Verificare se un albero binario è bilanciato

Definizione: Un albero è bilanciato nel Numero dei Nodi, brevemente n-bilanciato , quando, per ogni sottoalbero t radicato in un suo nodo, il numero dei nodi del sottoalbero sinistro di t, meno il numero dei nodi del sottoalbero destro di t è in valore assoluto al più 1.

E' evidente che un albero t,
con sottoalberi t1 e t2, rispettivamente con n1 e n2 nodi,
è n-bilanciato sse

t1 e t2 sono n-bilanciati

e

$$|n1-n2| \leq 1.$$

Verificare se un albero binario è bilanciato: soluzione inefficiente

```
int nBilIneff(TreePtr  tPtr)
/* verifica se l'albero T e' n-bilanciato, VERSIONE INEFFICIENTE.
  *postc: restuisce 1 se l'albero è bilanciato, 0 altrimenti */
 { int nnr, nnl;
   if (!tPtr) return 1;
  else
       if (nBilIneff(tPtr->lPtr))
              if (nBilIneff(tPtr->rPtr))
                    { nnl = numNodi(tPtr->lPtr);
                      nnr = numNodi(tPtr->rPtr);
                      if (abs(nnr - nnl)<=1) return 1;
                      }
  return 0;
}
```

Verificare se un albero binario è bilanciato: unica scansione dell'albero

```
int nBilAus(TreePtr  tPtr, int *ris)
/* verifica il bilanciamento di tPtr con un unica scansione dell'albero.
* prec: ris != NULL
postc: se l'albero è bilanciato restituisce il numero dei nodi nell'albero
* e ris è diverso da zero, altrimenti ris è zero e il valore restituito non è
significativo*/
 { int nnl, nnr;
  if (!tPtr) return 0;
         else
         if (ris)
              {nnl = nBilAus(tPtr->lPtr, ris);
               if (ris)  nnr = nBilAus(tPtr->rPtr,ris);
              if (abs(nnl-nnr)>1)  *ris = 0;}
 return (nnl+nnr+1); }

int nBil(TreePtr  tPtr)
 /* verifica il bilanciamento nel numero di nodi di tPtr,
  *postc: restuisce 1 se l'albero è bilanciato, 0 altrimenti */
{ int ris = 1;
nBilAus(tPtr,&ris);
return ris;}
```

Verificare se un albero binario è bilanciato: nessun parametro aggiuntivo

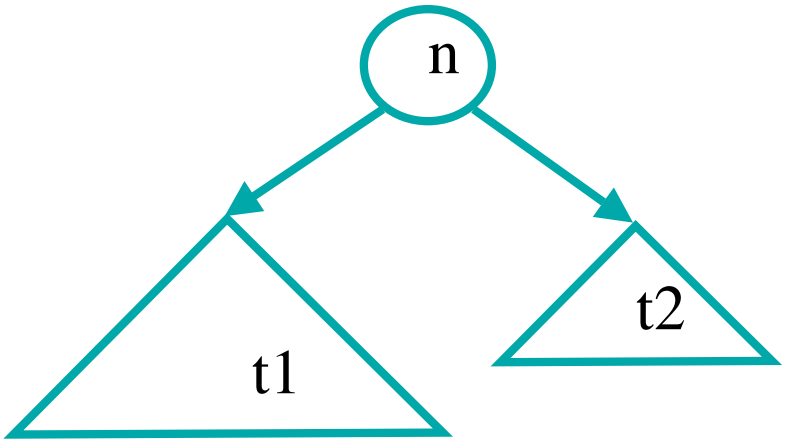
```
int nBilEffAus(TreePtr  tPtr)
/* verifica il bilanciamento di tPtr con un unica scansione dell'albero.
* postc: restituisce -1 se l'albero in input è sbilanciato, il numero dei nodi
nell'albero altrimenti. */
 { int nnl, nnr;
  if (!tPtr) return 0;
        else
        { nnl = nBilEffAus(tPtr->lPtr);
          if (nnl >= 0)
              { nnr = nBilEffAus(tPtr->rPtr);
                if (nnr>=0 && abs(nnl-nnr)<=1)
                    return (nnl+nnr+1);
               }
          }
   return -1;
 }

int nBilEff(TreePtr  tPtr)
 /* verifica il bilanciamento nel numero di nodi di tPtr,
  *postc: restuisce 1 se l'albero è bilanciato, 0 altrimenti */
 { if (nBilEffAus(tPtr) == -1) return 0; else return 1;}
```

