Primo Appello di Programmazione I

15 Gennaio 2008
Prof. Roberto Sebastiani

Codice:

Nome	Cognome	Matricola

La directory 'esame' contiene 4 sotto-directory: 'uno', 'due', 'tre' e 'quattro'. Le soluzioni vanno scritte negli spazi e nei modi indicati esercizio per esercizio.
NOTA: il codice dato non può essere modificato

Modalità di questo appello

Durante la prova gli studenti sono vincolati a seguire le regole seguenti:

- Non è consentito l'uso di alcun libro di testo o fotocopia. In caso lo studente necessitasse di carta (?), gli/le verranno forniti fogli di carta bianca su richiesta, che dovranno essere riconsegnati a fine prova. È consentito l'uso di una penna. Non è consentito l'uso di alcuno strumento calcolatore.
- È vietato lo scambio di qualsiasi informazione, orale o scritta. È vietato guardare nel terminale del vicino.
- È vietato l'uso di telefoni cellulari o di qualsiasi strumento elettronico.
- È vietato allontanarsi dall'aula durante la prova, anche se si ha già consegnato. (Ogni necessità fisiologica va espletata PRIMA dell'inizio della prova.)
- È vietato qualunque accesso, in lettura o scrittura, a file esterni alla directory di lavoro assegnata a ciascun studente. Le uniche operazioni consentite sono l'apertura, l'editing, la copia, la rimozione e la compilazione di file all'interno della propria directory di lavoro.
- Sono ovviamente vietati l'uso di email, ftp, ssh, telnet ed ogni strumento che consenta di accedere a file esterni alla directory di lavoro. Le operazioni di copia, rimozione e spostamento di file devono essere circoscritte alla directory di lavoro.
- Ogni altra attività non espressamente citata qui sopra o autorizzata dal docente è vietata.

Ogni violazione delle regole di cui sopra comporterà automaticamente l'annullamento della prova e il divieto di accesso ad un certo numero di appelli successivi, a seconda della gravità e della recidività della violazione.

NOTA IMPORTANTE: DURANTE LA PROVA PER OGNI STUDENTE VERRÀ ATTIVATO UN TRACCIATORE SOFTWARE CHE REGISTRERÀ TUTTE LE OPERAZIONI ESEGUITE (ANCHE ALL'INTERNO DELL'EDITOR!!). L'ANNULLAMENTO DELLA PROVA DI UNO STUDENTE POTRÀ AVVENIRE ANCHE IN UN SECONDO MOMENTO, SE L'ANALISI DELLE TRACCE SOFTWARE RIVELASSERO IRREGOLARITÀ.

1 Scrivere un programma, nel file `esercizio1.cc`, che, presi come argomenti del `main` i nomi di due file ed un carattere, copi fino ad un massimo di 100 caratteri del primo file nel secondo ad eccezione del carattere specificato come argomento. Se ad esempio l'eseguibile è `a.out`, il comando

```
./a.out input output c
```

creerà un nuovo file di nome `output` e vi copierà il contenuto del file denominato `input`, leggendo al massimo 100 caratteri e non inserendo gli eventuali caratteri 'c' presenti nel file `input`. Per semplicità si assuma che il testo contenuto nel file di input sia su un'unica riga.

Nelle figure 1 e 2, e 3 e 4 riportiamo due esempi di file `input` e `output`.

```
Filastrocca delle parole: Fatevi avanti! Chi ne vuole?
```

Figura 1: `input`

```
Filastroa delle parole: Fatevi avanti! Chi ne vuole?
```

Figura 2: `output`

```
Filastrocca delle parole: Fatevi avanti! Chi ne vuole? Di parole ho la testa
piena, con dentro la luna e la balena. Ci sono parole per gli amici: Buon
giorno, Buon anno, Siate felici! Parole belle e parole buone; parole per ogni
sorta di persone. Di G. Rodari.
```

Figura 3: `input1`

```
Filastroa delle parole: Fatevi avanti! Chi ne vuole? Di parole ho la testa
piena, on dentro la lu
```

Figura 4: `output1`

NOTA: nel conteggio dei caratteri vanno compresi anche i caratteri scartati.

