dell'albero in modo piu' generale e riutilizzabile (infatti se dovessimo gestire alberi di altre cose che interi, dovremmo cambiare la funzione leggiElem (caratteristica di TipoElem) e non quella di lettura dell'albero

- la stesura della funzione leggiAlberoDaFile()

- la gestione della funzione leggiSottoAlberoDaFile(), importante per l'algoritmo applicato in leggiAlberoDaFile().

Suggerimento 6:
Per leggere un'informazione di tipo `TipoElemAlbero`, invece di usare `scanf("%d"...)`, che andrebbe bene solo per alberi di interi, usiamo una chiamata alla funzione `leggiElemAlberoDaFile(FILE * f, TipoElemAlbero *el)`.

In questo modo, quando dovremo usare la funzione di costruzione dell'albero per alberi con elementi diversi (ad esempio, caratteri), la funzione di costruzione rimarra' intatta mentre dovremo aggiornare solo la funzione `leggiElemAlberoDaFile()` (in altre parole, non sara' necessario mettere le mani nella funzione di costruzione, che e' piu' complicata, mentre bastera' ritoccare una funzione meno complicata) .

******* Su come e' fatta questa funzione ritorneremo in un suggerimento verso la fine della sezione; adesso torniamo alle funzioni piu' complesse …

Suggerimento 7:

provare a completare questo (corposo) brano della funzione `leggiAlberoDaFile()`

```
TipoAlbero leggiAlberoDaFile(char *nomeDelFile){
  TipoAlbero res;
  TipoElemAlbero el;
  char ccc;
  FILE *f;

  f = fopen(nomeDelFile, "r");     /* apertura file */

  fscanf(f, "%c", &ccc);           /* legge la parentesi aperta */
  fscanf(f, "%c", &ccc);           /* legge carattere successivo */

  if (ccc == ')')
    return NULL;              /* ha letto (): albero vuoto */
  else {
    /* ccc non era ')', quindi c'e' da leggere l'informazione della radice */
        leggiElemAlberoDaFile(f, &el);
    /* e poi creare il nodo radice e avviare la costruzione
        dei due sottoalberi (prima sinistro e poi destro) */
  ...
```

Suggerimento 8:

Nella funzione `leggiAlberoDaFile()`, per costruire un sottoalbero, si usa la chiamata alla funzione

```
TipoAlbero leggiSottoAlberoDaFile(FILE *fin);
```

questa funzione

- presume che il file fin metta a disposizione i dati relativi ad un (sotto)albero
- (inotre presume che il cursore di lettura del file sia posizionato proprio sulla parentesi iniziale del (sotto)albero) e
- legge tali dati, dalla '(' di inizio albero alla ')' di fine albero

Per far cio', ovviamente, bisogna fare qualcosa di molto simile a quanto fatto nella funzione `leggiAlberoDaFile()`:

- leggere dal file `fin` la '(' di inizio albero
- leggere dal file `fin` un altro carattere
  - se questo carattere e' ')' abbiamo incontrato i caratteri "()" di un albero vuoto e quindi viene restituito NULL (il sottoalbero letto e' vuoto!)
  - altrimenti abbiamo un vero albero, con l'informazione da assegnare alla radice pronta per essere letta dal file, per cui dobbiamo
    - allocare un nodo
    - leggere dal file `fin` e assegnare al campo info l'informazione dal file
    - costruire il sottoalbero sinistro (sul cui inizio e' posizionato il cursore di lettura del filea) e assegnarlo al campo sin
    - ne frattempo il cursore di input e' transitato sul sottoalbero sinistro e quindi ora e' posizionato sul sottoalbero destro, per cui dobbiamo costruire il sottoalbero destro e assegnarlo al campo des
    - leggere dal file `fin` la ')' che termina questo albero

Suggerimento 9:

provare a completare questo (corposo) brano della funzione leggiSottoAlberoDaFile()

```
TipoAlbero leggiSottoAlberoDaFile(FILE * fin){
  TipoAlbero res;
  TipoElemAlbero el;
  char ccc;

  fscanf(fin, "%c", &ccc);       /* legge la parentesi aperta */
  fscanf(fin, "%c", &ccc);       /* legge carattere successivo */

  if (ccc == ')')
    return NULL;              /* ha letto (): albero vuoto */
  else {
    /* ccc non era ')', quindi c'e' da leggere l'informazione della radice */
    leggiElemAlberoDaFile(fin, &el);
   /* ora si crea il nodo radice e si avvia la costruzione dei due
sottoalberi (prima sinistro e poi destro) */
  ...
```