Verificare se un albero binario è un albero binario di ricerca

```
int ABR(TreePtr tPtr)
/* verifica se tPtr è un albero binario di ricerca, di interi,  senza ripetizioni
*postc: restituisce 1 se tPtr e' un albero binario di ricerca, di interi,
*senza ripetizioni, e 0 altrimenti */
{int m = INT_MIN;
return ABRaus(tPtr,&m);}

int ABRaus (TreePtr tPtr, int *prec)
/* verifica se tPtr è un albero binario di ricerca, di interi,  senza ripetizioni
*prec: prec != NULL.
*postc: restituisce 1 se tPtr e' un albero binario di ricerca, di interi,
*senza ripetizioni, e 0 altrimenti */
{int ris;
if (!tPtr) return 1;
if ( ris = (ABRaus(tPtr->left,prec)  && *prec < tPtr->elem))
/* in prec ho sempre l'ultimo nodo visitato, che poiche'
la visita e' inorder deve essere minore di quello correntemente visitato */
{*prec = tPtr->elem;
ris = ABRaus(tPtr->right,prec);}
 return ris;}
```

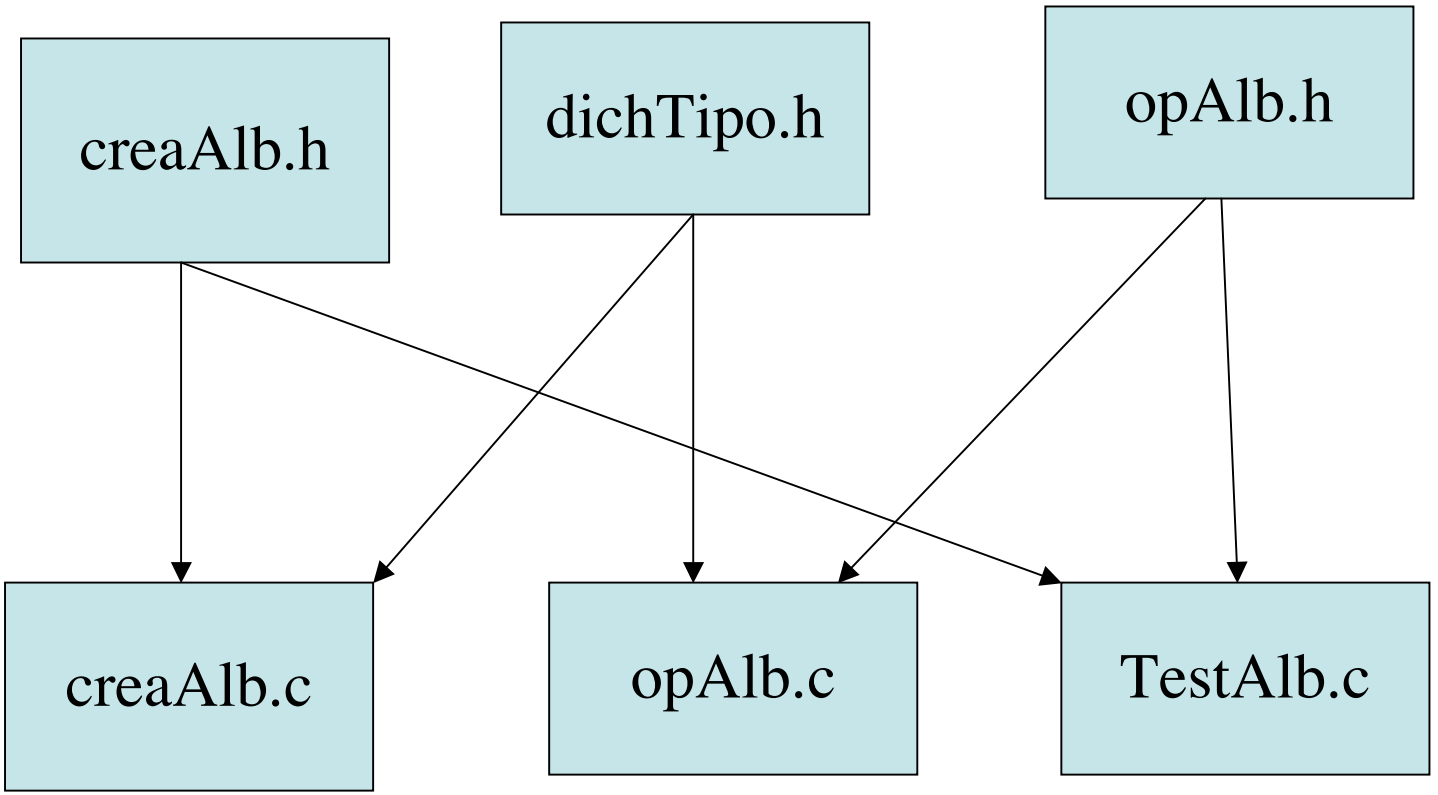


Nella chiamata su n prec = max(t1), poi prec diventa n
e sarà confrontato con min(t2)

```
 int cammInt(TreePtr tPtr)
/* Calcola la lunghezza del cammino interno, cioè
 la somma dei livelli di tutti i nodi non foglia.
 postc:restituisce la lunghezza del cammino interno, -1 se l'albero è vuoto */
{int liv = 0;
if (tPtr ) return cammIntAus(tPtr,liv);
else return -1;}

int cammIntAus(TreePtr tPtr, int liv)
 /* prec: tPtr != NULL
* postc:   restituisce la lunghezza del cammino interno di un albero  non nullo
* calcola in liv il livello di tPtr  */
{int cammD,cammS;
if (!tPtr -> left  && !tPtr -> right  ) return 0;
if (tPtr -> left) cammS = cammIntAus(tPtr -> left,liv+1) ; else cammS = 0;
if (tPtr -> right)  cammD = cammIntAus(tPtr -> right,liv+1); else cammD = 0;
return cammS+cammD+liv;}
```

Schema inclusioni tra i file del programma per il testing di funzioni su alberi di interi, che restituiscono un intero



Testing di funzioni su alberi di interi, che restituiscono un intero

Nome File: dichTipo.h

```
struct Tree{
        struct Tree *right;
        int elem;
        struct Tree *left;
};
```

Testing

Nome File: creaAlb.h

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <limits.h>

