Appunti per il corso di

Algoritmi e Strutture Dati

1° anno - Laurea e Diploma in Informatica

a.a. 1997/98

(autore: prof. Gerardo COSTA)
(docente: prof. Enrico PUPPO)

2ª Puntata

Procedure ricorsive

Tipi di dato

Versione: Maggio '99

Indice

1 Induzione Aritmetica

Induzione, induzione aritmetica, principio di induzione, dimostrazioni per induzione,.....

Cerchiamo di chiarire un po' la faccenda, vedendo: il principio di induzione aritmetica (nella versione "a passi"), poi le dimostrazioni per induzione che si fondano su questo principio, infine l'induzione aritmetica "generalizzata". Più avanti vedremo le definizioni induttive.

L'argomento verrà ripreso nei corsi degli anni successivi dando al tutto un'impostazione uniforme e più rigorosa.

Indichiamo con **Z** l'insieme degli interi e con **N** l'insieme dei naturali.

1.1 Principio di induzione aritmetica "a passi"

I.1. Formulazione tratta dalle dispense di Algebra.

Sia P una *affermazione sui numeri interi* e c un intero. Se

•1 base P è vera per il numero c

•2 passo per ogni intero n, n>c: se P è vera per n-1, allora P è vera per n

allora P è vera per ogni numero intero k = c.

Note.

•3 Spesso, invece di *affermazione sui numeri interi*, si dice *proprietà dei numeri interi* (vedere anche oltre).

•4 Il *passo* è chiamato *passo induttivo*.

•5 Spesso il passo induttivo si formula in modo alternativo:

per ogni n, n = c: se P è vera per n, allora P è vera per n+1

(lo stesso vale, sotto, in I.2 e I.3)

I.2. Formulazione usando funzioni.

Consideriamo una funzione Pf : **Z** → {vero, falso} e sia c un intero. Se:

•6 base Pf(c) = vero

•7 passo per ogni intero n, n>c Pf(n-1) = vero implica Pf(n) = vero

allora Pf(k) = vero, per ogni numero intero k = c.

I.3. Formulazione usando insiemi

Sia Ps un sottinsieme di **Z** e sia c un intero. Se:

•8 base $c \in Ps$

•9 passo per ogni intero n, n > c: $n-1 \in Ps$ implica $n \in Ps$

allora: $k \in Ps$, per ogni numero intero k = c.

I.4. Collegamenti

Il collegamento tra la versione con funzioni e quella con insiemi è semplice:

•10 data Pf si considera l'insieme $Ps = \{ x \in \mathbf{Z} \mid Pf(x) = vero\}$; allora si vede che la versione con Ps è solo una traduzione di quella con Pf;

•11 nell'altro senso, dato Ps basta considerare la funzione Pf t.c. Pf(x)=vero sse $x \in Ps$.

Quindi le formulazioni in I.2 ed I.3 sono equivalenti.

Il collegamento con la prima versione (quella in termini di affermazioni o proprietà) si ha osservando che uno dei modi più semplici di precisare il concetto di "proprietà" è quello di far ricorso ai sottinsiemi (o a funzioni come Pf). Una proprietà è completamente caratterizzata dall'insieme degli oggetti/elementi che godono della proprietà stessa; quindi, ad esempio, la proprietà di essere numero primo equivale ad appartenere all'insieme dei primi.

Quindi I.2 ed I.3 sono solamente un modo di precisare I.1.

Di fatto, tutte e tre le formulazioni sono usate, quindi è utile conoscerle e saper passare da una all'altra.

1.2 Dimostrazioni per induzione aritmetica "a passi"

Abbiamo visto il *principio di induzione* (aritmetica "a passi"). Su questo principio si fondano le *dimostrazioni per induzione* (aritmetica "a passi"), che hanno lo schema seguente.

Voglio dimostrare:

Tesi tutti gli interi $k = c$ hanno la proprietà P

Allora :

•1 base dimostro che c ha la proprietà P;

•2 passo supponendo che n-1 abbia la proprietà P [questa è l'*ipotesi induttiva*]
dimostro che anche n ha la proprietà P.

In base al principio di induzione, questa dimostrazione funziona!

Esempi: altre a quelli visti (o che si vedranno) nei corsi di matematica e di informatica, qui sotto c'è un esempio collegato ai coefficienti binomiali.

1.3 Induzione aritmetica "generalizzata"

Diamo l'enunciato del *principio* utilizzando le funzioni (le altre formulazioni e lo schema di dimostrazione si ricavano per analogia con quanto visto sopra - un utile esercizio).

Consideriamo quindi $Pf : \mathbf{Z} \rightarrow \{vero, falso\}$ ed un intero c.

Se:

- base $Pf(c) = vero$
- passo per ogni $n > c$: ($Pf(i) = vero$ per ogni i, $c = i < n$) implica $Pf(n) = vero$

allora $Pf(k) = vero$, per ogni numero intero $k = c$.

Esempi: un esempio di dimostrazione per induzione generalizzata è la dimostrazione di correttezza dell'algoritmo merge sort, per ordinare un array, che vedremo.

Nota 1.

L'induzione generalizzata estende quella a passi; dal punto di vista delle dimostrazioni: ogni dimostrazione per induzione a passi è anche una dimostrazione per induzione generalizzata; non è vero il contrario.

Nota 2.

E' possibile una formulazione più compatta, che riunisce le due condizioni (base e passo) in una sola:

se: [*] per ogni n = c : (Pf(i) = vero per ogni i, c = i < n) implica Pf(n) = vero

allora Pf(k) = vero, per ogni numero intero k = c.