VALUTAZIONE: questo esercizio vale 6 punti (al punteggio di tutti gli esercizi va poi sommato 10).

1 **esercizio1.cc**

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc,char* argv[]){

  fstream my_in, my_out;
  char c, tmp;
  int i=0;

  if (argc!=4) {
    cout << "Usage: ./a.out <sourcefile> <newfile> <char>\n";
    exit(0);
  }

  my_in.open(argv[1],ios::in);
  my_out.open(argv[2],ios::out);
  tmp=*argv[3];

  while ((my_in.get(c)) && (i<100)) {
    i++;
    if (c!=tmp)
      my_out.put(c);
  }

  my_in.close();
  my_out.close();
  return(0);
}
```

1 Scrivere un programma, nel file `esercizio1.cc`, che, presi come argomenti del `main` i nomi di due file ed un carattere, copi fino ad un massimo di 100 caratteri del primo file nel secondo duplicando tutti i caratteri uguali al carattere specificato come argomento. Se ad esempio l'eseguibile è `a.out`, il comando

```
./a.out input output c
```

creerà un nuovo file di nome `output` e vi copierà il contenuto del file denominato `input`, leggendo al massimo 100 caratteri e duplicando gli eventuali caratteri 'c' presenti nel file `input`. Per semplicità si assuma che il testo contenuto nel file di input sia su un'unica riga.

Nelle figure 1 e 2, e 3 e 4 riportiamo due esempi di file `input` e `output`.

```
Filastrocca delle parole: Fatevi avanti! Chi ne vuole?
```

Figura 5: `input`

```
Filastrocccca delle parole: Fatevi avanti! Chi ne vuole?
```

Figura 6: `output`

```
Filastrocca delle parole: Fatevi avanti! Chi ne vuole? Di parole ho la testa
piena, con dentro la luna e la balena. Ci sono parole per gli amici: Buon
giorno, Buon anno, Siate felici! Parole belle e parole buone; parole per ogni
sorta di persone. Di G. Rodari.
```

Figura 7: `input1`

```
Filastrocccca delle parole: Fatevi avanti! Chi ne vuole? Di parole ho la testa
piena, ccon dentro la lu
```

Figura 8: `output1`

NOTA: ogni carattere duplicato va contato una volta sola.

VALUTAZIONE: questo esercizio vale 6 punti (al punteggio di tutti gli esercizi va poi sommato 10).

1 **esercizio1.cc**

```
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc,char* argv[]){

  fstream my_in, my_out;
  char c, tmp;
  int i=0;

  if (argc!=4) {
    cout << "Usage: ./a.out <sourcefile> <newfile> <char>\n";
    exit(0);
  }

  my_in.open(argv[1],ios::in);
  my_out.open(argv[2],ios::out);
  tmp=*argv[3];

  while ((my_in.get(c)) && (i<100)) {
    i++;
    if (c==tmp)
      my_out.put(c);
      my_out.put(c);
  }

  my_in.close();
  my_out.close();
  return(0);
}
```

2 Nel file `esercizio2.cc` scrivere la procedura **ricorsiva** `merge_ricorsivo` che, presi come parametri due array di interi ordinati secondo ordine **crescente** e le loro dimensioni, realizzi utilizzando **la ricorsione** il *merge* dei due array in un terzo array passato anch'esso come parametro.

Si ricordi che l'operazione di *merge* (fusione) è quell'operazione che, presi due array distinti ed ordinati (si suppongano essere rispettivamente di dimensione `n` ed `m`), produce un nuovo array contenente tutti e soli gli elementi contenuti nei due array originari (quindi avente dimensione `n+m`), rispettando nuovamente l'ordinamento prescelto (in questo caso **crescente**).

Si supponga ad esempio di ricevere in ingresso gli array ordinati:

0	1	5	8	9	9	30

4	8	15	16	23	42

rispettivamente costituiti da 7 e 6 elementi. La procedura `merge_ricorsivo` dovrà produrre il seguente array:

0	1	4	5	8	8	9	9	15	16	23	30	42

contenente 13 elementi, anch'essi ordinati in ordine **crescente**.

Notare che il file `esercizio2.cc` contiene già le procedure di acquisizione e stampa degli array (che non devono essere modificate). Si supponga, inoltre, che l'utente inserirsca sempre e comunque vettori di interi già ordinati in modo corretto.

N.B.: **La funzione `merge_ricorsivo` non può essere iterativa: al suo interno, non ci possono quindi essere cicli o chiamate ad altre funzioni contenenti cicli. Vi possono essere chiamate a altre funzioni purché ricorsive.**

Inoltre, all'interno di questo programma non è ammesso l'utilizzo di variabili globali o di tipo `static`, ad eccezione di quelle eventualmente già presenti.

VALUTAZIONE: questo esercizio vale 7 punti (al punteggio di tutti gli esercizi va poi sommato 10).

2 **esercizio2.cc**

