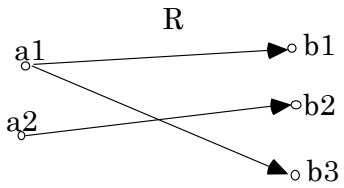
GRAFI E ALBERI

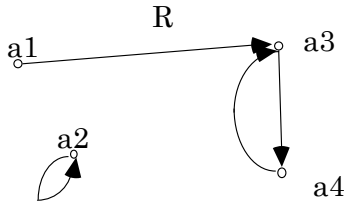
Sono strutture dati per rappresentare ***relazioni binarie*** su un insieme di elementi.

Relazione tra insiemi A, B: $R \subseteq A \times B$



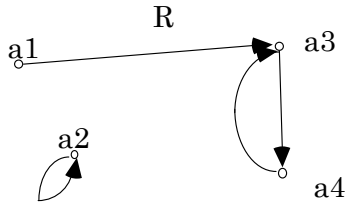
Una relazione su A={a1,a2} e B={b1,b2,b3} definisce un ***grafo bipartito*** su A e B.

Se A=B, $R \subseteq A \times A$:



Grafo orientato G=<A,R> dove:

- A, insieme finito non vuoto (***nodi***);
- R, insieme degli archi orientati (se $<a_i,a_j> \in R$ allora c'è un ***arco orientato*** da a_i ad a_j).



Grado di ingresso di a_i, numero di archi che hanno a_i come nodo finale.

Grado di uscita di a_i, numero di archi che hanno a_i come nodo iniziale.

Grafo completo, se $\forall a_i,a_j \in A$ esiste un arco da a_i ad a_j, cioè: $R = A \times A$

Il grafo in figura non è un grafo completo.

I grafi sono quindi strutture dati usate (in informatica) per rappresentare relazioni.

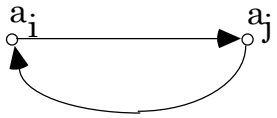
Le proprietà delle relazioni si riflettono nella struttura del grafo.

Proprietà delle relazioni:

- R è **totale** se $\forall a_i,a_j \in A$, $<a_i,a_j> \in R$ (grafo completo)

- $R \subseteq A \times A$, è **riflessiva** se $\forall a \in A$, $<a,a> \in R$

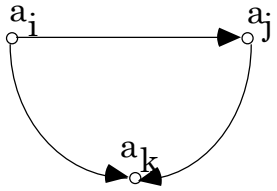
- È **simmetrica**, se $<a_i,a_j> \in R$ implica $<a_j,a_i> \in R$



(grafo simmetrico ---> grafo non orientato)

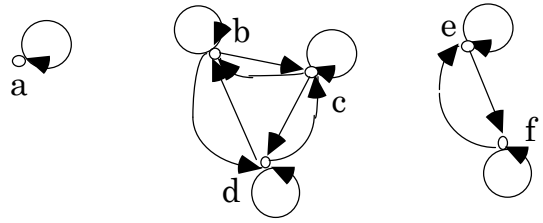
- È **antisimmetrica** se $<a_i,a_j> \in R$ implica $<a_j,a_i> \notin R$

- È **transitiva** se $<a_i,a_j> \in R$ e $<a_j,a_k> \in R$ implica $<a_i,a_k> \in R$



- **Relazione di equivalenza**, se riflessiva, simmetrica e transitiva.

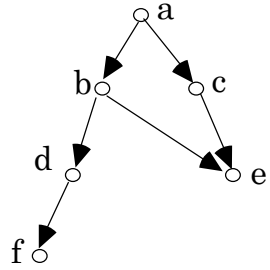
Grafo di una relazione di equivalenza:



Ogni "isola", una classe di equivalenza.

- Una relazione R è detta **relazione d'ordine** se è riflessiva, antisimmetrica, transitiva.

Il grafo viene disegnato in modo semplificato:



non viene disegnato

Cammino in un grafo:

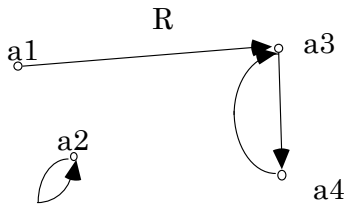
In un grafo G=<A,R>, dati due nodi $\alpha,\beta \in A$ si dice ***cammino*** da α a β una sequenza di nodi:

$$a_0, a_1, a_2, ..., a_n$$

tali che:

- n>=0;
- $a_0=\alpha$ ed $a_n=\beta$;
- per ogni $i \subseteq [0..n]$, $<a_i, a_{i+1}> \in R$

n, ***lunghezza*** del cammino.



Cammino da a1 ad a4: a1, a3, a4
(ma anche: a1, a3, a4, a3, a4)

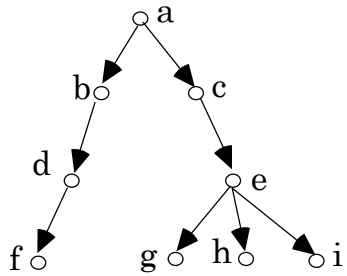
Se i nodi del cammino sono distinti, ***cammino semplice***.

Se $\alpha=\beta$, il cammino è detto ***ciclo***.

Un grafo senza cicli è detto ***aciclico***.

ALBERI

Un albero (finito) è un ***grafo orientato aciclico***, in cui esiste un nodo (***radice***) con grado di ingresso 0. Ogni altro nodo ha grado di ingresso 1.



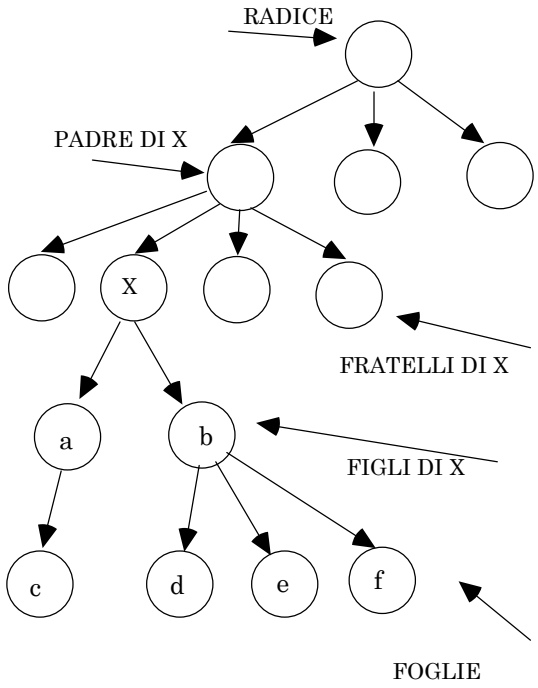
Nodi con grado di uscita 0, ***foglie***.

Rappresenta una ***relazione d'ordine parziale***.

Esiste esattamente un cammino (semplice) dalla radice a qualunque altro nodo.

Gli alberi sono usati per rappresentare relazioni gerarchiche fra oggetti.

Sono strutture dati importanti perché sono utilizzate per la realizzazione di ***indici*** in memoria centrale (***alberi binari***).



