# FONDAMENTI DI INFORMATICA C

FRANCO ZAMBONELLI

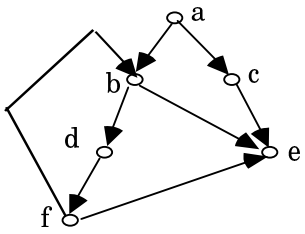
# GLI ALBERI

## STRUTTURE NON LINEARI

Si dicono in generale **GRAFI**
- Insieme di nodi (contengono gli atomi)
- Connessi da relazioni binarie (archi orientati "da" un nodo "a" un altro)

**Grado di ingresso** di un nodo
- numero di archi che entrano nel nodo

**Grado di uscita** di un nodo
- numero di archi che partono dal nodo




La **dimensione** di un grafo è praticamente data dal numero di dimensioni che servono per poterlo "disegnare" senza incroci
- Le molecole del DNA formano un grafo tridimensionale
- Le liste sono casi monodimensionali di grafi (ogni nodo ha un arco di ingresso e uno di uscita)

Le strutture di relazione tra elementi si rappresentano come grafi



## ALBERO BINARIO

La struttura di grafo ad **albero**, bi-dimensionale, presenta numerosissime applicazioni e riveste un ruolo fondamentale nell'informatica.

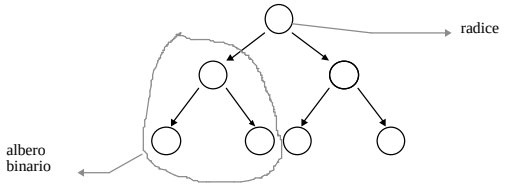
Ci soffermiamo sul cosiddetto *albero binario:* alberi più complessi possono essere facilmente ricavati come generalizzazione degli alberi binari. Sono possibili due definizioni teoriche di albero:

*Definizione 1*

Un albero è un insieme finito di nodi e archi orientati. Ogni arco collega il nodo *padre* ad un nodo *figlio.*
- Ogni nodo ha esattamente un padre (grado di ingresso + 1)
- Ogni nodo ha al più due figli (grado di uscita <= 2).
- Il nodo *radice* non ha padre (grado di ingresso = 0)

*Definizione Alternativa (a carattere ricorsivo)*

Albero binario è un insieme finito di nodi e può essere:
1. insieme vuoto oppure
2. nodo radice + due alberi binari disgiunti, detti rispettivamente *sottoalbero sinistro* e *sottoalbero destro*




**Terminologia relativa agli alberi**

- *livello di un nodo:* distanza dalla radice, espressa come numero di archi di cui è composto il cammino dalla radice al nodo; la radice è a livello 0;
- *profondità (o altezza) dell'albero:* massimo livello dei nodi dell'albero;
- *foglia:* nodo senza figli;
- *predecessore:* relazione strutturale, ottenibile applicando la proprietà transitiva alla relazione padre-figlio.

*Predecessore* è una relazione d'ordine *parziale*, cioè dati due nodi dell'albero o uno è predecessore dell'altro, in quanto esiste un percorso di archi dall'uno all'altro, oppure non c'è relazione.

Si può anche definire una relazione *successore,* inversa di predecessore.

Il massimo numero di nodi a livello $i$ $(i \geq 0)$ è $2^i$ , come si può verificare per induzione:
- $2^0=1$,
- se il massimo numero di nodi a livello $i$ è $2^i$, poichè ogni nodo al livello $i$ può avere al più due figli, al livello $i+1$ il massimo numero di nodi è $2*2^i=2^{i+1}$.
- Il massimo numero complessivo di nodi in un albero di profondità $k$ è $2^0 + 2^1 + ... + 2^k = 2^{k+1}-1$.

## Visite di un albero

**Visitare un grafo**, in genere significa:
- percorrere tutti i suoi nodi una volta e una sola
- o piú, in generale, visitare un insieme specifico di nodi…

Si visita un albero per *visualizzare, elaborare, modificare* il contenuto informativo dei nodi.