Si osservi che [*] è quasi uguale al passo, ma con "n = c " al posto di "n > c ".

I casi in cui n>c corrispondono al passo, mentre il caso n=c corrisponde alla base, anche perché, per n=c:

•1 non esistono i t.c. c = i < n; dunque: (Pf(i) = vero per ogni i, c = i < n) è vero

•2 "vero implica bla-bla" equivale a "bla-bla" , che in questo caso è "Pf(c)= vero".

1.4 Coefficienti binomiali

Per ragioni "tipografiche" usiamo la notazione [n : k] invece di quella più familiare "n su k".

[n : k] si definisce per n e k (interi) tali che 0 = k = n.

Una definizione utilizza il fattoriale: [n : k] = n! / (k! * (n-k)!).

Fattoriale.

Se consideriamo n oggetti (distinti), con n>0, e contiamo quante sono le possibili *permutazioni* di tali oggetti (cioè le possibili successioni di lunghezza n di tali oggetti), vediamo che esse sono date dal prodotto n*(n-1)*(n-2)*....*1.

Questo fatto si dimostra facilmente per induzione aritmetica su n; **per esercizio**.

Il numero n*(n-1)*(n-2)*....*1 si indica con n! e si legge “n fattoriale” (o fattoriale di n).

Per convenzione si pone anche 0! = 1.

Ritornando ai coefficienti binomiali, vi sono tuttavia dei problemi nell'utilizzare questa definizione per scrivere un algoritmo che calcola [n : k]; infatti, anche quando il valore di [n : k] è abbastanza piccolo, i termini fattoriali possono avere un valore molto grande, che eccede le dimensioni massime per gli interi. L'ostacolo si può aggirare usando i "float" ma si perde in precisione (vedremo che il valore finale deve essere un intero). Quindi sarebbe necessario scrivere un algoritmo che cerca di semplificare i termini al numeratore e al denominatore, prima di eseguire moltiplicazioni e divisioni.

Non approfondiamo questo punto, ma vediamo un'altra definizione (esempio di *definizione induttiva*, come vedremo meglio in seguito) per i coefficienti binomiali.

(i) [base] [n : 0] = [n : n] = 1

(ii) [passo (con 0 < k < n)] [n : k] = [n-1 : k-1] + [n-1 : k]

Questa definizione si presta ad una semplice realizzazione ricorsiva del calcolo dei coefficienti binomiali. Vedremo, in seguito, che si tratta di un algoritmo molto inefficiente e che conviene piuttosto utilizzare una vecchia idea, il triangolo di Pascal-Tartaglia. Ora pero` ci limitiamo a verificare che la definizione induttiva data è corretta, nel senso che: definisce un (unico) numero intero, per ogni scelta di n e di k (purché 0 = k = n).

La nostra tesi è dunque (per [n : k] come definito da (i) e (ii)):

[T1] per ogni n e k interi, 0 = k = n, [n : k] è un intero = 1.

Questa tesi si può dimostrare per induzione su n (e con c=0).

base: n = 0 (e quindi k=0). Immediato da (i): il valore è 1.

passo: L'ipotesi induttiva è: [n-1 : k] è un intero = 1 (per ogni k t.c. 0 =k = n-1).

Consideriamo [n : k] con 0 =k = n.

Se k=0 oppure k=n, la tesi è immediata (il valore è 1).

Altrimenti: [n : k] = [n-1 : k-1] + [n-1 : k]; ma per ipotesi induttiva, i due coefficienti a destra hanno un valore intero = 1, quindi la loro somma è un intero = 1 (anzi >1).

Osservazione.

La formulazione di [T1] è un po' ambigua, e l'ambiguità si riflette in una certa mancanza di chiarezza nella dimostrazione; il fatto è che l'induzione è su n, ma c'è k sempre tra i piedi....

In realtà la tesi andrebbe scritta cosí:

[T1'] per ogni n intero =0 : (per ogni k intero, 0 = k = n, [n : k] è un intero = 1).

Alternativamente, si può usare una funzione:

CB : **N** → {vero, falso},

CB(n) = vero sse per ogni intero k, 0 = k = n, [n : k] è un intero = 1.

La tesi diventa allora: CB(n) = vero per ogni n di **N**.

A questo punto la dimostrazione dovrebbe risultare più chiara; in particolare, nel **passo**:

•1 l'ipotesi induttiva è CB(n-1) = vero

•2 nel dimostrare che anche CB(n) = vero, si procede per casi:

1) k=0, oppure k=n; qui si conclude direttamente, non serve l'ipotesi induttiva;
2) 0<k<n: qui serve l'ipotesi induttiva, su n-1, e si applica una volta relativamente a k (notare che ci siamo: se 0 < k < n, allora 0 < k = n-1) ed una volta relativamente a k-1 (anche qui ci siamo: se 0 < k < n , allora 0 = k-1 < n).

2

Procedure e funzioni ricorsive

Si chiamano *ricorsive* le procedure o funzioni che, direttamente o indirettamente, richiamano sè stesse; si dice anche che si sta usando la *tecnica della ricorsione*.

C'è uno stretto legame tra ricorsione, induzione e *definizioni induttive*; cercheremo di evidenziarlo man mano che procediamo.

Come al solito presentiamo i concetti tramite esempi.