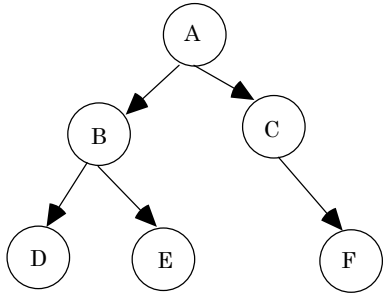
Discendenti di X, calcolati come chiusura transitiva della relazione “figlio di”.

discendenti(X)={a,b,c,d,e,f}

Altezza di un albero, lunghezza del cammino più lungo dalla radice ad una foglia.

ALBERI BINARI

Albero in cui ogni nodo ha al massimo due figli (***figlio destro*** e ***figlio sinistro***).



Definizione induttiva:

Un albero binario è un grafo orientato aciclico che o è vuoto (cioè ha un insieme vuoto di nodi), oppure è formato da un nodo radice e da due sotto-alberi binari (sinistro e destro).

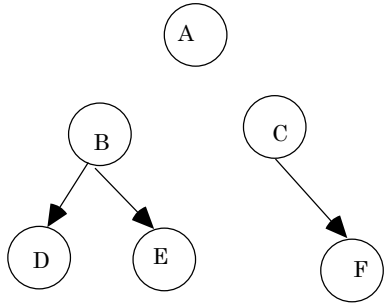
Alberi binari:

Come **tipo di dato astratto**, <S,Op,C> dove:

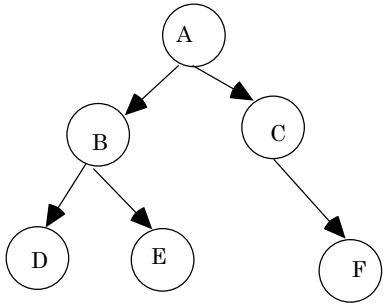
• S=(alb-bin, element_type, boolean)
con alb-bin dominio di interesse e
element_type tipo degli elementi dell'albero;

• Op=(radice, sinistro, destro, test-vuoto, costruisci)
radice: alb-bin → element_type (***root***)
sinistro: alb-bin → alb-bin (***left***)
destro: alb-bin → alb-bin (***right***)
test-vuoto: alb-bin → boolean (***empty***)
costruisci: alb-bin × element_type × alb-bin → alb-bin
(***cons_tree***)

• C=(albero-vuoto) (***empty_tree***)



COSTRUISCI



L'importanza degli alberi binari deriva dalla facilità con cui possono essere memorizzati.

Proprietà degli alberi binari:

Al massimo il **numero di nodi a livello i** (i>=0) di un albero binario vale 2^i.

Si puo` dimostrare per ***induzione***:

i=0, si ha 2^0=1 nodo (radice)

Se al livello i il numero massimo di nodi è 2^i allora al livello i+1 è 2^{i+1}.

Infatti al livello i+1 ci possono essere al massimo il doppio dei nodi presenti al livello i, poiché ogni nodo puo` avere al massimo due nodi figli: $2*2^i=2^{i+1}$

In un albero binario di altezza p, il **numero massimo di nodi** è:

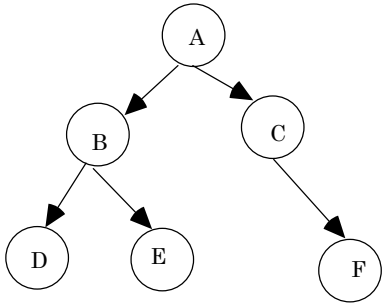
$$\sum_{(i=0..p)} 2^i = 2^{p+1} - 1$$

Un albero binario è **completo** se ha 2^{p+1} - 1 nodi, essendo p la sua profondità.

In un albero binario completo ciascun nodo (eccettuato le foglie) ha grado di uscita uguale a 2.

Algoritmi di visita per alberi binari:

Consentono di analizzare tutti i nodi dell'albero in un determinato ordine.



Preordine o ordine anticipato:

Analizza radice, albero sinistro, albero destro.

```
visita in preordine (T):
      se test-vuoto(T)=false
      allora esegui
            analizza radice(T)
            visita in preordine (sinistro(T))
            visita in preordine (destro(T))
      fine;
```

A B D E C F

Postordine o ordine ritardato:

```
Analizza albero sinistro, albero destro, radice.

visita in postordine (T):
      se test-vuoto(T) =false
      allora esegui
            visita in postordine (sinistro(T))
            visita in postordine (destro(T))
            analizza radice(T)
      fine;
```

D E B F C A

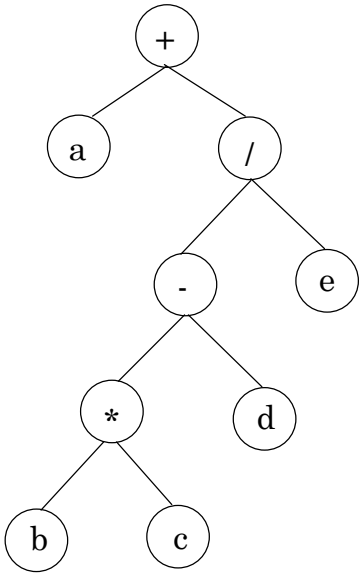
Simmetrica:

```
Analizza albero sinistro, radice, albero destro.

visita simmetrica(T):
      se test-vuoto(T)=false
      allora esegui
            visita simmetrica (sinistro(T))
            analizza radice(T)
            visita simmetrica(destro(T))
      fine;
```

D B E A C F

Esempio: a+(b*c-d)/e



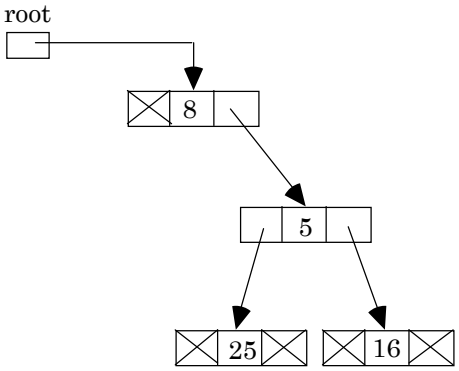
Ordine anticipato: +a/-*bcde
(**polacca prefissa**, operatore, I^{o} operando, II^{o} operando)

Ordine ritardato: abc*d-e/+
(**polacca postfissa**, I^{o} operando, II^{o} operando, operatore)

Visita simmetrica: a+b*c-d/e
(priorità degli operatori)

Rappresentazione collegata di alberi binari:

Rappresentazione attraverso ***strutture*** e ***puntatori***:



Ogni ***nodo*** ha tre campi: uno per l'informazione associata al nodo, uno per il puntatore al sottoalbero sinistro e uno per il puntatore al sottoalbero destro.

```
typedef int el_type;
typedef struct nodo
          {el_type  value;
           struct nodo *left, *right;
                                    } NODO;
typedef NODO * tree;
```

Vantaggi:

- si ha un'occupazione di memoria efficiente;
- operare modifiche è agevole;
- le procedure di visita sono realizzate in modo diretto (ricorsione).