La visita di un albero può seguire uno delle seguenti modalità base:
- *preordine*: la visita della radice è seguita dalla visita dei sottoalberi
- *postordine*: la visita dei sottoalberi è seguita dalla visita della radice
- *ordine centrale (simmetrica)*: la visita della radice è intermedia alle visite dei sottoalberi;

La definizione ricorsiva di albero suggerisce la realizzazione delle varie operazioni con **programmazione ricorsiva**: le posizioni dei figli di un nodo possono essere trattate come posizioni di radici di alberi.

```
void PreOrdine(Posiz N, AlbBin T){
  if <albero non vuoto>
   {
    Visita(N);                          /* A */
    PreOrdine(<figlio sinistro>, T):    /* B */
    PreOrdine(<figlio destro>, T);      /* C */
   }
} /* end PreOrdine */
```

Le altre modalità di visita si ottengono semplicemente modificando l'ordine delle operazioni nel corpo della procedura:
- B,C,A per il post-ordine e
- B,A,C per l'ordine centrale.

## Alberi Binari di Ricerca (BST)

La struttura ad albero binario si presta alla gestione di insiemi di dati su cui è definita una *relazione d'ordine lineare*.
- alberi binari con dati ordinati sono detti *Alberi Binari di Ricerca (BST = Binary Search Trees)*
- *U*so albero per ricerche con tecnica dicotomica.

Si suppone che siano disponibili due funzioni `Minore` e `Uguale` in grado di verificare la relazione d'ordine e l'uguaglianza fra due atomi.

Si suppone inoltre di non voler memorizzare nel BST due atomi uguali.

**Nella pratica,** l'albero binario di ricerca si presta alla realizzazione della struttura astratta *dizionario*,
- informazione memorizzata è partizionata in due componenti: *chiave* e *informazione associata*.
- relazione d'ordine e operazioni di confronto soltanto sulla base della chiave.



## Operazioni in Alberi Binari di Ricerca

Ordinamento a carattere ricorsivo
- l'atomo nella radice di un BST partiziona gli atomi dei nodi in due sottoinsiemi:
- quelli minori stanno nei nodi del sottoalbero sinistro, quelli maggiori nei nodi del sottoalbero destro.
- Lo stesso vale per tutti i sottoalberi

Anche nei BST si limitano le operazioni possibili, poiché è necessario che ogni inserimento e cancellazione rispetti il criterio di ordinamento detto.

| *Descrizione* | *intestazione delle funzioni in C* |
|---|---|
| Esegue una visualizzazione completa degli atomi di B secondo il criterio di ordinamento | `void BstVisitaOrd (Bst B)` |
| recupera l'atomo contenuto in un nodo che risulta *uguale* ad A | `int BstRecupera(Atomo *A, Bst B)` |

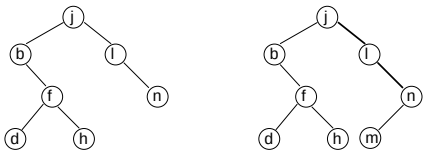
E' possibile realizzare l'albero binario rappresentando con i puntatori le relazioni d'ordine:
- ogni nodo è interessato dalle due relazioni dei *più* grandi e *più piccoli*
- *i*n mancanza di successori da uno dei due lati, il rispettivo puntatore avrà valore `nil`.

```
/* OPERAZIONI DI BST */

/* Bst.h*/
#define BstChiaveEsistente 1
      /*In inser. per duplicati */
#define BstChiaveAssente 2
      /* In cancellaz. e recupero. */
#define BstOK 0

typedef struct TNodo{
  Atomo Dato;
  struct TNodo  *Sx,*Dx;
} Nodo;

typedef Nodo *Posiz;
typedef Posiz Bst;

extern void BstCrea(Bst *B);
extern int BstInserisci(Bst *B, Atomo A);
extern void VisitaOrdinata(Bst B);
extern int BstCancella(Bst *B, Atomo A);
extern int BstRecupera(Bst B, Atomo *A);

extern int BstStato;
```

**/* IMPLEMENTAZIONE BST */**

```
/* albero binario di ricerca implem. con
puntatori */
#include "InfoBase.h"
#include "Bst.h"
#include <stdlib.h>

int BstStato;
Posiz BstTrova(Bst B, Atomo A);
void CancellaMin(Atomo *Min, Bst *B);

/* BstCrea */
void BstCrea(Bst *B){
  (*B) = NULL;
} /* end BstCrea */
```

**OPERAZIONE DI INSERIMENTO**




Inserimento della chiave *m*

Si sfrutta il criterio di ordinamento per cercare il punto in cui inserire un nuovo elemento con tecnica dicotomica

```
- se il l'albero considerato è vuoto crea un nodo e
  l'albero diventa a un nodo,
  altrimenti se l'oggetto da inserire è uguale alla
  radice
       - rifiuta l'inserimento
  altrimenti se l'oggetto da inserire è < della
  radice
       - inserisce nel sottoalbero sinistro
  altrimenti
       - inserisce nel sottoalbero destro
```

Il nuovo nodo sarà sempre inserito come foglia, quindi con i due sottoalberi vuoti.