```
using namespace std;
#include <iostream>
#include <cstdlib>

const int MAX_DIM = 100;
int acquisisci_array (int v[]);
void stampa_array (int v[], int d);

// Inserire qui la dichiarazione delle funzioni usate
void merge_ricorsivo (int *v1, int d1, int *v2, int d2, int *vm);

int main ()
{
  int d1, d2;
  int v1[MAX_DIM];
  int v2[MAX_DIM];
  int v_merge[2 * MAX_DIM];

  cout << "Primo vettore." << endl;
  d1 = acquisisci_array(v1);
  cout << "Secondo vettore." << endl;
  d2 = acquisisci_array(v2);

  merge_ricorsivo(v1, d1, v2, d2, v_merge);

  stampa_array(v_merge, d1 + d2);
  return 0;
}

int acquisisci_array (int v[])
{
  int d;
  cout << "Inserire la dimensione del vettore (compresa tra 1 e "
       << MAX_DIM << "): ";
  cin >> d;
  cout << "Inserire gli elementi del vettore: " << endl;
  for (int i = 0; i < d; i++)
    cin >> v[i];
  return d;
}

void stampa_array (int v[], int d)
{
  for (int i = 0; i < d; i++)
    cout << v[i] << " ";
  cout << endl;
}



// Inserire qui la definizione delle funzioni usate
```

```
void merge_ricorsivo (int *v1, int d1, int *v2, int d2, int *vm)
{
  if ((d1 > 0) && ((d2 == 0) || (v1[0] <= v2[0])))  {
    vm[0] = v1[0];
    merge_ricorsivo(v1+1, d1-1, v2, d2, vm+1);
  }
  else if (d2 > 0) {
    vm[0] = v2[0];
    merge_ricorsivo(v1, d1, v2+1, d2-1, vm+1);
  }
}

/* //Alternativa 1
void merge_ricorsivo1 (int *v1, int i1, int d1, int *v2, int i2, int d2, int *vm, int im) {
  if (i1<=d1 && ( i2>d2 || v1[i1]<=v2[i2])) {
      vm[im]=v1[i1];
      merge_ricorsivo1(v1, i1+1, d1, v2, i2, d2, vm, im+1);
  }
  else if (i2<=d2) {
      vm[im]=v2[i2];
      merge_ricorsivo1(v1, i1, d1, v2, i2+1, d2, vm, im+1);
  }
}

void merge_ricorsivo (int *v1, int d1, int *v2, int d2, int *vm)
{
  merge_ricorsivo1(v1, 0, d1-1, v2, 0, d2, vm, 0);
}
*/