L'astrazione di dato albero binario:

Operazioni:

root(T)	---> il valore del nodo radice di T;
left(T)	---> sottoalbero sinistro di T;
right(T)	---> sottoalbero destro di T;
empty(T)	---> test-vuoto
cons_tree(e,L,R)	---> costruice l'albero di radice e, sottoalbero sinistro L, e sotto-albero destro R
emptytree	---> costruttore albero vuoto (NULL)
preorder	visita in preordine
inorder	visita in ordine
postorder	visita in postordine

```
/* INTERFACE  tree.h */
typedef char el_type;
typedef struct nodo
       {el_type  value;
        struct nodo *left, *right;
       } NODO;
typedef NODO* tree;
typedef int  boolean;

/* operatori esportabili */

boolean empty(tree t);
tree  emptytree(void);
el_type root(tree t);
tree  left(tree t);
tree  right(tree t);
tree  cons_tree(el_type e,tree l,tree r);

/* procedure di visita - stampa */
void preorder(tree t);
void inorder(tree t);
void postorder(tree t);

tree ord_ins(el_type e, tree t);
boolean member(el_type e, tree t);
boolean member_ord(el_type e, tree t);
```



```
/* IMPLEMENTATION - tree.c */
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"

void showel(el_type C)
{ printf("%c",C); }

/* operatori primitivi   */
boolean empty(tree t)
/* test di albero vuoto */
{ return (t==NULL); }

tree  emptytree(void)
/* inizializza un albero vuoto */
{ return NULL; }

el_type root (tree t)
/* restituisce la radice dell'albero t */
{   if (empty(t)) abort();
    else return(t->value);
}

tree  left (tree t)
/* restituisce il sottoalbero sinistro */
{   if (empty(t)) return(NULL);
    else return(t->left);
}

tree  right (tree t)
/* restituisce il sottoalbero destro */
{   if (empty(t)) return(NULL);
    else return(t->right);
}
```

```
tree  cons_tree(el_type e, tree l, tree r)
/* costruisce un albero che ha nella
 radice e; per sottoalberi sinistro e
 destro l ed r rispettivamente    */
{tree t;
    t = (NODO *) malloc(sizeof(NODO));
    t-> value = e;
    t-> left = l;
    t-> right = r;
    return (t);
}
```


Procedure di visita:

La stampa in ordine simmetrico degli elementi di un albero **t** si ottiene con una procedura di visita simmetrica o in ordine (realizzata dalla funzione **inorder**):

```
void inorder(tree t)
{if(empty(t))
   {inorder(left(t);
    showel(root(t));
    inorder(right(t));
   }
}
```

Commenti:

La funzione, ricorsiva, riceve il puntatore **t** al nodo radice di un albero binario e visita dapprima il sottoalbero sinistro (visualizzando così tutti gli elementi minori di quello memorizzato nel nodo radice di **t**), poi visualizza il valore memorizzato nella radice di **t** e successivamente visita il sottoalbero destro (visualizzando così tutti gli elementi maggiori o uguali a quello memorizzato nel nodo radice di **t**). Le visite dei due sottoalberi sono ottenute con chiamate ricorsive alla funzione **inorder**, passando come argomenti delle chiamate rispettivamente il riferimento al sottoalbero sinistro di **t** e il riferimento al sottoalbero destro di **t**.

La visita in ordine anticipato è realizzata dalla funzione **preorder**:

```
void preorder(tree t)
/* visita in preordine */
{
if (! empty(t))
       {  showel( root(t) );
          preorder (left(t));
          preorder (right(t));
       };
};
```

Commenti:

Tale funzione, anch'essa ricorsiva, riceve il puntatore **t** al nodo radice di un albero binario e visualizza dapprima il valore memorizzato nella radice di **t**, successivamente visita il sottoalbero sinistro, poi visita il sottoalbero destro. Le visite dei due sottoalberi sono ottenute con chiamate ricorsive alla funzione **preorder**, passando come argomenti delle chiamate rispettivamente il riferimento al sottoalbero sinistro di **t** e il riferimento al sottoalbero destro di **t**.

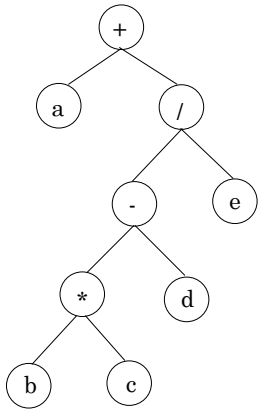
La visita in ordine ritardato è realizzata dalla funzione **postorder**:

```
void postorder(tree t)
/* visita in postordine */
{
if (! empty(t))
       {  postorder (left(t));
          postorder (right(t));
          showel( root(t) );
       };
};
```

Commenti:

Tale funzione, anch'essa ricorsiva, riceve il puntatore **t** al nodo radice di un albero binario e visita dapprima il sottoalbero sinistro, poi il sottoalbero destro per visualizzare solo alla fine il valore memorizzato nella radice di **t**. Le visite dei due sottoalberi sono ottenute con chiamate ricorsive alla funzione **postorder**, passando come argomenti delle chiamate rispettivamente il riferimento al sottoalbero sinistro di **t** e il riferimento al sottoalbero destro di **t**.

Esempio di costruzione di un albero binario:




```
#include <stdio.h>
#include "tree.h"

void main (void)
{ tree    t1,t2;
  el_type  ch;
   t1=cons_tree('b',emptytree,emptytree);
   t2=cons_tree('c',emptytree,emptytree);
   t1=cons_tree('*',t1,t2);
   t2=cons_tree('d',emptytree,emptytree);
   t1=cons_tree('-',t1,t2);
   t2=cons_tree('e',emptytree,emptytree);
   t2=cons_tree('/',t1,t2);
   t1=cons_tree('a',emptytree,emptytree);
   t1=cons_tree('+',t1,t2);

   printf("\nStampa in ordine\n");
   inorder(t1);
}
```


Ricerca in un albero binario:

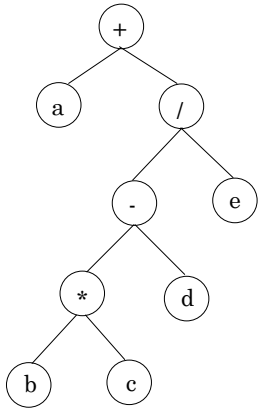
```
boolean member(el_type e, tree t)
{
if (empty(t)) return(0);
else
    if (e==root(t)) return(1);
    else   return(member(e, left(t)) ||
           member(e, right(t)) );
}
```

Commenti:

La ricerca restituisce il valore falso (**0**) se l'albero **t** è vuoto. Se l'albero non è vuoto e l'elemento cercato **e** è uguale al valore della radice dell'albero, la ricerca termina restituendo il valore vero (**1**). Altrimenti, il valore restituito è determinato, attraverso due chiamate ricorsive, dal risultato della ricerca dell'elemento **e** nel sottoalbero sinistro e nel sottoalbero destro di **t**. Basta che l'elemento **e** appartenga a uno dei due sottoalberi perché il risultato restituito sia vero (operatore logico or, **||**).

Esempio:

```
if member('*',t1) printf("trovato");
```



Nel caso peggiore esploro tutto l'albero (costo O(N), dove N è il numero dei nodi)

☞Come ottimizare la ricerca?

Alberi binari di ricerca

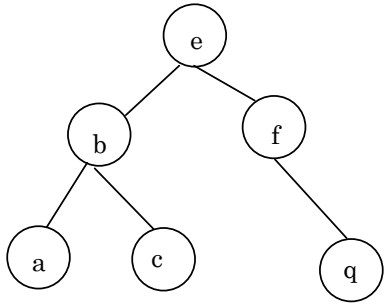
Sono alberi binari utilizzati per memorizzare grosse quantità di dati su cui si esegue spesso ***un'operazione di ricerca*** di un dato.