**/* impl. operazione inserimento */**

```
/* BstInserisci */
int BstInserisci(Bst *B, Atomo A) {
  BstStato=BstChiaveEsistente;
  if (*B==NULL)
  {             /* Creazione nodo */
    (*B)=(Bst)malloc(sizeof(Nodo));
    (*B)->Dato=A;
    (*B)->Sx=NULL;
    (*B)->Dx=NULL;
    BstStato=BstOK;
  }
  else
    if (Minore(A,(*B)->Dato))
      BstInserisci(&(*B)->Sx,A);
    else
      if (Minore((*B)->Dato,A))
        BstInserisci(&(*B)->Dx,A);
  return BstStato;
} /* end BstInserisci*/
```

**OPERAZIONE DI AGGIORNAMENTO**

L'operazione di aggiornamento richiede di trovare la posizione del nodo con chiave uguale a quella dell'atomo di ingresso.

- si programma una funzione interna che restituisce la posizione del nodo contenente un atomo assegnato
- la funzione ricorsiva `BstTrova` realizza la ricerca dicotomica, come semplice variazione della visita in pre-ordine centrale
- la funzione `BstRecupera` restituisce quindi l'atomo A corrispondente alla chiave cercata.
- in caso di non ritrovamento della chiave, la funzione restituisce lo stato di *chiave assente.*

**AGGIORNAMENTO**

```
/* BstTrova: usato solo internamente */
Posiz BstTrova(Bst B, Atomo A) {
  BstStato=BstOK; /* verra' cambiato valore
                     solo se necessario */
  if (B==NULL){
    BstStato=BstChiaveAssente;
    return NULL;
  }
  else
    if (Uguale(B->Dato,A))
      return (Posiz)B;
    else
      if (Minore(A,B->Dato))
        return (Posiz)BstTrova(B->Sx,A);
      else
        return (Posiz)BstTrova(B->Dx,A);
}  /* end BstTrova */

/* BstRecupera */
int BstRecupera(Bst B, Atomo *A) {
  Posiz P;
  P=BstTrova(B,(*A));
  *A=(P->Dato);
  return BstStato;
} /* end BstRecupera */
```

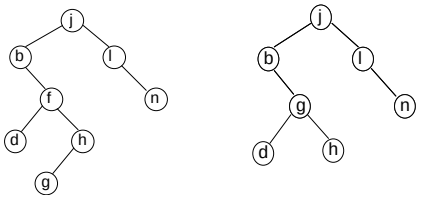
**CANCELLAZIONE DI UN NODO**

La cancellazione di un nodo qualunque N dell'albero pone due problemi:

a) non perdere i sottoalberi di N, reinserendoli nell'albero
b) garantire il mantenimento del criterio di ordinamento dell'albero.

Occorre distinguere diversi casi, a seconda che N sia foglia oppure abbia uno o due sottoalberi non vuoti:

- se N è foglia non è necessaria alcuna azione correttiva per mantenere la condizione di BST;
- se N non è foglia è necessario modificare la struttura dell'albero, tenendo conto che l'atomo minimo del sottoalbero destro di N, che chiameremo MinDx(N), partiziona i valori dell'albero che ha radice in N: i valori del sottoalbero sinistro di N sono tutti minori di MinDx(N)
- se N ha due figli, può essere sostituito dal minimo del suo sottoalbero destro MinDx(N)
- se N ha un solo figlio può essere direttamente sostituito dal figlio






**ALGORITMO DI CANCELLAZIONE**

Richiede:

- l'individuazione la cancellazione del nodo con chiave minima di un sottoalbero, per mantenere la proprietà di ordinamento, secondo il seguente algoritmo
- si mostra solo la cancellazione della radice di un albero; la cancellazione di un nodo qualunque dell'albero sarà preceduta da una fase di ricerca):

```
- se la radice ha due sottoalberi non vuoti
  - cancella il nodo con chiave minima e
 memorizzane il
    contenuto Min
  - sostituisci al contenuto della radice il
 dato Min
  altrimenti
  - se c'è soltanto il sottoalbero sinistro
    - sostituisci alla radice il suo figlio
 sinistro
    altrimenti
    - se c'è soltanto il sottoalbero destro
      - sostituisci alla radice il suo figlio
 destro
      altrimenti
      - cancella la radice, poiché è una
 foglia
```

```
CANCELLAZIONE /* BstCancella */
int BstCancella(Bst *B, Atomo A) {
  Posiz Temp;
  Atomo Min;
  BstStato=BstOK;
  if ((*B)==NULL)
    BstStato = BstChiaveAssente;
  else
    if (Minore(A,(*B)->Dato))
      BstCancella(&(*B)->Sx,A);
    else
      if (Minore((*B)->Dato,A))
        BstCancella(&(*B)->Dx,A);
      else{ /*B punta al nodo che contiene A */
        if (((*B)->Sx!=NULL) && ((*B)-
>Dx!=NULL))
        {     /* Ci sono entrambi i sottoalberi
   */
          CancellaMin(&Min,&(*B)->Dx);
          (*B)->Dato=Min;
        }
```

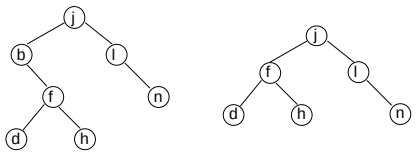
```
/* CONTINUA BstCancella */
     else
       {     /* c'e' al piu' un figlio, Sx o Dx
   */
         Temp=(*B);
         if ((*B)->Sx!=NULL)
           (*B)=(*B)->Sx;  /* solo sottoalb. Sx
   */
         else
           if ((*B)->Dx!=NULL)
             (*B)=(*B)->Dx; /* solo sottoalb. Dx
   */
           else  /* il nodo e' una foglia:     */
             (*B)=NULL;   /* eliminazione
diretta  */
         free(Temp);
       }
     }
  return BstStato;
}  /* end CancellaBst */
```

**CANCELLAZIONE (continua..)**

L'operazione `CancellaMin` per cancellare il nodo con chiave minima dell'albero non ha rilevanza di per se, pertanto non è stata inclusa, fra le operazioni di base.

L'elemento minimo Min di un BST è nel nodo più a sinistra e valgono le seguenti proprietà:

- Min è figlio sinistro del genitore;
- Min può essere:
  - foglia,
  - non foglia: in tal caso avrà soltanto un figlio destro (radice di un sottoalbero);
- tutti i nodi del sottoalbero destro di Min hanno chiavi minori della chiave del genitore di Min;
- all'eliminazione del nodo contenente Min il suo figlio destro può divenire figlio sinistro del genitore del nodo eliminato.






```
/* CancellaMin: uso interno */
void CancellaMin(Atomo *Min, Bst *B){
  Posiz Temp;

  if ((*B)!=NULL)
    if ((*B)->Sx!=NULL)
      CancellaMin(Min,&(*B)->Sx);
    else
    {        /* B punta al nodo con il minimo
*/
      Temp=*B;
      *Min=(*B)->Dato;
      *B=(*B)->Dx;
      free(Temp);
    }
} /* end CancellaMin */
```

Quando sono presenti entrambi i sottoalberi, l'eliminazione fisica (con free) di un nodo è eseguita dalla `CancellaMin`

La `Cancella` sostituisce nel nodo da cancellare effettivamente l'atomo restituito come parametro di uscita da `CancellaMin`. Negli altri casi la `Cancella` esegue l'eliminazione diretta dal nodo.

**VISUALIZZAZIONE ORDINATA BST**

Procedura di visualizzazione completa e ordinata di un BST:

- semplice visita in ordine simmetrico.

```
/* VisitaOrdinata */
void VisitaOrdinata(Bst B) {
  if (B!=NULL)
  {
    VisitaOrdinata(B->Sx);
    Visualizza(B->Dato);
    VisitaOrdinata(B->Dx);
  }
} /* end VisitaOrdinata */
```

## Esempio di utilizzo di alberi binari di ricerca

Programma per la creazione, cancellazione e visualizzazione di un *dizionario*

- informazione è costituita da una `chiave`, in base alla quale si eseguono ricerche e ordinamenti
- e da una `stringa` associata.

Il programma esegue tre cicli:

- inserimento di dati nell'albero,
- ricerca di chiavi nell'albero
- cancellazione di chiavi dall'albero;

Per il confronto fra stringhe si fa ricorso alla *libreria standard* `string.h`.