#ifndef TIPO                    /*così evitiamo la multipla definizione di un tipo*/
#define TIPO
typedef struct Tree TREE;
typedef TREE * TreePtr;
#endif

TreePtr creaAlb(FILE* fIn);
/* prec: fIn != NULL
postc: restituisce un albero binario non vuoto leggendo i dati dal file in input*/
TreePtr addElem(TreePtr t, TreePtr nuovo );
/*inserisce l' elemento nuovo nell'ABR t
prec: nuovo !=NULL;
postc: restituisce t con l'aggiunta di nuovo se non presente, altr. t non
modificato*/
TreePtr creaABR(FILE* fIn);
  /*prec: fIn != NULL
  Post: restituisce un puntatore alla radice di un ABR
 non vuoto i cui dati provengono da un file*/
```

Testing di funzioni su alberi di interi, che restituiscono un intero

Nome File: creaAlb.c

```
#include "dichTipo.h"
#include "creaAlb.h"

 TreePtr creaAlb(FILE* fln)
/*Post: restituisce un un albero binario non vuoto i cui dati provengono da un file*/
{TreePtr t;
int j,val;
/*legge un intero  dal file di input  che indica quali figli deve avere: j=-1 per il solo figlio
sinistro,j=0 per nessun figlio, j=1 per il solo figlio destro, j=2 per entrambi*/
            fscanf(fln,"%d%d",&j,&val);
            t= (TreePtr) calloc(1,sizeof(TREE));
            if (t==NULL)
                        printf("memoria non allocata\n");
            else
                        {t->elem=val;
                        switch(j)  /* se j==0 non si deve fare nulla */
                                    {case 1: t->right = creaAlb(fln); break;
                                     case -1: t->left = creaAlb(fln); break;
                                     case 2: t->left = creaAlb(fln);
                                                 t->right = creaAlb(fln); break;
                                    case 0: break;
                                    default: printf("valore di j = %d non corretto!\n",j);break; }
                        }
return t; }
```

```
TreePtr addElem(TreePtr t, TreePtr nuovo )
/*aggiunge un elemento in un ABR di interi
prec: nuovo !=NULL;
postc: restituisce t con l'aggiunta di nuovo se non presente, t non modificato altrimenti*/
 {assert(nuovo);
...
 }