In un ***albero binario di ricerca***, ogni nodo N ha la seguente proprietà:

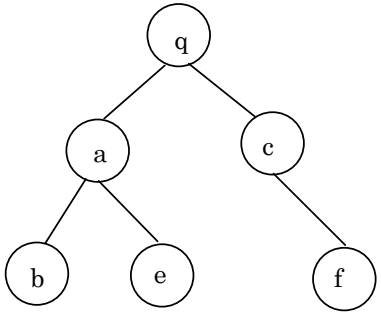
- tutti i nodi del sottoalbero sinistro di N hanno un valore minore o uguale a quello di N e
- tutti i nodi del sottoalbero destro di N hanno un valore maggiore di quello di N.

Esempio:

Albero binario di ricerca:

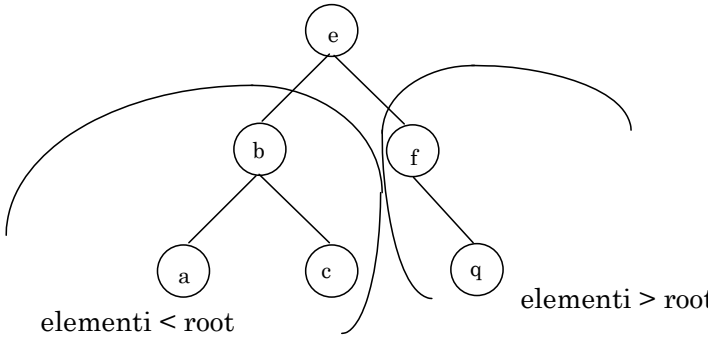


Albero binario:



La procedura **`ord_ins`** della libreria **`tree`** serve per creare alberi binari di ricerca (eventualmente con duplicazione di elementi).

Inserimento in un albero binario di ricerca:




```
tree ord_ins(el_type e, tree t)
/* albero binario di ricerca con
   duplicazioni                      */
{
if (empty(t))    /* inserimento */
   return(cons_tree(e,emptytree(),
                    emptytree()) );
else
   if (e <= root(t))
   return(cons_tree(root(t),
                    ord_ins(e,left(t)),
                    right(t)) );
   else
      if (e > root(t))
         return(cons_tree(root(t),
                    left(t),
                    ord_ins(e,right(t))) );
};
```


Commenti:

La procedura **ord_ins** realizza l'inserimento di un elemento `e` in un albero binario di ricerca **t**.
Se l'albero **t** è vuoto, la funzione **ord_ins** costruisce con la funzione primitiva **cons_tree** un nuovo albero, avente un unico nodo che memorizza l'elemento **e**, e che ha sottoalbero destro e sinistro vuoti.
Altrimenti, se **t** non è vuoto, la funzione si chiama ricorsivamente per costruire un nuovo albero (sempre tramite la funzione **cons_tree**).
Se il valore da inserire è inferiore a quello memorizzato nella radice di **t**, l'albero costruito ha nella radice lo stesso valore della radice di **t** (**root(t)**), lo stesso sottoalbero destro (`right(t)`) e come sottoalbero sinistro quello ottenuto inserendo `e` nel sottoalbero sinistro di **t** (con **ord_ins(e,left(t))**).
Se il valore da inserire è maggiore o uguale a quello memorizzato nella radice di **t**, invece, l'albero costruito ha nella radice lo stesso valore della radice di **t** (**root(t)**), lo stesso sottoalbero sinistro (**left(t)**) e come sottoalbero destro quello ottenuto inserendo **e** nel sottoalbero destro di **t** (con **ord_ins(e,right(t))**).
Si noti che questo tipo di inserimento è molto costoso in termini di occupazione di memoria poiché porta a una duplicazione della struttura dati (in pratica, ad ogni chiamata della funzione **cons_tree** si ricopia e ricrea un nodo dell'albero originario).

Una versione meno costosa in termini di memoria è presentata nell'esercizio 10.1.

Ricerca binaria in un albero binario di ricerca:

```
boolean member_ord(el_type e, tree t)
{
if (empty(t)) return(0);
else
    if (e==root(t)) return(1);
    else
        if (e < root(t))
            return(member_ord(e,left(t)) );
        else return(member_ord(e,right(t)) );
}
```

Commenti:

In questo caso la ricerca restituisce il valore falso (**0**) se l'albero **t** è vuoto. Se l'albero non è vuoto e l'elemento cercato **e** è uguale al valore della radice dell'albero, la ricerca termina restituendo il valore vero (**1**). Altrimenti, se il valore cercato è minore di quello della radice si prosegue la ricerca nel sotto-albero sinistro. Se invece è maggiore di quello della radice, si prosegue la ricerca nel sotto-alberodestro.

Ad ogni passo si dimezza il problema, eliminando metà dei nodi (più efficiente).

Il numero di confronti è (nel caso peggiore) proporzionale alla profondità dell'albero.

È perciò importante mantenere l'albero ***bilanciato*** (tutti i cmmini dalla radice alle foglie hanno più o meno la stessa altezza).

Gli alberi binari di ricerca sono spesso utilizzati per realizzare ***indici*** su file (si veda l'esercizio 10.2).

0 Esercizi d'esame:

Esercizio 10.1

Sia data dal dispositivo di standard input una sequenza di interi, forniti in ordine casuale. La sequenza ha lunghezza non nota a priori ed è terminata dal valore zero.
Realizzare un programma che acquisisca la sequenza, la memorizzi in un albero binario di ricerca e la visualizzi ordinata sul dispositivo d'uscita standard tramite visita in ordine.
Si visualizzi poi il contenuto dell'albero applicando la visita in preordine e in postordine. Infine, letto un intero a terminale, si verifichi se compare o meno nell'albero.

Soluzione:

La struttura del programma principale è la seguente:

```
/* file main1.c */
#include<stdio.h>
typedef int el_type;
typedef enum {false,true} boolean;
typedef struct nodo
      {el_type value;
       struct nodo *left,*right;} NODO;
typedef NODO *tree;

/* prototipi funzioni ausiliarie */
void showel(el_type);
boolean isless (el_type e1, el_type e2);
boolean isequal (el_type e1, el_type e2);

tree ordins(el_type,tree);
void inorder(tree);
void preorder(tree);
void postorder(tree);
boolean member(el_type, tree);
```

```
void main (void)
{tree T=NULL;
 el_type n;
 do
  {printf("\n Dammi un valore intero:\t");
   scanf("%d",&n);
   T=ordins(n,T);
  }  while (n!=0);

 printf("\n visita in ordine:\n");
 inorder(T);
 printf("\n visita in preordine:\n");
 preorder(T);
 printf("\n visita in postordine:\n");
 postorder(T);

 printf("\n Numero da cercare: \t");
 scanf("%d", &n);
 if (member(n,T))
      {printf("Trovato! %d \n", n);}
 else {printf("Non trovato! %d \n", n) ;}
}
```

```
/* funzioni ausiliarie */
void showel(el_type e)
{ printf("%d \n", e);}

boolean isless (el_type e1, el_type e2)
{ if (e1<e2) return true;
  else return false;}

boolean isequal (el_type e1, el_type e2)
{ if (e1==e2) return true;
  else return false;}
```


Il codice della funzione **ordins** che realizza l'inserimento di un elemento nell'albero è dato da:

```
tree ordins(el_type e,tree t)
{if (t==NULL)
     {t=(tree)malloc(sizeof(NODO));
      t->value=e;
      t->left=NULL;
      t->right=NULL;
      return t; }
 else
     {if(isless(e,t->value))
         {t->left=ordins(e,t->left);
          return t;}
      else
         {t->right=ordins(e,t->right);
          return t;}  }
}
```

Commenti:

L'inserimento avviene attraverso l'allocazione esplicita di memoria con la primitiva **malloc** della libreria standard. Se l'albero è vuoto, si crea il suo primo nodo, altrimenti il problema dell'inserimento di un valore intero **e** nell'albero non vuoto referenziato dal puntatore **t** è risolto chiamando ricorsivamente la funzione **ordins**. L'inserimento avviene nel sottoalbero sinistro di **t** (**t->left**) se il valore di **e** è inferiore a quello memorizzato nel nodo radice di **t** (**t->value**), oppure nel sottoalbero destro di **t** (**t->right**) se il valore di **e** è maggiore o uguale a quello memorizzato nel nodo radice di **t** (**t->value**).