/* //Alternativa 2
void merge_ricorsivo (int *v1, int d1, int *v2, int d2, int *vm)
{
  if (d1>0 && d2>0) {
    if (*v1 <= *v2) {
      *vm = *v1;
      merge_ricorsivo(v1+1, d1-1, v2, d2, vm+1);
    }
    else {
      *vm = *v2;
      merge_ricorsivo(v1, d1, v2+1, d2-1, vm+1);
    }
  }
  else if (d1>0) {
    *vm = *v1;
    merge_ricorsivo(v1+1, d1-1, v2, d2, vm+1);
  }
  else if (d2>0) {
    *vm = *v2;
    merge_ricorsivo(v1, d1, v2+1, d2-1, vm+1);
  }
}
*/
```

2 Nel file `esercizio2.cc` scrivere la procedura **ricorsiva** `merge_ricorsivo` che, presi come parametri due array di interi ordinati secondo ordine **decrescente** e le loro dimensioni, realizzi utilizzando **la ricorsione** il *merge* dei due array in un terzo array passato anch'esso come parametro.

Si ricordi che l'operazione di *merge* (fusione) è quell'operazione che, presi due array distinti ed ordinati (si suppongano essere rispettivamente di dimensione `n` ed `m`), produce un nuovo array contenente tutti e soli gli elementi contenuti nei due array originari (quindi avente dimensione `n+m`), rispettando nuovamente l'ordinamento prescelto (in questo caso **decrescente**).

Ad esempio, la procedura `merge_ricorsivo` dovrà produrre il seguente array:

70	60	42	30	23	16	15	15	10	10	8	5	4	1

contenente 14 elementi ordinati in ordine **decrescente**, se si suppone di aver ricevuto in ingresso gli array ordinati:

42	23	16	15	8	4

70	60	30	15	10	10	5	1

rispettivamente costituiti da 6 ed 8 elementi.

Notare che il file `esercizio2.cc` contiene già le procedure di acquisizione e stampa degli array (che non devono essere modificate). Si supponga, inoltre, che l'utente inserirsca sempre e comunque vettori di interi già ordinati in modo corretto.

N.B.: **La funzione `merge_ricorsivo` non può essere iterativa: al suo interno, non ci possono quindi essere cicli o chiamate ad altre funzioni contenenti cicli. Vi possono essere chiamate a altre funzioni purché ricorsive.**

Inoltre, all'interno di questo programma non è ammesso l'utilizzo di variabili globali o di tipo `static`, ad eccezione di quelle eventualmente già presenti.

VALUTAZIONE: questo esercizio vale 7 punti (al punteggio di tutti gli esercizi va poi sommato 10).

2 **esercizio2.cc**

```
using namespace std;
#include <iostream>
#include <cstdlib>

const int MAX_DIM = 100;
void stampa_array (int v[], int d);
int acquisisci_array (int v[]);

// Inserire qui la dichiarazione delle funzioni usate
void merge_ricorsivo (int *v1, int d1, int *v2, int d2, int *vm);

int main ()
{
  int d1, d2;
  int v1[MAX_DIM];
  int v2[MAX_DIM];
  int v_merge[2 * MAX_DIM];

  cout << "Primo vettore." << endl;
  d1 = acquisisci_array(v1);
  cout << "Secondo vettore." << endl;
  d2 = acquisisci_array(v2);

  merge_ricorsivo(v1, d1, v2, d2, v_merge);

  stampa_array(v_merge, d1 + d2);
  return 0;
}

void stampa_array (int v[], int d)
{
  for (int i = 0; i < d; i++)
    cout << v[i] << " ";
  cout << endl;
}

int acquisisci_array (int v[])
{
  int d;
  cout << "Inserire la dimensione del vettore (compresa tra 1 e "
       << MAX_DIM << "): ";
  cin >> d;
  cout << "Inserire gli elementi del vettore: " << endl;
  for (int i = 0; i < d; i++)
    cin >> v[i];
  return d;
}



// Inserire qui la definizione delle funzioni usate
```

```
void merge_ricorsivo (int *v1, int d1, int *v2, int d2, int *vm)
{
  if ((d1 > 0) && ((d2 == 0) || (v1[0] >= v2[0])))  {
    vm[0] = v1[0];
    merge_ricorsivo(v1+1, d1-1, v2, d2, vm+1);
  }
  else if (d2 > 0) {
    vm[0] = v2[0];
    merge_ricorsivo(v1, d1, v2+1, d2-1, vm+1);
  }
}

/* //Alternativa 1
void merge_ricorsivo1 (int *v1, int i1, int d1, int *v2, int i2, int d2, int *vm, int im) {
  if (i1<=d1 && ( i2>d2 || v1[i1]>=v2[i2])) {
      vm[im]=v1[i1];
      merge_ricorsivo1(v1, i1+1, d1, v2, i2, d2, vm, im+1);
  }
  else if (i2<=d2) {
      vm[im]=v2[i2];
      merge_ricorsivo1(v1, i1, d1, v2, i2+1, d2, vm, im+1);
  }
}

void merge_ricorsivo (int *v1, int d1, int *v2, int d2, int *vm)
{
  merge_ricorsivo1(v1, 0, d1-1, v2, 0, d2, vm, 0);
}
*/