***** Torniamo sulle funzioni accessorie: lettura da file e stampa di elementi dell'albero.**

Ora bisogna programmare la funzione di lettura da file di un elemento da inserire in un albero

e quella di stampa dell'elemento medesimo

nonché poi la funzione che produce la stampa dell'albero in forma parentetica (usando la funzione di stampa-elemento)

La prima funzione viene chiamata dagli algoritmi di costruzione albero: e' importante che essa sia una funzione "generica" (che incapsula in sé stessa le necessità e le caratteristiche di gestione/uso particolari del TipoElem in questione);

la seconda viene usata nella stampa dell'albero l'albero in forma parentetica.

Provare a progettare e implementare tali funzioni; seguono suggerimenti in proposito.

Suggerimento 10:
`leggiElemAlberoDaFile()` e `stampaElemAlbero()`sono funzioni programmate ad hoc per il tipo dell'informazione contenuta nei nodi dell'albero (`TipoElemAlbero`, che in questo esercizio e' `int`). La prima riceve una variabile `FILE*` corrispondente ad un file aperto in lettura e l'indirizzo di una variabile di tipo `TipoElemAlbero`, che riempie con la lettura fatta dal file; la seconda riceve un parametro di tipo `TipoElemAlbero` e lo stampa. Se l'albero e' di interi, si tratta di volgari `scanf` e `printf` con formato di conversione `"%d"`!

Suggerimento 11:
La stampa di un (sotto)albero, resa in forma parentetica prevede la

- stampa della '('
- stampa dell'informazione collegata alla radice
- stampa del sottoalbero sin
- stampa del sottoalbero des
- stampa della ')' di fine (sotto)albero;

Pero', se il (sotto)albero e' vuoto bisogna stampare solo `"()"`.

8.2. Costruzione di un albero di caratteri (directory ALBERI2 e ALBERI2_H)

Scrivere un programma che

- riceve il nome di un file contenente la rappresentazione parentetica di un albero binario di CARATTERI;
- ne costruisce l'albero binario in memoria centrale, mediante struct e puntatori;
- stampa l'albero tre volte, limitando le stampe alle sole informazioni contenute nei nodi (niente parentesi):
  - la prima volta attuando una *strategia di visita* in preordine;
  - la seconda volta attuando una *strategia di visita* in postordine,
  - la terza volta attuando una *strategia di visita* simmetrica;
- poi dealloca poi l'albero e saluta.

Nei file `ACHAR1.TXT`, `ACHAR2.TXT` ci sono le rappresentazioni parentetiche di tre alberi. Provare con queste il programma.

L'esercizio puo' essere svolto secondo due approcci: si puo' scegliere di costruire completamente il programma in un unico file; oppure si puo' decidere di costruire file header in cui siano definite/dichiarate le strutture dati e le funzioni usate nella main, e includere questi file in quello con il programma principale.

I suggerimenti seguenti sono stati sviluppati seguendo il primo, ma sono pienamente leggibili e utili in tutti e due gli approcci.

Una proposta di soluzione, nel primo approccio, e' data nella directory ALBERI2 (file alberi2.c).

Una proposta di soluzione sviluppata con il secondo approccio e' data nella directory ALBERI2_H.

In entrambe le directory (ci sono opportuni file README.TXT da leggere, se interessa)

Suggerimento 1 di 3:

Se l'albero e' il seguente
`( A( B()())( C()()))`
l'output del programma sara' (potrebbe essere)

```
- nome del file con la rappr. par.: ACHAR1.TXT

 - stampa in preordine:  A B C
 - stampa in postordine: B C A
 - stampa in simmetrica: B A C
 - adesso dealloco l'albero:
FINE
```

Se l'albero e' il seguente
`( D( B( A()())( C()()))( F( E()())()))`
l'output del programma sara' (potrebbe essere)

```
- nome del file con la rappr. par.: ACHAR2.TXT

 -  stampa in preordine: D B A C F E
 - stampa in postordine: A C B E F D
 - stampa in simmetrica: A B C D E F
 - adesso dealloco l'albero:
FINE
```

Suggerimento 2 di 3:
Nella definizione dei tipi di dato c'e' un'unica modifica da fare!

```
typedef char TipoElemAlbero;          /* albero di caratteri */
```

il resto non cambia rispetto ad ALBERI1.C (esercizio precedente).