Le procedure di visita:

La stampa in ordine degli elementi letti da ingresso e memorizzati nell'albero **T** si ottiene visitando **T** con una procedura di visita simmetrica (realizzata dalla funzione **inorder**):

```
void inorder(tree t)
{if(t!=NULL)
   {inorder(t->left);
    showel(t->value);
    inorder(t->right);
   }
}
```

La visita in ordine anticipato è realizzata dalla funzione **preorder**:

```
void preorder(tree t)
{if(t!=NULL)
   {showel(t->value);
    preorder(t->left);
    preorder(t->right);
   }
}
```

La visita in ordine ritardato è realizzata dalla funzione **postorder**:

```
void postorder(tree t)
{if(t!=NULL)
   {postorder(t->left);
    postorder(t->right);
    showel(t->value);
   }
}
```

La funzione **member_ord** realizza la ricerca binaria dell'elemento **e** nell'albero **t**:


```
boolean member_ord(el_type e, tree t)
{if(t==NULL) return false;
 else
   {if (isequal(e,t->value))
              return true;
    else if (isless(e,t->value))
             return member(e,t->left))
          else return member(e,t->right));
   }
}
```

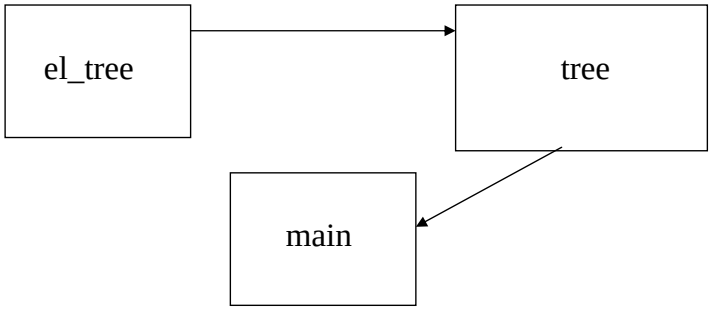
Esercizio 10.2

Un file di testo, PAROLE.TXT, contiene un elenco di parole, una per linea. Ciascuna parola è lunga al più trenta caratteri e può comparire più volte nel file. Un secondo file di testo, ESCLUSE.TXT, contiene parole in copia unica (una per linea) che occorre cancellare dal file precedente.
Si definisca un programma che:

- per operare tale cancellazione crea in memoria centrale un albero binario di ricerca T contenente le parole del file PAROLE.TXT che *non compaiono* nel file ESCLUSE.TXT (la stessa parola può essere inserita più volte nell'albero);
- riscriva il contenuto dell'albero T sul file PAROLE.TXT, in modo che il file sia ordinato;
- letta una parola a terminale, accedendo all'albero T, indichi a terminale se la parola compare nell'albero.

Soluzione:

Strutturiamo l'applicazione è in una serie di moduli.



Il primo modulo (costituito dai file **el_tree.h** e **el_tree.c**) realizza il tipo degli elementi, **el_type**, come stringa di trenta caratteri.

L'interfaccia del modulo è rappresentata dal file **el_tree.h**:

```
/* file el_tree.h */
#include <stdio.h>

typedef struct{char parola[30];} el_type;
typedef int boolean;

void shownode(el_type e);
boolean minore(el_type e1, el_type e2);
boolean uguale(el_type e1,el_type e2);
```

Sul dominio del tipo **el_type** è definita una relazione di ordine totale che è utilizzata per creare alberi binari di ricerca.

La relazione di ordinamento sugli elementi di tipo **el_type** e, in particolare, le operazioni di confronto di uguaglianza e disuguaglianza (minore) sono realizzate dalle funzioni **uguale** e **minore**; la visualizzazione degli elementi è realizzata dalla funzione **shownode**.

```
/* file el_tree.c */
#include "el_tree.h"
#include <string.h>
void shownode(el_type e)
{ printf("%s\t", e.parola); }

boolean minore(el_type e1, el_type e2)
{ return (strcmp(e1.parola,e2.parola)<0||
          !strcmp(e1.parola,e2.parola));  }

boolean uguale(el_type e1,el_type e2)
{ return (!strcmp(e1.parola,e2.parola));   }
```

Il secondo modulo (realizzato dai file **tree.h** e **tree.c**) definisce il tipo **tree** come albero binario di elementi di tipo **el_type**. L'interfaccia del modulo è costituita dal file **tree.h**:

```
/* file tree.h */
#include "el_tree.h"
typedef struct nodo
{   el_type value;
          struct nodo *left, *right;
       } NODO;

typedef NODO *tree;

/* operazioni primitive esportate */

tree emptytree();
boolean empty(tree t);
el_type root(tree t);
tree left(tree t);
tree right(tree t);
tree cons_tree(el_type e, tree l, tree r);

/* operazioni non primitive esportate */
void preorder(tree t);
void postorder(tree t);
void inorder(tree t);
tree ord_ins(el_type e, tree t);
boolean member(el_type e, tree t);
boolean member_ord(el_type e, tree t);
```

La realizzazione delle operazioni del modulo è contenuta nel file **tree.c**:


```
/* file tree.c */
#include <stdlib.h>
#include "tree.h"

tree emptytree()
{   return NULL; }

/* operazioni primitive esportate */
boolean empty(tree t)
{ return (t==NULL);
}

el_type root(tree t)
{ if (empty(t))
       exit(0);
  else return (t->value);
 }

tree left(tree t)
{   if (empty(t))
       exit(0);
    else return (t->left);
}

tree right(tree t)
{   if (empty(t))
       exit(0);
    else return (t->right);
}

tree cons_tree(el_type e, tree l, tree r)
{ tree t;
  t=(NODO *)malloc(sizeof(NODO));
  t->value=e;
  t->left=l;
  t->right=r;
  return t;
}

/* operazioni non primitive esportate */
```

```
void postorder(tree t)
{   if (!empty(t))
       {   postrder(left(t));
           prostder(right(t));
           shownode(root(t));
       }
    return;
}

void preorder(tree t)
{   if (!empty(t))
       {   shownode(root(t));
           preorder(left(t));
           preorder(right(t));

       }
    return;
}

void inorder(tree t)
{   if (!empty(t))
       {   inorder(left(t));
           shownode(root(t));
           inorder(right(t));

       }
    return;
}
```



```
tree ord_ins(el_type e, tree t)
{   if (empty(t))
       return cons_tree(e, emptytree(),
                                emptytree());
    else
       if (minore(e,root(t)))
          return cons_tree(root(t),
           ord_ins(e,left(t)), right(t));
       else
          return cons_tree(root(t), left(t),
     ord_ins(e,right(t)));
}

boolean member(el_type e, tree t)
{   if (empty(t))
       return 0;
    else
       if (uguale(e, root(t)))
          return 1;
       else
          return( member(e,left(t))||
 member(e, right(t)));
}

boolean member_ord(el_type e, tree t)
{   if (empty(t))
       return 0;
    else
       if (uguale(e, root(t)))
          return 1;
       else
          if (minore(e,root(t)))
             return member_ord(e, left(t));
          else
             return member_ord(e, right(t));

}
```