/* //Alternativa 2
void merge_ricorsivo (int *v1, int d1, int *v2, int d2, int *vm)
{
  if (d1>0 && d2>0) {
    if (*v1 >= *v2) {
      *vm = *v1;
      merge_ricorsivo(v1+1, d1-1, v2, d2, vm+1);
    }
    else {
      *vm = *v2;
      merge_ricorsivo(v1, d1, v2+1, d2-1, vm+1);
    }
  }
  else if (d1>0) {
    *vm = *v1;
    merge_ricorsivo(v1+1, d1-1, v2, d2, vm+1);
  }
  else if (d2>0) {
    *vm = *v2;
    merge_ricorsivo(v1, d1, v2+1, d2-1, vm+1);
  }
}
*/
```

3 Nel file `albero_main.cc` è definita la funzione `main` che contiene un menu per gestire un albero binario di ricerca di `double`. Scrivere, in un nuovo file `albero.cc`, le definizioni delle funzioni dichiarate nello header file `albero.h` in modo tale che:

- `init` inizializzi l'albero;
- `empty` controlli se l'albero è vuoto, restituendo `TRUE` in caso affermativo e `FALSE` in caso contrario;
- `insert` inserisca l'elemento passato come parametro nell'albero, restituendo `TRUE` se l'operazione è andata a buon fine, e `FALSE` altrimenti (cioè se l'elemento è già presente). L'albero deve essere ordinato in maniera **crescente**. Esempio: l'inserimento dei seguenti valori:

  ```
  5.5  2  10
  ```

  deve produrre il seguente albero:

  ```
          5.5
  2              10
  ```

- `search` cerchi nell'albero l'elemento passato in input, resituendo `TRUE` se l'elemento è presente, e `FALSE` altrimenti;
- `print` stampi a video il contenuto dell'albero, in ordine **crescente**. Esempio: l'albero qui sopra deve essere stampato come:

  ```
  2  5.5  10
  ```

VALUTAZIONE: questo esercizio vale 7 punti (al punteggio di tutti gli esercizi va poi sommato 10).

3 albero_main.cc

```
using namespace std;
#include <iostream>
#include "albero.h"

int main()
{
    char res;
    Tree tree;
    double val;
    init(tree);
    do {
        cout << "\nOperazioni possibili:\n"
             << " Inserimento (i)\n"
             << " Ricerca (r)\n"
             << " Stampa ordinata (s)\n"
             << " Fine (f)\n";
        cout << "Operazione selezionata: ";
        cin >> res;
        switch (res) {
        case 'i':
            cout << "Valore : ";
            cin >> val;
            if (insert(tree, val) == FALSE) {
                cout << "Valore gia` presente!" << endl;
            }
            break;
        case 'r':
            cout << "Valore: ";
            cin >> val;
            if (search(tree, val) == TRUE) {
                cout << "Valore presente: " << val << endl;
            } else {
                cout << "Valore non presente" << endl;
            }
            break;
        case 's':
            if (empty(tree) == TRUE) {
                cout << "Albero vuoto!" << endl;
            } else {
                print(tree);
            }
            break;
        case 'f':
            break;
        default:
            cout << "Opzione errata\n";
        }
    } while (res != 'f');

    return 0;
}
```

3 albero.h

```
// -*- C++ -*-
#ifndef ALBERO_H
#define ALBERO_H

struct Node {
    double val;
    Node *left;
    Node *right;
};

typedef Node * Tree;

enum boolean { FALSE, TRUE };

void init(Tree &t);
boolean empty(const Tree &t);
boolean insert(Tree &t, double val);
boolean search(const Tree &t, double val);
void print(const Tree &t);