TreePtr creaABR(FILE* fln)
/*Post: restituisce un puntatore alla radice di un albero binario di ricerca. di interi,
 non vuoto i cui dati provengono da un file, uno 0 indica la fine dei dati di un albero*/
{TreePtr t=NULL,nuovo;
int val;
/*legge un intero  dal file di input */
fscanf(fln,"%d",&val);
while (val !=0)
            {nuovo = (TreePtr) calloc(1,sizeof(TREE));
            if (nuovo==NULL)
                        printf("memoria non allocata\n");
            else
             {nuovo->elem=val;
               t = addElem(t,nuovo);}  /*precondizione rispettata*/
              fscanf(fln,"%d",&val);
             }
                return t;
}
```

Testing di funzioni su alberi di interi, che restituiscono un intero

Nome File: opAlb.h

```
#include "dichTipo.h"
#include "creaAlb.h"

int nBil(TreePtr  tPtr);
/* verifica il bilanciamento nel numero di nodi di tPtr,
  *postc: restuisce 1 se l'albero è bilanciato, 0 altrimenti */

  int ABR(TreePtr tPtr);
/*postc: restituisce 1 se tPtr e' un albero binario di ricerca, di interi,
*senza ripetizioni, e 0 altrimenti */

int cammInt(TreePtr tPtr);
/* Calcola la lunghezza del cammino interno, cioè
 la somma dei livelli di tutti i nodi non foglia.
 postc:restituisce la lunghezza del cammino interno, -1 se l'albero è vuoto */
```

Testing di funzioni su alberi di interi, che restituiscono un intero

Nome File: opAlb.c

```
#include "opAlb.h"

static int nBilEffAus(TreePtr  tPtr);
/*postc: se l'albero è bilanciato restituisce il numero dei nodi nell'albero
*, altrimenti -1*/

static int ABRaus (TreePtr tPtr, int *prec);
/* verifica se tPtr è un albero binario di ricerca, di interi,  senza ripetizioni
*postc: restituisce 1 se tPtr e' un albero binario di ricerca, di interi,
*senza ripetizioni, e 0 altrimenti */

static int cammIntAus(TreePtr tPtr, int liv);
 /* prec: tPtr != NULL
* postc:   restituisce la lunghezza del cammino interno di un albero  non nullo
* calcola in liv il livello di tPtr  */
```

qui il codice delle funzioni nBil,ABR e CammInt

Nome File: TestAlb.c

```
#include "creaAlb.h"
#include "opAlb.h"
#define NUM_TEST  10  /*in alternativa si può leggere da file*/
#define NOME_F_IN  "fInputAB"
#define NOME_F_OUT  "fOutputCInt"
#define costrAlb( fPtr)   creaAlb(fPtr)
#define funzDaTest(t)   cammInt(t)

int main()
{int i,n,vettErr[NUM_TEST]={0},risT=0;
TreePtr t = NULL;
FILE* fIn,*fOut;
if ( (( fIn= fopen(NOME_F_IN,"r")) == NULL) ||  \
((fOut = fopen(NOME_F_OUT,"r")) == NULL))
   {fprintf(stderr, "il file %s o %s non può essere aperto \n ", \
     NOME_F_IN,NOME_F_OUT);exit(1);}
for (n=0;n<NUM_TEST;n++)
          {if (n>0) t = costrAlb(fIn); /*il primo test è su t == NULL*/
           fscanf(fOut,"%d",&i);
           if (  funzDaTest(t) != i) {risT++; vettErr[i]=i;}}
if (risT == 0) printf("tutto O.K.!\n");
else for (i=0;i<NUM_TEST;i++)
           if (vettErr[i] != 0) printf("il caso i= %d ha dato errore\n",i);
fclose(fIn);fclose(fOut);return 0;}
```

0 4
-1 4
0 8
1 4
0 5
2 6
0 7
0 3
-1 1
-1 4
0 8
1 9
1 6
0 2
-1 4
1 8
0 6
1 4
-1 3
0 6
2 8
2 15
0 10
0 5
2 4
0 5
0 1

Nome File: fInputAB

Nome File: fOutputCInt

-1
0
0
0
0
1
1
1
1
2

Testing di funzioni su alberi di interi con un secondo par. input, che restituiscono un intero

Nome File: opAlb.c

```
#include "dichTipo.h"
#include "creaAlb.h"

int estraeInf(TreePtr t);
/*postc:restituisce l'info del nodo , se t è NULL dà 0*/