Header per il trattamento del dato*:*

```
/* InfoBase.h */
#include <string.h>
typedef int boolean;
typedef struct {
  char chiave[16];
  char stringa[48];
} Atomo;

extern void Acquisisci(Atomo *A);
extern void AcquisisciChiave(Atomo *A);
extern void Visualizza(Atomo A);
extern boolean Minore(Atomo A,Atomo B);
extern boolean AtomoNullo(Atomo A);
extern boolean Uguale(Atomo A,Atomo B);
```

**Libreria per il trattamento del dato**:

```
/* InfoBase.c */

#include <stdio.h>
#include "InfoBase.h"


#define Terminatore "."


void Acquisisci(Atomo *A){
  printf("inserire la chiave (max 15
caratteri, ");
  printf(Terminatore);
  printf(" per terminare)      ");
  scanf("%s", &(A->chiave));
  if (!AtomoNullo(*A)){
    printf("inserire l'informazione (max 47
car.) ");
    scanf("%s", &(A->stringa));
  }
}


void AcquisisciChiave(Atomo *A){
  printf("inserire la chiave (max 15
caratteri) ");
  scanf("%s", &(A->chiave));
}

void Visualizza(Atomo A){
  printf("chiave: %s\tinfo:
%s\n",A.chiave,A.stringa);
}
/*   end  Visualizza */
```



```
/* infobase.c  -- Continua */

boolean Minore(Atomo A,Atomo B){
  return (strcmp(A.chiave,B.chiave)<0);
                    /* strcmp presa da
string.h */
}


boolean AtomoNullo(Atomo A){
  return !strcmp(A.chiave,Terminatore);
}

boolean Uguale(Atomo A,Atomo B){
  return !strcmp(A.chiave,B.chiave);
}
```

### ESEMPIO DI UTILIZZO

Infine, il seguente programma è un esempio di utilizzo delle librerie sopra illustrate:

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

#include "InfoBase.h"
#include "Bst.h"

int main(void){
  Atomo A;
  Bst B;
  BstCrea(&B);
  printf("Ciclo di inserimento\n");
  do { /* ripeti finche' atomo non nullo */
    Acquisisci(&A);
    if (!AtomoNullo(A)){
      if(BstInserisci(&B,A)) /*l'unico err.
possibile
              e' quello di "chiave gia'
esistente" */
       printf("Chiave esistente\n");
    } } while (!AtomoNullo(A));
```

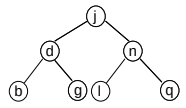
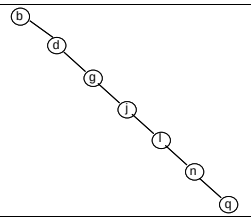
```
  VisitaOrdinata(B);

  printf("Ciclo di ricerca\n");
  do { /* ripeti finche' atomo non nullo */
    AcquisisciChiave(&A);
    if (!AtomoNullo(A)){
      if(BstRecupera(B,&A)){/* l'unico err.
possibile e' quello di "chiave assente" */
        printf("Chiave inesistente\n");
      }
      else{
        Visualizza(A);
      }
    }
  } while (!AtomoNullo(A));