#endif
```

3 soluzione_A31.cc

```
#include <iostream>
using namespace std;
#include "albero.h"

void init(Tree &t)
{
    t = NULL;
}


boolean empty(const Tree &t)
{
    return (t == NULL) ? TRUE : FALSE;
}


boolean insert(Tree &t, double val)
{
    // caso base
    if (empty(t) == TRUE) {
        t = new Node;
        t->val = val;
        t->left = t->right = NULL;
        return TRUE;
    }
    // caso ricorsivo. Controllo se scendere a sinistra o a destra
    if (val < t->val) {
        // scendo a sinistra
```

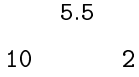
```
        return insert(t->left, val);
    } else if (val > t->val) {
        // scendo a destra
        return insert(t->right, val);
    } else {
        // elemento gia' presente, restituisco false
        return FALSE;
    }
}



boolean search(const Tree &t, double val)
{
    if (empty(t) == TRUE) {
        return FALSE;
    } else if (val == t->val) {
        return TRUE;
    } else if (val < t->val) {
        // scendo a sinistra
        return search(t->left, val);
    } else {
        // scendo a destra
        return search(t->right, val);
    }
}


void print(const Tree &t)
{
    if (empty(t) == FALSE) {
        // prima stampo gli elementi minori di t->val (cioe' quelli a sx)
        print(t->left);
        // poi stampo t->val
        cout << t-> val << ' ';
        // poi stampo gli elementi maggiori (cioe' quelli a dx)
        print(t->right);
    }
}
```

3 Nel file `albero_main.cc` è definita la funzione `main` che contiene un menu per gestire un albero binario di ricerca di `double`. Scrivere, in un nuovo file `albero.cc`, le definizioni delle funzioni dichiarate nello header file `albero.h` in modo tale che:

- `init` inizializzi l'albero;
- `empty` controlli se l'albero è vuoto, restituendo `TRUE` in caso affermativo e `FALSE` in caso contrario;
- `insert` inserisca l'elemento passato come parametro nell'albero, restituendo `TRUE` se l'operazione è andata a buon fine, e `FALSE` altrimenti (cioè se l'elemento è già presente). L'albero deve essere ordinato in maniera **decrescente**. Esempio: l'inserimento dei seguenti valori:

  ```
  5.5  2  10
  ```

  deve produrre il seguente albero:

- `search` cerchi nell'albero l'elemento passato in input, resituendo `TRUE` se l'elemento è presente, e `FALSE` altrimenti;
- `print` stampi a video il contenuto dell'albero, in ordine **decrescente**. Esempio: l'albero qui sopra deve essere stampato come:

  ```
  10  5.5  2
  ```

VALUTAZIONE: questo esercizio vale 7 punti (al punteggio di tutti gli esercizi va poi sommato 10).

3 albero_main.cc

```
using namespace std;
#include <iostream>
#include "albero.h"

int main()
{
    char res;
    Tree tree;
    double val;
    init(tree);
    do {
        cout << "\nOperazioni possibili:\n"
             << " Inserimento (i)\n"
             << " Ricerca (r)\n"
             << " Stampa ordinata (s)\n"
             << " Fine (f)\n";
        cout << "Operazione selezionata: ";
        cin >> res;
        switch (res) {
        case 'i':
            cout << "Valore : ";
            cin >> val;
            if (insert(tree, val) == FALSE) {
                cout << "Valore gia` presente!" << endl;
            }
            break;
        case 'r':
            cout << "Valore: ";
            cin >> val;
            if (search(tree, val) == TRUE) {
                cout << "Valore presente: " << val << endl;
            } else {
                cout << "Valore non presente" << endl;
            }
            break;
        case 's':
            if (empty(tree) == TRUE) {
                cout << "Albero vuoto!" << endl;
            } else {
                print(tree);
            }
            break;
        case 'f':
            break;
        default:
            cout << "Opzione errata\n";
        }
    } while (res != 'f');

    return 0;
}
```

3 albero.h

```
// -*- C++ -*-
#ifndef ALBERO_H
#define ALBERO_H

struct Node {
    double val;
    Node *left;
    Node *right;
};

typedef Node * Tree;

enum boolean { FALSE, TRUE };

void init(Tree &t);
boolean empty(const Tree &t);
boolean insert(Tree &t, double val);
boolean search(const Tree &t, double val);
void print(const Tree &t);