2.1 Il fattoriale

Se consideriamo n oggetti (distinti), con n>0, e contiamo quante sono le possibili *permutazioni* di tali oggetti (cioè le possibili successioni di lunghezza n di tali oggetti), vediamo che esse sono date dal prodotto n*(n-1)*(n-2)*....*1.

Questo fatto si dimostra facilmente per induzione aritmetica su n.

Il numero n*(n-1)*(n-2)*....*1 si indica con n! e si legge "n fattoriale" (o fattoriale di n).

Per convenzione si pone anche 0! = 1.

Notare che il fattoriale cresce molto rapidamente al crescere di n; ad esempio: 6! = 720, 8! = 40320; 10! = 3628800; del resto $n! >> p^p$, dove p = n div 2.

Dunque abbiamo una funzione fatt : **N** → **N** tale che fatt(n) = n!

E` facile scrivere l'algoritmo per calcolare fatt(n), cioè n!, usando un "for" (**per esercizio**).

Un altro modo di procedere è basato sulla seguente eguaglianza: (n+1)! = (n+1) * n!

Questa uguaglianza può essere utilizzata per dare una definizione induttiva della funzione fattoriale:

[base] fatt(0) = 1

[passo] fatt(n+1) = (n+1) * fatt(n)

Sia in Pascal che in C si può tradurre direttamente questa definizione induttiva in una definizione di *funzione ricorsiva*; vediamolo sotto forma di pseudo-codice:

```
function fatt (n : integer) : integer
    if n < 0 then return (-1)        questo equivale ad un messaggio di errore
    else  if  n=0  then return (1)  else return ( n * fatt (n-1) )
```

Per capire come procede l'esecuzione di una chiamata, supponiamo che in un programma ci sia l'istruzione scrivi(fatt (3)). L'esecuzione procede schematicamente così:

fatt(3) → 3 * fatt(2) → 3 * 2 * fatt(1) → 3*2*1* fatt(0) → 3*2*1*1 → scrivi 6

Il fattoriale ci fornisce un esempio di *ricorsione diretta*: nel corpo della funzione fatt compare una chiamata a fatt. Il punto chiave è che il nome "fatt" è noto all'interno del corpo della funzione fatt e si riferisce alla funzione stessa.

2.2 Coefficienti binomiali

Un altro esempio naturale di funzione ricorsiva ci viene dai coefficienti binomiali.

Ricordiamo la loro definizione induttiva, scrivendo [n : k] invece di "n su k".

- [base] [n : 0] = [n : n] = 1
- [passo] per 0<k<n [n : k] = [n-1 : k-1] + [n-1 : k]

Parlando di induzione aritmetica, abbiamo dimostrato che questa è una buona definizione, nel senso che, per ogni k ed n, tali che 0=k=n, individua un (unico) valore intero (=1).

Scrivere una funzione in Pascal o in C è a questo punto banale; in pseudo-codice:

```
function cobin(n, k : integer) : integer
dichiarazioni:        var temp1, temp2 : integer
istruzioni:
        if (0 = k) and (k = n) then
                if  (k=0)  or  (k=n)   then  return(1)
                else      {    temp1 ← cobin(n-1, k-1)
                               temp2 ← cobin(n-1, k)
                               return (temp1 + temp2)       }
        else  return (-1)       come messaggio di errore
```

Notare che temp1 e temp2 non sono necessari; su poteva scrivere semplicemente:

```
                if  (k=0)  or  (k=n)   then  return(1)
                else       return ( cobin(n-1, k-1) + cobin(n-1, k) )
```

Per concretezza, vediamo un programma in C per calcolare $(a + b)^n$, con a, b reali (razionali) ed n intero =0, letti da tastiera, usando la formula $\sum_{k = 0 \,..\, n} [n : k]\ a^k\ b^{n-k}$

Nota: il programma è corretto, ma è molto stupido (per calcolare $(a + b)^n$, il modo ovvio è sommare a con b e poi calcolare la potenza usando moltiplicazioni successive), quindi dobbiamo considerarlo solo un esercizio.

Inoltre, "corretto", qui non vuol dire che i risultati siano esatti, perché bisogna tener conto degli errori dovuti all'aritmetica dei float nel calcolo delle potenze di a e b. Ad esempio:

con n = 3 a = 1.4 b = 2.6

sulla mia macchina ho ottenuto 63.999992 invece del valore esatto: 64

```
#include <stdio.h>

int cobin (int n, int k) ;   /*  prototipo della funzione che calcola il
                        coefficiente  [n : k]                */
main ( )
{
  int n;  float a, b;
  int k, j ;
  float auxa, auxb, res ;

  scanf ("%d %f %f", &n, &a, &b);

  if ( n < 0 ) printf("errore: esponente negativo");
  else {
       res = 0.0;

       for (k=0; k<=n; k++)    {
           auxa = auxb = 1.0;
           for (j=1; j<=k; j++)   auxa = a * auxa ; /* calcola a^k */
           for (j=1; j<=n-k ; j++)  auxb = b * auxb ; /* calcola b^n-k */
           res = res + ( cobin (n, k) * auxa * auxb );
        }  /* chiude il for su k */

       printf ("\n\n n = %d \n a = %f \n b = %f \n res = %f ", n, a, b, res);

      }  /* chiude il ramo else */
}  /* chiude il main */
```

```
int cobin( int n, int k )
{
  int temp1, temp2;
  if  ( (0 <= k) && (k <= n) )
       if  ( (k==0) || (k==n) )  return(1) ;
       else   {    temp1 = cobin(n-1, k-1);
                     temp2 = cobin(n-1, k);
                     return (temp1 + temp2);
          }
  else
return (-1) ; /* come messaggio di errore */
}
```

Nota. Ci sono due n e due k in quello che abbiamo scritto sopra: la coppia n, k di parametri formali della funzione cobin la coppia n, k di variabili del main. non c'è pericolo di confusione, in base alle regole sulla visibilità dei nomi viste in precedenza.