Il programma principale

Il programma principale, contenuto nel file `main.c`, si preoccupa di creare il file e successivamente leggerne il contenuto creando l'albero binario di ricerca `T` inizialmente vuoto. L'algoritmo identifica le parole del primo file che non compaiono nel secondo file con una scansione del file `PAROLE.TXT` e, per ciascuna parola letta, nella ricerca di tale parola letta nel file `ESCLUSE.TXT`, ripartendo dall'inizio ogni volta. L'algoritmo è realizzato da due cicli `while` innestati. La scansione del primo file è realizzata dal ciclo `while` più esterno; letta una parola (assegnata al campo `el.parola`), si effettua la ricerca nel secondo file con il ciclo `while` innestato. La ricerca s'interrompe quando si raggiunge la fine del secondo file o non appena si trova la parola cercata. Solo nel caso in cui si esca dal ciclo innestato con il valore della variabile `trovato` che è `false` (parola non trovata), si inserisce l'elemento `el` nell'albero `T`, inizialmente vuoto, con la procedura `ord_ins`.
Terminata la scansione del primo file, si esce dal ciclo `while` più esterno, si chiudono i due file e si stampa l'elenco di parole secondo l'ordine lessicografico con una visita in ordine simmetrico dell'albero `T`. La stampa a terminale si ottiene tramite la funzione `inorder` e la scrittura del file `PAROLE.TXT` tramite la funzione `inorder_write`.
Infine, letta una parola a terminale, tramite la funzione `member_ord` verifica se tale parola compare nell'albero.

Il codice del programma principale, contenuto nel file `main.c` è il seguente:


```
/* file main.c */
#include"tree.h"

int cont=0;
void inorder_write(tree t,FILE *f1);

main()
{   tree T;
    FILE *f1,*f2;
    el_type e1,e2;
    char s[30];
    int trovato;

    /*domanda a) */
    T=NULL;
    f1=fopen ("parole.txt","r");
    f2=fopen ("escluse.txt","r");

    fscanf(f1,"%s", &e1.parola);
    while (!feof(f1))
    {   trovato=0;
        rewind(f2);
        fscanf(f2, "%s", &e2.parola);
        while(!feof(f2) && !trovato)
            {
            if (uguale(e1,e2))
                trovato=1;
            else fscanf(f2,"%s",&e2.parola);
            }
        if (!trovato)
            T=ord_ins(e1,T);
        fscanf(f1,"%s", &e1.parola);
        }
    fclose(f1);
    fclose(f2);
```

```
    /* domanda b) */
    inorder(T); /* non richiesto: stampa
dell'albero*/
    f1=fopen("parole.txt", "w");
    inorder_write(T,f1);
    fclose(f1);

    /*domanda c) */
    printf("\nDammi una parola: ");
    scanf("%s", &e1.parola);
    if (member_ord(e1,T))
          printf("la parola %s è presente\n",
                  e1.parola);
   else
     printf("la parola %s non è presente\n",
                  e1.parola);

}

void inorder_write(tree t,FILE *f1)
{
    if (!empty(t))
        {inorder_write(left(t),f1);
         fprintf(f1,"%s\n",root(t).parola);
         inorder_write(right(t),f1);
        }
    return;
}
```

Anche in questo caso, per compilare il programma realizzato e produrne un eseguibile, essendo l'applicazione organizzata su più file sorgente, occorre aprire un progetto (per esempio `main.gpr`) e inserirvi i nomi dei file sorgente (ovvero `main.c`, `tree.c` ed el_tree.c).

ESERCIZIO 10.3

Un file binario ad accesso diretto (PERSONE.DAT) contiene un elenco di informazioni su persone (cognome, nome, data di nascita, città di residenza). Il cognome di ciascuna persona costituisce la ***chiave unica*** di tale elenco.
Per effettuare ricerche su questo elenco, tutti i cognomi con la stessa lettera iniziale devono essere memorizzati in un ***albero binario di ricerca*** mantenendo, per ciascun cognome il numero di record corrispondente. Per memorizzare il riferimento alla corrispondente struttura ad albero, per ciascuna lettera dell'alfabeto, si utilizza un vettore.
Si definisca un programma C che:

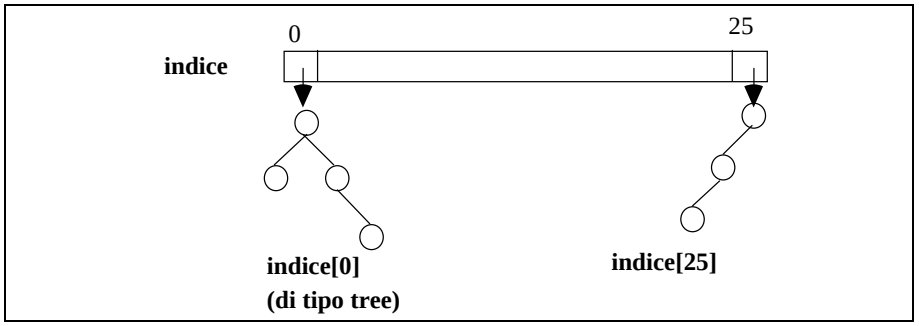
a) costruisca le strutture dati in memoria principale a partire dal file PERSONE.DAT;

b) visualizzi, dato un cognome, l'elenco di cognomi con la stessa iniziale che lo precedono secondo l'usuale ordinamento alfabetico;

c) calcoli l'altezza di ciascun albero albero.

Strutture dati:

Le strutture dati risultano del tipo:



dove indice è un vettore di 26 componenti.

Ciascuna componente del vettore è un puntatore ad albero binario (di tipo **`tree`**).

Ciascun nodo di un albero binario di tipo tree memorizza un cognome ed il numero d'ordine del record corrispondente nel file PERSONE.DAT. Poiché il cognome è la chiave di tale file, non esistono persone con lo stesso cognome.

```
typedef   struct
             {char cognome[15];
              long  pos;} el_type;
typedef struct nodo
          {el_type  value;
           struct nodo *left, *right;
                     } NODO;
typedef NODO * tree;
```


Struttura del programma:

a) Ciclo di lettura dei record che costituiscono PERSONE.DAT. Il file puo` essere letto come un file sequenziale. Ad ogni lettura si incrementa un variabile che "conta" il numero di record letti (oppure ftell). Ciascun cognome letto (ed il corrispondente numero) viene inserito nel corrispondente albero binario di ricerca (indice[R.cognome[0]-'A']);

b) Per stampare tutti i cognomi che precedono uno dato e che iniziano con la stessa lettera si puo` visitare l'albero secondo la visita simmetrica e stampare solo gli elementi minori di quello dato (ovvero tutte le radici dei sottoalberi esplorati, e tutti i sotto-alberi sinistri).

c) L'altezza dell'albero puo` essere calcolata tramite una procedura ricorsiva. Se l'albero e' vuoto l'altezza e' nulla, altrimenti e' 1+ il massimo tra l'altezza del sottoalbero sinistro e l'altezza del sottoalbero destro.