TreePtr stringToNode(TreePtr tPtr, char *s);
/*postc:restituisce il puntatore al nodo raggiunto dalla radice
seguendo il cammino descritto da s*/

int  foglieLivK(TreePtr tPtr, int k);
/*restituisce il numero di foglie di livello k.
* prec: k>=0;
* postc: restituisce 0 se l'albero e' vuoto, altrimenti il numero
delle foglie il cui cammino padri-figli dalla radice e' lungo k */
```

```
int  foglieLivK(TreePtr tPtr, int k)
/*restituisce il numero di foglie la cui distanza dalla radice e' k.
* prec: k>=0;
* postc: restituisce 0 se l'albero e' vuoto, altrimenti il numero
delle foglie il cui cammino padri-figli dalla radice e' lungo k */
  {if (!tPtr ) return 0;
   if ((!tPtr->left && !tPtr->right && k==0 )  return 1; /*è una foglia!*/
    return foglieLivK(tPtr -> left,k-1) + foglieLivK(tPtr -> right,k-1) ;
   }

int estraeInf(TreePtr t)
/*postc: restituisce l'info del nodo , se t è NULL dà 0*/
{if (t) return t ->elem; else return 0;}
```

Nome File: TestAlb2Str.c

```
#include "creaAlb.h"
#include "opAlb.h"
#define NUM_TEST1  10
#define NUM_TEST2  7
#define NOME_F_IN1  "fInputAB”
#define NOME_F_IN2  "fInputStr"
#define NOME_F_OUT  "fOutputSTN”
#define TIPO_IN2  char*
#define MAX_SIZE 4
#define costrAlb( fPtr)   creaAlb(fPtr)
#define funzDaTest(t) estraeInf(stringToNode(t,x))
#define caricaIN2(x)  caricaStr(x)

char** caricaStr(FILE * fIn2);

char** caricaStr(FILE * fIn2)
{int i;
char** vett= malloc(NUM_TEST2 * sizeof(char*));
vett[0] = "";
for (i=1;i< NUM_TEST2; i++)
{vett[i] = malloc(MAX_SIZE*sizeof(char));
fscanf(fIn2, "%s", vett[i] );}
return vett;}
```

Nome File: TestAlb2.c

```
int main()
{int i,m,n,vettErr[NUM_TEST1*NUM_TEST2]={0},risT=0;
TreePtr t = NULL;
TIPO_IN2 * vett;
FILE* fIn1,fIn2*fOut;
if ( ( fIn1= fopen(NOME_F_IN1,"r")) == NULL ||  \
( ( fIn2= fopen(NOME_F_IN2,"r")) == NULL ||  \
(fOut = fopen(NOME_F_OUT,"r") )== NULL  )
   {fprintf(stderr, "il file %s,%s o %s non può essere aperto \n ", \
     NOME_F_IN1, NOME_F_IN2,NOME_F_OUT);exit(1);}
vett = caricaIN2(fIn2); fclose(fIn2);
for (n=0;n<NUM_TEST1;n++)
          {if (n>0) t = costrAlb(fIn); /*il primo test è su t == NULL*/
           fscanf(fOut,"%d",&i);
           for (i=0;i<NUM_TEST2;i++)
                {fscanf(fOut,"%d",&j);
                if (funzDaTest(t,vett[i]);!= i) {risT++; vettErr[n*NUM_TEST2+i]=1;}}
if (risT == 0) printf("tutto O.K.!\n");
else for (i=0;i<NUM_TEST1*NUM_TEST2;i++)
           if (vettErr[i] != 0) printf("il caso i= %d ha dato errore\n",i);
fclose(fIn);fclose(fOut);return 0;}
```

Nome File: TestAlb2Int.c

```
#include "creaAlb.h"
#include "opAlb.h"
#define NUM_TEST1  10
#define NUM_TEST2  3
#define NOME_F_IN1  "fInputAB”
#define NOME_F_IN2  "fInputInt"
#define NOME_F_OUT  "fOutputLF”
#define TIPO_IN2  int
#define costrAlb( fPtr)   creaAlb(fPtr)
#define funzDaTest(t) foglieLivK(t,x)
#define caricaIN2(x)  caricaInt(x)

int* caricaInt(FILE * fIn2);

int* caricaInt(FILE * fIn2)
{int i;
int* vett= malloc(NUM_TEST2 * sizeof(int));
for (i=0;i< NUM_TEST2; i++)
{fscanf(fIn2, "%d",&vett[i]);}
return vett;}
```

stesso main del caso su stringhe