2.3 Un primo esempio di procedura ricorsiva

Questo esempio serve solo per familiarizzarci con il meccanismo della ricorsione; nessun programmatore userebbe la ricorsione in un caso come questo.

Vogliamo una procedura che:

dato un array di caratteri, sia arca : array [1 .. n] of char,

cambia tutte le 'a' in 'A'

La soluzione standard usa una istruzione for per scorrere l'array da sinistra a destra, ...

Si arriva alla soluzione ricorsiva ragionando in questo modo:

supponiamo di aver già guardato e modificato l'array da 1 a k-1; cosa resta da fare?
controllare arca[k] e poi continuare sul resto.

Nota. Da qui in poi, quando utilizziamo un array come parametro di una procedura, specificheremo la sua dimensione mettendo un identificatore (di solito, e comunque in questo caso, n) come estremo destro dell'intervallo tra parentesi quadre, mentre l'estremo sinistro sarà sempre 1. Tale identificatore potrà essere usato nel codice della procedura per riferirci all'indice dell'ultima cella dell'array. La procedura non potrà comunque mai cambiare il suo valore. Nell'implementazione, la dimensione n dell'array dovrà essere specificata in uno dei modi seguenti:

- **Come costante:** alla vecchia maniera che abbiamo usato fino ad ora; allora n sarà una costante di programma (di solito in questo caso usiamo identificatori con lettere maiuscole, ma non è obbligatorio).
- **Come variabile globale:** n sarà il nome di una variabile dichiarata nello scope più esterno (e mai ridichiarata in uno più interno!). Le differenze rispetto al caso precedente sono che:
  1. il valore di n dovrà essere stabilito da qualche parte nel programma prima di chiamare la procedura;
  2. l'array che si passa come parametro potrà essere allocato dinamicamente nel modo già visto facendo riferimento al valore di n, sempre prima di chiamare la procedura

     oppure

     potrà essere dichiarato in modo statico di una dimensione massima (maggiore o uguale a n), mentre n stabilirà di volta in volta qual'è la porzione di array effettivamente utilizzata.
- **Come parametro della procedura:** è il modo più pulito come stile di programmazione. In questo caso n è un parametro formale, quindi il suo nome è visibile solo dentro la procedura. Bisogna ricordarsi di

- aggiungere un parametro in più nella dichiarazione della procedura
- passare ad ogni chiamata il valore della lunghezza effettiva dell'array che si vuole utilizzare.

Le regole 1 e 2 enunciate per il caso precedente valgono anche qui, ma riferite al valore che viene passato (parametro attuale) invece che ad una variabile globale.

Ci sono casi in cui il valore di n non viene mai usato nel codice della procedura. In questo caso (e solo in questo) è superfluo inserire n come ulteriore parametro. Bisogna comunque essere coerenti (cioè fare la stessa cosa) nella dichiarazione e nelle successive chiamate. Vedremo un esempio con la ricerca binaria.

Chiamiamo a_to_A la procedura per modificare l'array.

Cosa deve fare a_to_A su tutto arca (cioè su arca da 1 a n) ?

guardare arca[1], se è 'a' cambiarlo, e poi .... *fare* a_to_A su arca da 2 a n

Il passo generico, a_to_A su arca da k fino ad n, è :

se k=n allora controlla arca[k]; se è 'a' modificalo; riparti con a_to_A da k+1 a n

altrimenti niente

Dunque:

```
procedura  a_to_A (arca: array [1 .. n] od char,        parametro IN-OUT
                   k : int                 parametro IN)
 if k = n  then    {    if arca[k] = 'a'  then  arca[k] ← 'A'
                        a_to_A (arca, k+1)
                   }
```

Se ora ho un particolare array, mio_arca : array [1 .. n] of char, per modificarlo tutto la chiamata è
a_to_A(mio_arca, 1)

Per non lasciare troppe cose nel vago, scriviamo una versione C di procedura e programma chiamante, usando array di soli 10 elementi. Useremo il metodo di implementazion in cui la dimensione dell'array viene passata come ulteriore parametro della procedura.

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

void a_to_A(char arca [ ], int k, int n);   /*  prototipo  */

main()
{
     int j,max;
     char  * mio_arca;

     scanf(ú%d\nî, &max);
     mio_arca = calloc(max, sizeof(char));
     for(j=0; j < SUP ; j++) scanf("%c\n", &mio_arca[j]);

     a_to_A(mio_arca, 0, max);

     printf("\n OUTPUT: \n");
     for(j=0; j < max ; j++)  printf("%c   ", mio_arca[j]);
}