File utilizzati:

main.c	programma principale
el.h	file header del tipo di dato astratto el_type (strutture)
el.c	implementazione delle operazioni definite per il tipo el_type
trees.h	file header del tipo di dato astratto albero binario
trees.c	implementazione della libreria albero binario

Nel ***project file*** vanno menzionati tutti i file con estensione ***.c*** utilizzati.

```
/* MAIN FILE */
#include "trees.h"
#include <stdio.h>

void main ()
{ tree  indice[26];
  int i, pos;
  FILE *f;
  record_type  *REC;
  el_type  EL;
  char COGNOME[15];
  REC= (record_type *)
       malloc(sizeof(record_type));
  f = fopen("PERSONE.DAT", "rb");
/* CICLO DI SCANSIONE DEL FILE */
  pos=0;
  while (!(feof(f)))
  { fread(REC, sizeof(record_type), 1, f);
    for (i=0;i<15;i++)
          EL.cognome[i]=(*REC).cognome[i];
    EL.pos=pos++;
/* INSERIMENTO NELL'ALBERO */
  printf("Inserimento albero %d-esimo\n",
          (EL.cognome[0]-'A'));
  indice[(EL.cognome[0]-'A')] =
  ord_ins(EL,indice[(EL.cognome[0]-'A')]);
  inorder(indice[(EL.cognome[0]-'A')]);
  }

fclose(f);
puts("Chiusura del file");
gets("COGNOME");
```



```
/* VISUALIZZAZIONE ELEMENTI MINORI */
  puts("Inserisci un cognome: ");
  gets(COGNOME);

  stampa_minori(indice[(COGNOME[0]-
'A')],COGNOME);


/* CALCOLO ALTEZZA DEGLI ALBERI */

for (i=0;i<25;i++)
   { printf("Altezza dell'albero %d-esimo %d
      \n",i,height(indice[i]));}

}
```

```
/* ELEMENT TYPE - file el.h*/
typedef   struct
        {  char nome[15];
           char cognome[15];
           char datan[7];
           char citta[15]; } record_type;
typedef   struct  {char cognome[15];
                   long  pos;} el_type;
typedef   enum {false,true}  bool;
/* operatori esportabili */
bool isequal(el_type, el_type);
bool isless(el_type, el_type);
void showel (el_type);
```

```
/* ELEMENT TYPE - file el.c*/
#include "el.h"
#include <stdio.h>
#include <string.h>

bool isequal(el_type e1, el_type e2)
/* due elementi sono uguali se hanno lo
stesso cognome */
{
 if (strcmp(e1.cognome,e2.cognome)==0)
       return(true);
 else  return false;
}

bool isless(el_type e1, el_type e2)
{
  if (strcmp(e1.cognome,e2.cognome) <0)
       return true;
  else    return(false);
}


void showel (el_type e)
{ puts(e.cognome);
  printf("\t%d\n",e.pos);
}
```



```
/* TREE INTERFACE  - file trees.h*/
#include "el.h"

typedef struct nodo
          {el_type  value;
           struct nodo *left, *right;
                     } NODO;
typedef NODO * tree;

/* operatori esportabili */
bool empty(tree t);
tree  emptytree(void);
el_type root(tree t);
tree  left(tree t);
tree  right(tree t);
tree  cons_tree(el_type e, tree l, tree r);
tree ord_ins(el_type e, tree t);
void inorder(tree t);
void stampa_minori(tree t, char cognome[]);
int height (tree t);
```

```
/* IMPLEMENTATION - file trees.c */
#include "trees.h"
#include <stdlib.h>
bool empty(tree t)
/* test di albero vuoto */
{ return (t==NULL); }

tree  emptytree(void)
/* inizializza un albero vuoto */
{ return NULL; }

el_type root(tree t)
/* restituisce la radice dell'albero t */
{ if (!empty(t))
     return (t->value); }

tree  left (tree t)
/* restituisce il sottoalbero sinistro */
{   if (empty(t)) return(NULL);
    else return(t->left);   }

tree  right (tree t)
/* restituisce il sottoalbero destro */
{   if (empty(t)) return(NULL);
    else return(t->right);  }

tree  cons_tree(el_type e, tree l, tree r)
/* costruisce un albero che ha nella  radice e; per
  sottoalberi sinistro e destro l ed r          */
{tree t;
    t = (NODO *) malloc(sizeof(NODO));
    t-> value = e;
    t-> left = l;
    t-> right = r;
    return t;     }
```



```
tree ord_ins(el_type e, tree t)
/* albero binario di ricerca senza duplicazioni */
{if (empty(t))   /* inserimento */
    {printf("albero vuoto");
    return(cons_tree(e,emptytree(),
                          emptytree())); }
 else
   if (isless(e,root(t)))
      {printf("inserimento nell'albero
                di un elemento minore della
                radice");
    return(cons_tree(root(t),
           ord_ins(e,left(t)),
           right(t)          ) );}
   else
    return(cons_tree( root(t),
            left(t),
            ord_ins(e,right(t)) ) );
}

void stampa_minori(tree t, char cognome[])
{el_type EL;
  if (!empty(t))
     {EL=root(t);
      if (strcmp(EL.cognome, cognome)==0)
          inorder(left(t)) ;
      else
        if (strcmp(EL.cognome, cognome)<0)
         {inorder(left(t));
          showel(EL);
          stampa_minori(right(t),cognome);}
        else
         {stampa_minori(left(t),cognome);}
     }
}
```

```
void inorder(tree t)
{
  if (!empty(t))
     { inorder(left(t));
       showel(root(t));
       inorder(right(t));
     }
}


int height (tree t)
{if (empty(t))
   return 0;
else
      return (1+ max(height(left(t)),
                     height(right(t)) ) );
}
```


ESERCIZIO 10.5

Es. n° 2 – Scritto del 29 Marzo 2006

Due file binari contengono dati relativi a progetti di opere edili (EDI.DAT) e i progettisti delle opere edili (PROG.DAT). Il primo file contiene record che riportano codice progetto (intero), email del progettista (stringa di 20 caratteri), costo previsto (intero). Il secondo file contiene il nome e cognome del progettista (stringhe di 20 caratteri), il suo email (stringa di 20 caratteri) e il codice del progetto (intero) che ha eseguito.

Si scriva un programma C **strutturato in (almeno) tre funzioni** dedicate rispettivamente:

a) a creare un albero binario di ricerca T, ordinato in base al codice del progetto, che contiene in ciascun nodo: codice progetto, costo previsto , nome e cognome del progettista;
b) leggere a terminale il cognome e nome di una persona e accedendo all'albero T determini quanti progetti sono assegnati a tale persona e qual e' la somma dei loro costi previsti;
c) accedendo a T stampi a video l'elenco ordinato dei codici progetto.