  printf("Ciclo di cancellazione\n");
  do { /* ripeti finche' atomo non nullo */
    AcquisisciChiave(&A);
    if (!AtomoNullo(A)){
      if(BstCancella(&B,A)){/* l'unico err.
possibile
              e' quello di "chiave assente" */
        printf("Chiave inesistente\n");
      }
    else VisitaOrdinata(B); /* in caso di
successo mostra l'albero dopo la cancell. */
    }
  } while (!AtomoNullo(A));
  printf("Fine Lavoro");
  return 0;
}
```



### Complessità delle operazioni su alberi: Visioni Alternative

Sarebbe più interessante esprimere la complessità delle operazioni in funzione di una dimensione più significativa, quale il numero di nodi *n* dell'albero, ma il legame fra *n* e *h* non è noto a priori, poiché:

- la *forma* dell'albero dipende dall'ordine degli inserimenti, come mostrato in figura

| | |
|---|---|
| Albero generato dalla sequenza j,d,b,g,n,l,q;<br><br>quali altre sequenze lo generano? |  |
| Albero generato dalla sequenza b,d,g,j,l,n,q;<br><br>quali altre sequenze generano un albero di profondità massima? |  |

In particolare, il massimo valore di *h* è *n-1*,

- nel caso peggiore la complessità diventa lineare.

Viceversa

- *h* è minima se tutti i percorsi dalla radice alle foglie hanno lunghezza *h-1* o *h*.

Supponendo che tutti i percorsi siano uguali di lunghezza *h*, vale la relazione $n = 2^{h+1}-1$, quindi se *h* è minima *h* è *O(log n)*, e quindi

- le operazioni hanno complessità *O(log n).*

### Complessità delle operazioni su alberi

Tutte le operazioni su alberi, ad eccezione della creazione, vengono eseguite in tempo *O(h)*, dove *h* è la profondità dell'albero.

Prova intuitiva: le procedure viste

- non contengono cicli,
- sono ricorsive ed eseguono sempre una singola *navigazione verso il basso,*
- terminano quando viene incontrata una foglia.

Per effetto del secondo e del terzo punto

- il numero di attivazioni di procedura è sempre uguale alla profondità di una foglia
- l'assenza di cicli interni alle procedure rende la complessità di caso peggiore delle singole chiamate costante rispetto alle dimensioni del problema. N

## Albero bilanciato

Per garantire che le operazioni abbiano una complessità logaritmica esistono due possibilità:

- fidarsi di un comportamento casuale che genera un albero di profondità *quasi* minima,
- intraprendere azioni correttive *durante* la costruzione dell'albero in caso di *sbilanciamento.*

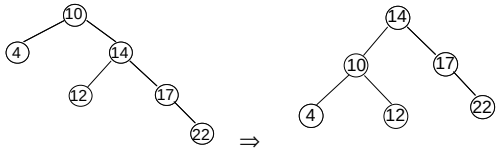
Un albero è *bilanciato* se:

- ad ogni nodo la differenza fra le profondità dei suoi due sottoalberi è al più 1.

La profondità di un albero bilanciato è sempre *O(log n).*

Un albero bilanciato può essere ottenuto per ridefinizione delle operazioni su BST: in questo caso si parla di *AVL-tree* (dai nomi degli inventori Adel'son-Velskii e Landis, 1962).

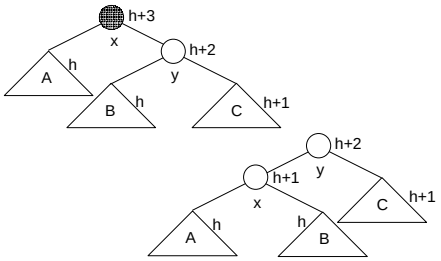
- gli algoritmi di inserimento e cancellazione sono estesi con operazioni di *soccorso,* chiamate per ristabilire il bilanciamento in caso di necessità.
- le operazioni di soccorso si basano sul concetto di *rotazione* di una porzione di albero, come mostrato in figura
- la struttura di un nodo deve venire estesa per contenere l'altezza del sottoalbero di cui il nodo è radice.



**Rotazione**

La figura mostra come variano le altezze dei sottoalberi di un nodo sottoposto a rotazione.

Se lo sbilanciamento di x è dovuto al figlio destro del figlio destro:
- una *rotazione verso sinistra* su x ristabilisce il bilanciamento.

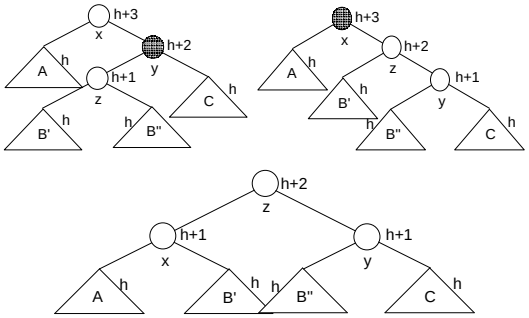



**Doppia Rotazione**

Se invece lo sbilanciamento nel nodo x è dovuto al figlio sinistro del figlio destro si deve fare una doppia rotazione

- una *rotazione verso destra* su y riporta al caso precedente,
- e una *rotazione verso sinistra* su x ristabilisce il bilanciamento.

Vedi figure sotto.




Ogni operazione di rotazione richiede la ristrutturazione dei puntatori e il ricalcolo delle altezze ai nodi.

- modifica le operazioni di inserimento e cancellazione, inserendo le opportune chiamate alle operazioni di soccorso
- dopo una modifica dell'albero si può chiamare una procedura `FissaAltezza` per ricalcolare l'altezza dell'albero
- per valutare se un albero è bilanciato, confrontando le altezze dei suoi figli chiamare una procedura `BilanciaSx` o `BilanciaDx`
- le procedure di bilanciamento dovranno poi chiamare in modo opportuno le procedure di rotazione `RuotaSx, RuotaDx,` che implementano le operazioni illustrate nelle figure