void a_to_A(char  arca[], int k, int n)
  {
    if  (k < n)
      {  if (arca[k] == 'a')  arca[k] = 'A';
        a_to_A(arca,k+1);
       }
   }
```

Esempio di esecuzione:

Input da tastiera: abcabcabca

Output su video: OUTPUT:
A b c A b c A b c A

2.4 Ricerca binaria

Vediamo ora un esempio più interessante: la ricerca di un elemento in un array ordinato, col metodo della ricerca binaria.

Per concretezza, fissiamo il tipo degli elementi: float. Schematicamente:

Dati: xx : float; aa : array [1 .. n] of float ordinato in modo crescente

Vogliamo: vero se per qualche k : aa[k] = xx

falso altrimenti

Il metodo della ricerca binaria, procede per “bisezione dell’array”:

1. guardo il “punto di mezzo” dell’array; sia y l’elemento che trovo;
2. se x = y allora ho finito, altrimenti:
3. se x < y allora (poiché l’array è ordinato) x può solo essere nella 1a metà dell’array; continuo cercando x nella 1a metà;
   se x > y allora x può solo essere nella 2a metà; continuo cercando x nella 2a metà.

Punto chiave: la parte “continuo cercando x nella 1a (o 2a) metà” corrisponde ad una chiamata ricorsiva. Quindi si definisce una funzione ricorsiva con 4 parametri:

xx (l’elemento da cercare); aa (l’array);

inf , sup : indici che delimitano la parte di aa dove cercare.

La ricerca di un elemento x in un array a (con x ed a di tipo giusto) si ottiene poi con la chiamata ricbin(x, a, 1, n).

Il "punto di mezzo" tra inf e sup è dato dall'indice med = (sup + inf) div 2. Se sup+inf è pari, non esiste un vero punto di mezzo e med corrisponde al punto "quasi in mezzo, a sinistra".

Scrivere lo schema di ricerca binaria è facile; l'unica difficoltà è capire cosa succede nei casi limite: inf molto vicino a sup.

Vediamo prima lo schema che corrisponde all'idea, senza preoccuparci dei casi limite.

```
function ricbin( xx: float,  aa : array [1 .. n] of float, inf, sup : integer) : boolean
var  med : integer
{    med ← (sup + inf) div 2
     if  aa[med] = xx  then  return (true)
     else      if  aa[med] > xx
               then      return  ( ricbin (xx, aa, inf, med-1) )
               else      return  ( ricbin (xx, aa, med+1, sup) )
}
```

Problemi con questo schema:

1. Non viene mai restituito "falso", quindi manca qualche cosa.
2. Casi limite (supponendo inf = sup):
   - inf = sup. allora med = inf = sup; quindi: inf > med-1 nella prima chiamata ricorsiva e med+1 > sup nella seconda chiamata ricorsiva;
   - inf = sup-1 allora med = inf; quindi: inf > med-1 nella prima chiamata ricorsiva, nessun problema nella seconda chiamata;
   - inf = sup-2, sup-3,..... tutto a posto

   Infine, come sempre negli array, bisogna controllare di non uscire dai limiti, 1 ed n.

   Vediamo una prima soluzione, molto diretta, in cui si assume sempre che sia:

   1 = inf = sup = n; questo è vero alla chiamata principale, con inf=1 e sup=n, e viene mantenuto dall'algoritmo.

```
function ricbin_1( xx: float,  aa : array [1 ..n] of float, inf, sup : integer ) : boolean
var   med : integer
{    if  inf = sup  then  return ( aa[inf] = xx )        produce vero o falso
     else      if  inf = sup-1  then  return ( (aa[inf] = xx)  or  (aa[sup] = xx) )
               else      {    med ← (sup + inf) div 2
                              if  aa[med] = xx  then  return (true)
                              else      if  aa[med] > xx
                                        then      return ( ricbin_1(xx, aa, inf, med-1) )
                                        else      return ( ricbin_1(xx, aa, med+1, sup) )
                         }
}
```

Questa soluzione segue fedelmente l'analisi dei casi particolare fatta precedentemente ed è quindi molto chiara.

Un'altra soluzione, meno ovvia, è quella di permettere le chiamate ricorsive anche con gli indici "in ordine rovesciato" (il primo maggiore del secondo), ma controllare subito questo fatto: quando troviamo che gli

indici sono in ordine rovesciato, concludiamo che abbiamo “esaurito l’array” senza aver trovato l’elemento che cercavamo, quindi restituiamo il valore falso.

```
function ricbin_2( xx: float,  aa : array [1 ..n] of float, inf, sup : integer ) : boolean
var  med : integer
{
    if  inf > sup  then  return (false)
    else  {  tutto questo anche nel caso inf=sup:
             med ← (sup + inf) div 2
             if  aa[med] = xx  then  return (true)
             else     if  aa[med] > xx
                      then     return ( ricbin_2 (xx, aa, inf, med-1) )
                      else     return ( ricbin_2 (xx, aa, med+1, sup) )
          }
}
```

Per esercizio:

- provare a simulare l’esecuzione della chiamata ricbin_2(x, arf, 1, n), con un n piccolo e nei due casi: x è in a, x non c’è;
- riscrivere l’algoritmo, in modo che produca l’indice della posizione in cui si è trovato l’elemento (e -1, se l'elemento non si trova).

L’idea di ricerca binaria si presta molto bene ad una implementazione ricorsiva, ma si può anche sviluppare in modo iterativo, usando un while. Qui sotto la versione in pseudo codice; una versione in C si trova in [KR] paragrafo 3.3.

Ricerca binaria iterativa:

```
function ricbin_3( xx: float,  aa : array [1 ..n] of float ) : boolean
var      inf, sup, med : integer
         trovato : boolean
{
    inf ← 1;   sup ← n  ;   trovato ← false
    while   (inf = sup)  and  not trovato   do
            if  inf = sup   then    {    trovato ← ( aa[inf] = xx )
                                         sup  ← inf – 1
                                    }
            else     {    med ← (sup + inf) div 2
                          if  aa[med] = xx  then   trovato ← true
                          else     if  aa[med] > xx then     sup ← med-1
                                                       else      inf ← med+1
                     }
    return (trovato)
}
```

2.5 Merge sort

Il problema è quello di ordinare (in modo crescente) un array.

Idea dell'algoritmo, formulato come procedura ricorsiva:

> si divide l'array "a metà"; si ordinano, separatamente, le due parti (questo si fa con due chiamate ricorsive alla procedura stessa); poi si combinano le due parti ordinate, con una operazione di "fusione" (merging, in inglese).

Quindi, come nel caso della ricerca binaria (e di tanti altri algoritmi), alla base c'è una strategia del tipo *divide et impera* (dividi e domina/conquista).

Nota. Nel caso di array di dimensione notevole, merge sort è preferibile rispetto ad altri metodi, concettualmente più semplici, come selection sort (chiariremo questo punto parlando di complessità), anche se ci sono metodi ancora migliori (ad esempio heapsort e quicksort, che non vedremo). Si presta inoltre ad essere adattato per ordinare liste e file; in questi casi è molto valido. Qui comunque ci serve come esempio di ricorsione, ed è un esempio molto interessante.

Prima di tutto, vediamo che l'operazione di fusione è facile. In generale:

date due successioni ordinate (in modo crescente): $X = x_1, ...x_p$ e $Y = y_1,...,y_q$ si tratta di ottenere una unica successione ordinata Z, di lunghezza p+q. Ad esempio:

se X = 2, 3, 10, 20 e Y = 4, 15, 25, 27

allora Z = 2, 3, 4, 10, 15, 20, 25, 27.

Un algoritmo per la fusione è:

usiamo 4 indici: i per la successione X, j per la successione Y, k per la successione Z
ed aux per una istruzione for;

$i \leftarrow 1$; $j \leftarrow 1$; $k \leftarrow 1$

while (i = p and j = q) do

{ if $x_i = y_j$ then { $z_k \leftarrow x_i$; i++ }
else { $z_k \leftarrow y_j$; j++ }
k++ }

a questo punto abbiamo esaurito una delle due successioni; nell'altra resta almeno un elemento (infatti: ad ogni confronto, un solo elemento viene copiato in Z); questi elementi rimasti sono tutti maggiori di quelli già in Z, non resta che portarli in Z.

if i = p then *sono rimaste delle x:* per aux = i,..., p : { $z_k \leftarrow x_{aux}$; k++ }
else *sono rimaste delle y:* per aux = j,..., q : { $z_k \leftarrow y_{aux}$; k++ }

Nel caso specifico dell'ordinamento di un array, le due successioni X ed Y corrispondono a 2 parti di array e la successione Z ad un array ausiliario, come vedremo. Ora ritorniamo allo schema dell'ordinamento sapendo che, più o meno, la fusione la sappiamo fare.

La procedura di ordinamento la chiamiamo ms.

Ha 3 parametri: aa (l'array da ordinare), inf e sup (che delimitano la porzione di array su cui si lavora).

Usa una procedura ausiliaria, chiamata merge che è quella che opera la fusione, più o meno come visto sopra.

Quando vogliamo ordinare AA : array [1 .. n] of integer la chiamata è: ms(AA, 1, n).

```
procedura ms (      aa : array [1 .. n] of integer      parametro IN-OUT
                    inf, sup : integer          parametri IN che variano tra 1 ed n, con inf = sup    )