#endif
```

3 soluzione_A32.cc

```
#include <iostream>
using namespace std;
#include "albero.h"

void init(Tree &t)
{
    t = NULL;
}


boolean empty(const Tree &t)
{
    return (t == NULL) ? TRUE : FALSE;
}


boolean insert(Tree &t, double val)
{
    // caso base
    if (empty(t) == TRUE) {
        t = new Node;
        t->val = val;
        t->left = t->right = NULL;
        return TRUE;
    }
    // caso ricorsivo. Controllo se scendere a sinistra o a destra
    if (val > t->val) {
        // scendo a sinistra
```

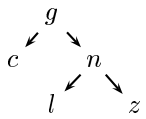
```
        return insert(t->left, val);
    } else if (val < t->val) {
        // scendo a destra
        return insert(t->right, val);
    } else {
        // elemento gia' presente, restituisco false
        return FALSE;
    }
}



boolean search(const Tree &t, double val)
{
    if (empty(t) == TRUE) {
        return FALSE;
    } else if (val == t->val) {
        return TRUE;
    } else if (val > t->val) {
        // scendo a sinistra
        return search(t->left, val);
    } else {
        // scendo a destra
        return search(t->right, val);
    }
}


void print(const Tree &t)
{
    if (empty(t) == FALSE) {
        // prima stampo gli elementi maggiori di t->val (cioe' quelli a sx)
        print(t->left);
        // poi stampo t->val
        cout << t-> val << ' ';
        // poi stampo gli elementi minori (cioe' quelli a dx)
        print(t->right);
    }
}
```

4 Dati i seguenti file:

- `tree.h` e `tree.o`, che implementano una libreria "albero di ricerca binaria di caratteri", a cui manca la funzione di stampa ordinata dell'albero;
- `stack.h` e `stack.o`, che implementano una libreria "stack di alberi";
- `esercizio4.cc` che contiene la funzione `main` con il menu che gestisce l'albero di ricerca binaria;

scrivere all'interno del file `esercizio4.cc` la definizione della funzione **iterativa** "`stampa`" che, preso in input un albero di ricerca binaria, ne stampi il contenuto in modo ordinato crescente.

Ad esempio, il seguente albero:



deve essere stampato nell'ordine 'c', 'g', 'l', 'n', 'z'.

N.B.: **La funzione non può essere ricorsiva: al suo interno, non ci possono quindi essere chiamate a sè stessa o ad altre funzioni ricorsive o mutualmente ricorsive.**

SUGGERIMENTO: utilizzare lo stack offerto nei file `stack.h` e `stack.o`.

VALUTAZIONE: questo esercizio permette di conseguire la lode se tutti gli esercizi precedenti sono corretti.

4 **esercizio4.cc**

```
#include <iostream>
#include "tree.h"
#include "stack.h"

using namespace std;

int main()
{
  char res,val;
  tree t, tmp;
  init(t);
  do {
    cout << "\nOperazioni possibili:\n"
         << "Inserimento (i)\n"
         << "Ricerca (r)\n"
         << "Stampa DFS (s)\n"
         << "Fine (f)\n";
    cin >> res;
    switch (res) {
    case 'i':
      cout << "Val? : ";
      cin >> val;
      insert(t,val);
      if (t==NULL)
        cout << "memoria esaurita!\n";
      break;
    case 'r':
      cout << "Val? : ";
      cin >> val;
      tmp=cerca(t,val);
      if (tmp!=NULL)
        cout << "Valore trovato!: " << val << endl;
      else
        cout << "Valore non trovato!\n";
      break;
    case 's':
      cout << "Stampa:\n";
      stampa(t);
      break;
    case 'f':
      break;
    default:
      cout << "Optione errata\n";
    }
  } while (res != 'f');
}


retval stampa(tree t) {
  stack_init();
  tree current = t;
  while(!vuoto(current) || !stack_vuoto()) {
```

```
    if (!vuoto(current)) {  // percorso all'ingiu'
      stack_push(current);
      current= current->left;
    }
    else { // !stack_vuoto() // torno su'
      stack_top(current);
      stack_pop();
      tree_node_print(current);
      current=current->right;
    }
  }
}
```