È possibile utilizzare ***librerie C*** (ad esempio per le stringhe).
Nel caso si strutturi a moduli l'applicazione qualunque ***libreria utente*** va riportata nello svolgimento.

```
/* ELEMENT TYPE - file el.h*/
typedef   struct
        { int codice;
          char email[20];
          int costo; } record_type1;

typedef   struct
        { char nome[20];
          char cognome[20];
          char email[20];
          int codice; } record_type2;

typedef   struct
        { int codice;
          int costo;
          char nome[20];
          char cognome[20];} el_type;

typedef   enum {false,true}  bool;
/* operatori esportabili */
bool isequal(el_type, el_type);
bool isless(el_type, el_type);
void showel (el_type);
```



```
/* ELEMENT TYPE - file el.c*/
#include "el.h"
#include <stdio.h>
#include <string.h>

bool isequal(el_type e1, el_type e2)
/* due elementi sono uguali se hanno lo
stesso cognome e nome */
{
 if ( (strcmp(e1.cognome,e2.cognome)==0) &&
      (strcmp(e1.nome,e2.nome)==0) )
         return(true);
 else    return false;
}

bool isless(el_type e1, el_type e2)
{/* in base al codice */
  return (e1.codice <= e2.codice);
}


void showel (el_type e)
{  printf("%d\n",e.codice); }
```

```
/* MAIN FILE */
#include "trees.h"
#include <stdio.h>

tree crea_albero(FILE * f, FILE *g, tree t);
int conta(el_type e, tree t);

void main ()
{ tree  T=NULL;
  int i, pos;
  FILE *f;
  FILE *g;
  el_type  EL;
  f = fopen("EDI.DAT", "rb");
  g = fopen("PROG.DAT", "rb");

/* PRIMA DOMANDA */
  T=crea_albero(f,g,T);
  close(f);
  close(g);

/* SECONDA DOMANDA */
  scanf("%s\n", &(EL.cognome));
  scanf("%s\n", &(EL.nome));
  printf("Num progetti assegnati: %d",
            conta(E2,T));

/* TERZA DOMANDA */
  inorder(T);

}
```



```
tree crea_albero(FILE * f, FILE *g, tree t)
{ record_type1  E1;
  record_type2  E2;
  bool trovato=false;
  el_type  EL;

  while (!(feof(f)))
  { fread(&E1, sizeof(record_type1), 1, f);
    reset(g);
    fread(&E2, sizeof(record_type2),1,g);
    while ((!feof(g)) && (!trovato))
    { if (E1.codice==E2.codice)
            trovato=true; }
      else
       fread(&E2, sizeof(record_type2),1,g);
     }
    if (trovato)
      { EL.codice=E1.codice;
        EL.costo= E2.costo;
        strcpy(EL.nome,E2.nome);
        strcpy(EL.cognome,E2.cognome);
        t=ord_ins(EL,t);
      }
  }
  return t;
}


int conta(el_type e, tree t)
{if (t==NULL) return 0;
 else
  if (isequal(e,root(t))
   return
(1+conta(e,left(t))+conta(e,right(t)));
  else
  return conta(e,left(t))+conta(e,right(t));
}
```

```
/* TREE INTERFACE  - file trees.h*/
#include "el.h"

typedef struct nodo
          {el_type  value;
           struct nodo *left, *right;
                    } NODO;
typedef NODO * tree;

/* operatori esportabili */
bool empty(tree t);
tree  emptytree(void);
el_type root(tree t);
tree  left(tree t);
tree  right(tree t);
tree  cons_tree(el_type e, tree l, tree r);
tree ord_ins(el_type e, tree t);
void inorder(tree t);
```



```
/* IMPLEMENTATION - file trees.c */
#include "trees.h"
#include <stdlib.h>
bool empty(tree t)
/* test di albero vuoto */
{ return (t==NULL); }

tree  emptytree(void)
/* inizializza un albero vuoto */
{ return NULL; }

el_type root(tree t)
/* restituisce la radice dell'albero t */
{ if (!empty(t))
     return (t->value); }

tree  left (tree t)
/* restituisce il sottoalbero sinistro */
{   if (empty(t)) return(NULL);
    else return(t->left);   }

tree  right (tree t)
/* restituisce il sottoalbero destro */
{   if (empty(t)) return(NULL);
    else return(t->right);  }

tree  cons_tree(el_type e, tree l, tree r)
/* costruisce un albero che ha nella  radice e; per
  sottoalberi sinistro e destro l ed r          */
{tree t;
    t = (NODO *) malloc(sizeof(NODO));
    t-> value = e;
    t-> left = l;
    t-> right = r;
    return t;     }
```

```
tree ord_ins(el_type e, tree t)
/* albero binario di ricerca senza duplicazioni */
{if (empty(t))   /* inserimento */
    {printf("albero vuoto");
    return(cons_tree(e,emptytree(),
                          emptytree())); }
 else
   if (isless(e,root(t)))
      {printf("inserimento nell'albero
                di un elemento minore della
                radice");
    return(cons_tree(root(t),
           ord_ins(e,left(t)),
           right(t)          ) );}
   else
    return(cons_tree( root(t),
             left(t),
             ord_ins(e,right(t)) ) );
}


void inorder(tree t)
{
  if (!empty(t))
     { inorder(left(t));
       showel(root(t));
       inorder(right(t));
     }
}
```


Inserimento in un albero binario di ricerca: versione iterativa

Il codice della funzione **ordins** è dato in questo caso da:

```
tree ordins(el_type e,tree t)
{tree p, root=t;
 if (root==NULL)
     {root=(tree)malloc(sizeof(NODO));
      root->value=e;
      root->left=NULL;
      root->right=NULL;}
 else
    {while (t!=NULL)
        if (isless(e,t->value))
          {p=t;
           t=t->left;}
        else
          {p=t;
           t=t->right;}
        /* punta al padre di t */
    }
     if ((isless(e,p->value))
       { p->left=(tree)malloc(sizeof(NODO));
         p=p->left;
         p->value=e;
         p->left=NULL;
         p->right=NULL;}
     else
       {p->right=(tree)malloc(sizeof(NODO));
         p=p->right;
         p->value=e;
         p->left=NULL;
         p->right=NULL;}
 return root;  }
```

Inserimento in un albero binario di ricerca:

```
tree ordins(el_type e,tree t)
/* ricorsiva senza copia di struttura */
{if (t==NULL)
     {t=(tree)malloc(sizeof(NODO));
      t->value=e;
      t->left=NULL;
      t->right=NULL;
      return t; }
 else
     {if(isless(e,t->value))
         {t->left=ordins(e,t->left);
          return t;}
      else
         {t->right=ordins(e,t->right);
          return t;}  }
}
```

```
tree ord_ins(el_type e, tree t)
/* ricorsiva con copia della struttura dati*/
{if (empty(t))     /* inserimento */
    return(cons_tree(e,emptytree(),emptytree()) );
 else
   if (e <= root(t))
    return(cons_tree(root(t),ord_ins(e,left(t)),
                                      right(t)) );
    else
        if (e > root(t))
            return(cons_tree(root(t),left(t),
                           ord_ins(e,right(t))) );
};
```


Ricerca binaria in un albero binario di ricerca: versione iterativa

```
boolean member (el_type e, tree t)
{ while (t!=NULL)
    {if (isless(e,t->value)) t=t->left;
     else
       if (isequal(e,t->value)) return 1;
       else t=t->right;       }
  return 0;
}
```