var   med: integer    anch'essa varia tra 1 ed n

{   if   inf < sup      quindi nell'array ci sono almeno 2 elementi
    then    {   med ← (inf+sup) div 2
                ms(aa, inf, med)
                    quando siamo arrivati qui, l'array è stato ordinato da  inf a  med
                ms(aa, med+1, sup)
                    quando siamo arrivati qui, l'array è stato ordinato da med+1  a  sup
                merge (aa, inf, med, sup)
            }
    else: niente, perché se inf=sup, l'array contiene 1 solo elemento ed è quindi ordinato
}
```

Cerchiamo di capire come funziona ms e se produce il risultato voluto.
Il ragionamento informale è il seguente.

Cosa deve fare ms(aa, inf, sup) ? Deve ordinare aa da inf a sup.
Sappiamo che inf = sup.

- Se inf = supnon c'è niente da fare, l'array è ordinato da inf a sup
- Altrimenti (inf < sup):

  se suppongo che
  la chiamata ms(aa, inf, med) faccia bene il suo lavoro, allora il suo effetto è:
  l'array è stato ordinato da inf a med;
  la chiamata ms(aa, med+1, sup) faccia bene il suo lavoro, allora il suo effetto è:
  l'array è stato ordinato da med+1 a sup;
  se inoltre la procedura merge è capace di fare la cosa seguente:
  prendere i due pezzi di array ordinati (da inf a med e da med+1 a sup) e fonderli
  lasciando l'array ordinato da inf a sup
  allora effettivamente, **ms(aa, inf, sup) ordina l'array da inf a sup.**

Questo ragionamento può lasciare perplessi, ma funziona. E` anche importante capire perché.

Per prima cosa: da quello che abbiamo già visto, dovrebbe essere chiaro che **si può** scrivere una procedura merge che soddisfi alle nostre richieste; quindi merge non è un problema per capire se ms funziona o no. Più avanti ritorneremo su merge, ma per ora ci basta sapere che si può fare.

A questo punto resta solo la supposizione che le due chiamate interne ad ms funzionino. Può sembrare che ci si morda la coda; non è cosí e lo si vede trasformando questo ragionamento in una dimostrazione per induzione.

Dimostrazione di correttezza per la procedura ms (nella dim. diamo per scontato che *ordinare* significa *riordinare gli elementi presenti*, e non mettere altri valori, a caso, ma in ordine crescente).

Ipotesi:

- 1 = inf = sup = n
- la procedura merge soddisfa alla seguente specifica:

se aa è ordinato (in modo crescente) da inf a med e da med+1 a sup (estremi compresi),

la chiamata merge(aa, inf, med, sup) ordina aa da inf a sup.

Tesi: la chiamata ms(aa, inf, sup) ordina aa da inf a sup

Dimostrazione per induzione aritmetica generalizzata su k = sup-inf

(ricordiamo che inf = sup, quindi k =0)

base: k=0. Questo corrisponde al caso inf=sup; abbiamo già visto che non c'è nulla da fare; correttamente, ms in questo caso non fa nulla (esegue solo il test inf < sup ed esce).

passo: k>0.

La procedura ms in questo caso: calcola med, esegue le due chiamate a se stessa e poi chiama merge. Il punto è che le due chiamate ricorsive sono fatte su parti di array di dimensione minore di k.

Ricordiamo che abbiamo l'ipotesi induttiva che è la seguente

la chiamata ms(aa, basso, alto) ordina aa da basso a alto, per ogni basso ed alto t.c.

alto-basso < k ed inoltre t.c rispettano le ipotesi: 1 = basso = alto = n

Consideriamo la prima chiamata: ms(aa, inf, med). Se dimostriamo che med-inf < k e che 1=inf=med=n otteniamo ms(aa, inf, med) ordina aa da inf a med.

In effetti si ha, per definizione di med: inf = med e med < sup e quindi otteniamo quello che vogliamo.

Per la seconda chiamata ms(aa, med+1, sup) si ragiona in modo analogo, infatti inf < med+1 e med+1 = sup.

In conclusione: ms(aa, inf, med) ordina aa da inf a med

ms(aa, med+1, sup) ordina aa da med+1 a sup

merge(aa, inf, med, sup) completa il lavoro

e l'array aa è ordinato da inf a sup.

Nota. Nella maggior parte dei libri che spiegano merge sort, l'accento è posto sulla simulazione dell'esecuzione dell'algoritmo, magari aiutandosi con un esempio. Per cui il discorso diviene qualcosa del tipo:

> L'effetto di ms(aa, inf, sup) è di dividere l'array; quello di ms(aa, inf, med) è di dividere ancora,....., fino a che non si arriva ad un solo elemento; allora merge, partendo dal basso, ricompone l'array, ordinandolo.

Può darsi che qualcuno si convinca della correttezza della procedura con un simile discorso. In realtà è solo un modo molto grossolano di descrivere l'esecuzione della procedura. Se si vuole seguire questa strada però non ci sono mezze misure: bisogna veramente simulare l'esecuzione, magari in modo schematico. Facciamolo, anche perchè è comunque utile avere un'idea di come è implementata la ricorsione. Per motivi che saranno subito chiari, ci limitiamo ad un piccolo array di 5 elementi : AA = 40 , 10, 20, 50, 30

Vediamo, schematicamente, come procede l'esecuzione di ms(AA, 1, 5), sempre supponendo che le chiamate alla procedura merge , producano il risultato voluto, come detto sopra.

1o passo:		**ms(AA, 1, 3)**
		ms(AA, 4, 5)
ms(AA, 1, 5)	*si espande in:*	**merge(AA, 1, 3, 5)**

Dunque: (1o passo) la chiamata ms(AA, 1, 5) viene "sostituita con il corpo della procedura"; semplificando, ed omettendo il test, il calcolo di med,..., viene sostituita da 3 chiamate a procedura.

L'esecuzione procede (2o passo) esaminando la prima di queste: ms(AA, 1, 3); le altre due chiamate aspettano fina a che non sia stato completato tutto quello che la chiamata ms(AA, 1, 3) ha generato.

Nel seguito, evidenziamo **in grassetto** quello che cambia.

2o passo:

		ms(AA, 1, 2)
		ms(AA, 3, 3)
ms(AA, 1, 3)	*si espande in:*	**merge(AA, 1, 2, 3)**
ms(AA, 4, 5)		ms(AA, 4, 5)
merge(AA, 1, 3, 5)		merge(AA, 1, 3, 5)

3o passo:

		ms(AA, 1, 1)
		ms(AA, 2, 2)
ms(AA, 1, 2)	*si espande in:*	**merge(AA, 1, 1, 2)**
ms(AA, 3, 3)		ms(AA, 3, 3)
merge(AA, 1, 2, 3)		merge(AA, 1, 2, 3)
ms(AA, 4, 5)		ms(AA, 4, 5)
merge(AA, 1, 3, 5)		merge(AA, 1, 3, 5)

4o passo:

ms(AA, 1, 1)	*si espande in:*	**“niente”**
ms(AA, 2, 2)		ms(AA, 2, 2)
merge(AA, 1, 1, 2)		merge(AA, 1, 1, 2)
ms(AA, 3, 3)		ms(AA, 3, 3)
merge(AA, 1, 2, 3)		merge(AA, 1, 2, 3)
ms(AA, 4, 5)		ms(AA, 4, 5)
merge(AA, 1, 3, 5)		merge(AA, 1, 3, 5)

5o passo:

ms(AA, 2, 2)	*si espande in:*	**“niente”**
merge(AA, 1, 1, 2)		merge(AA, 1, 1, 2)
ms(AA, 3, 3)		ms(AA, 3, 3)
merge(AA, 1, 2, 3)		merge(AA, 1, 2, 3)
ms(AA, 4, 5)		ms(AA, 4, 5)
merge(AA, 1, 3, 5)		merge(AA, 1, 3, 5)

6o passo:

merge(AA, 1, 1, 2)	*ha per effetto:*	AA = **10, 40,** 20, 50, 30
ms(AA, 3, 3)		ms(AA, 3, 3)
merge(AA, 1, 2, 3)		merge(AA, 1, 2, 3)
ms(AA, 4, 5)		ms(AA, 4, 5)
merge(AA, 1, 3, 5)		merge(AA, 1, 3, 5)

7o passo:

ms(AA, 3, 3)	*si espande in:*	**"niente"**
merge(AA, 1, 2, 3)		merge(AA, 1, 2, 3)
ms(AA, 4, 5)		ms(AA, 4, 5)
merge(AA, 1, 3, 5)		merge(AA, 1, 3, 5)

8o passo:

merge(AA, 1, 2, 3)	*ha per effetto:*	AA = **10, 20, 40**, 50, 30
ms(AA, 4, 5)		ms(AA, 4, 5)
merge(AA, 1, 3, 5)		merge(AA, 1, 3, 5)

9o passo:

		ms(AA, 4, 4)
		ms(AA, 5, 5)
ms(AA, 4, 5)	*si espande in:*	**merge(AA, 4, 4, 5)**
merge(AA, 1, 3, 5)		merge(AA, 1, 3, 5)

10 mo:	**ms(AA, 4, 4)**	*si espande in:*	**"niente"**
11 mo::	**ms(AA, 5, 5)**	*si espande in:*	**"niente"**
12 mo:	**merge(AA, 4, 4, 5)**	*ha per effetto:*	AA = 10, 20, 40, **30, 50**
	merge(AA, 1, 3, 5)		merge(AA, 1, 3, 5)

13 mo:	**merge(AA, 1, 3, 5)**	*ha per effetto:*	AA = **10, 20, 30, 40, 50**

Bene: con grande dispendio di energie abbiamo simulato l'esecuzione di merge sort su un ridicolo array di 5 elementi.... Il fatto è che non conviene cercare di imitare i calcolatori; il risultato è sempre e solo patetico. La strada da seguire è un'altra: ragionare "ad alto livello", partendo dall'idea originaria dell'algoritmo e precisandola, come abbiamo fatto sopra (anche se non sempre è facile arrivare fino ad una dimostrazione).

Veniamo alla **procedura merge.** Come abbiamo detto, la procedura, a partire dall'array aa ordinato (in modo crescente) da inf a med e da med+1 a sup (estremi compresi), deve modificare aa in modo che sia ordinato da inf a sup. La procedura si ottiene adattando in modo abbastanza semplice l'algoritmo visto precedentemente per le successioni.