Suggerimento 3 di 3:
Ecco i prototipi delle funzioni usate nella proposta di soluzione di `alberi2.c`:

- queste funzioni vanno cambiate rispetto all'esercizio precedente (dato che adesso si lavora con alberi di caratteri e non piu' di interi):

```
void leggiElemAlberoDaFile(FILE *fin, TipoElemAlbero *pelem);
void stampaElemAlbero(TipoElemAlbero elem);
```

- queste altre rimangono intatte (evviva: considerando lo sforzo fatto per produrle, e' un'ottima cosa il non doverci rimettere mano; i cambiamenti dovuti al cambio di tipo di informazione nei nodi si riflettono solo sulle funzioni "ad hoc" di lettura e stampa.

```
TipoAlbero leggiAlberoDaFile(char *nomeFile);
TipoAlbero leggiSottoAlberoDaFile(FILE *fin);
void deallocaAlbero(TipoAlbero *pAlb);
```

- queste sono poi le funzioni da definite ex novo, relative alla stampa secondo le varie strategie

```
void stampaAlberoPreordine(TipoAlbero alb);
void stampaAlberoPostordine(TipoAlbero alb);
void stampaAlberoSimmetrica(TipoAlbero alb);
```

8.3. Esercizio da inventare (nessuna soluzione proposta)

Ripercorrendo gli esercizi svolti durante la seconda lezione sugli alberi, sperimentare in un unico programma l'uso degli algoritmi visti a proposito di

- ricerca di elementi in un albero binario generico;
- calcolo del numero di occorrenze di un elemento;
- calcolo del massimo elemento contenuto in un albero;

8.4. Alberi di ricerca (nessuna soluzione proposta)

Scrivere un programma capace di costruire un albero di interi a partire dalla sua rappresentazione parentetica, verificando **poi** che si tratti di un albero di ricerca e permettendo successivamente di eseguire una serie di ricerche di elementi nell'albero, fatte usando l'algoritmo piu' conveniente.

La verifica che l'albero sia di ricerca viene effettuata applicando la definizione:

1) sappiamo che se il massimo degli elementi del sottoalbero sinistro e' maggiore della radice, l'albero NON e' di ricerca;
2) sappiamo anche che se il minimo degli elementi del sottoalbero destro e' minore della radice, l'albero NON e' di ricerca (infatti in tal caso non e' vero che tutti gli elementi del sottoalbero destro sono piu' grandi della radice);
3) e abbiamo riflettuto sul fatto che se il massimo del sottoalbero destro e' piu' piccolo della radice e il minimo del sottoalbero sinistro e' piu' grande della radice, allora l'albero complessivo "potrebbe" essere di ricerca … a patto che i sottoalberi sinistro e destro lo siano.

Quindi la verifica deve procedere ricorsivamente su tutti i possibili sottoalberi.
diRicerca(ALBERO)

ALB vuoto → **sì, è di ricerca**

ALB non vuoto → se (MASSIMO (SIN) < RAD) AND (MINIMO (DES) >RAD)
(c'è RAD, SIN, DES)
allora restituiamo il risultato di
diRicerca(SIN)
AND
diRicerca(DES)
altrimenti
no, non è di ricerca

Suggerimento

Abbiamo visto che un potenziale problema e' nel calcolo del massimo (minimo) di un albero: si puo' procedere, a tale riguardo, in modi diversi;

comunque ecco un modo per calcolare il massimo tra i valori di un albero (supponiamo che le informazioni nei nodi siano tali da poter usare l'operatore '>')

```
int maxAlb ( TipoAlb a )
{
   int m, maxPerA = a-> info;

   if ( a -> sin )
   {
        m = maxAlb ( a->sin);
        if( m > maxPerA)
        maxPerA = m;
   }
   if ( a-> des )
   {
        m = maxAlb( a->des );
        if ( m > maxPerA )
             maxPerA = m;
   }
   return maxPerA;
}
```