```
procedura merge (      aa : array [1 .. n] of integer      parametro IN-OUT
                       inf, med, sup : integer             parametri IN, che variano tra 1 ed n )
var:    bb : array [1 .. n] of integer  array ausiliario
        i, j, k : indici interi         i  scorre da inf a med, j scorre da med+1 a sup   nell'array aa
                                        k scorre da  inf a sup nell'array bb
        aux indice intero               ausiliario
{   i ← inf ;   k ← inf ;   j ← med+1
    while ( i = med   and   j = sup )  do
        {   if  aa[ i ] = aa[ j ]          then   { bb[ k ] ← aa[ i ]  ;   i++ }
                                           else   { bb[ k ] ← aa[ j ]  ;   j++ }
            k++
        }
            questo è il punto (*) - vedere sotto
    if  i = med   then   per aux = i, ..., med : { bb[ k ] ← aa[aux] ;  k++ }
                 else   per aux = j, ..., sup : { bb[ k ] ← aa[aux] ;  k++ }
    a questo punto bb contiene, da inf a sup, gli elementi di aa ordinati come si voleva;
    basta quindi ricopiare bb su aa:
    per aux = inf, ..., sup :  aa[aux] ← bb[aux]
}
```

La procedura si può migliorare (rendendola più astuta), osservando la situazione a conclusione del while, nel punto (*). Ci sono due casi possibili (si consiglia di seguire quanto sotto facendo un disegno dell'array):

1o caso: si esce del while con i = med+1 e j = sup e quindi k = j ; allora:

- aa[1], ..., aa[inf-1] e aa[sup+1], ..., aa[n] non ci interessano (e non sono stati toccati)
- aa[inf], ..., aa[med], aa[i],..., aa[j-1] sono stati copiati in bb; notare che aa[j-1] è in bb al posto k-1
- restano: aa[j], ..., aa[sup] che sono ordinati e maggiori o uguali dell'ultimo elemento messo in bb: bb[k-1]; se li lasciamo dove sono e ricopiamo gli elementi di bb in aa da inf a k-1 (= j-1) otteniamo aa ordinato da inf a sup.

2o caso: si esce del while con i = med e j = sup+1 ; allora:

- aa[1], ..., aa[inf-1] e aa[sup+1], ..., aa[n] non ci interessano (e non sono stati toccati)
- gli elementi aa[inf], ..., aa[i-1] e aa[med+1], ..., aa[sup] sono stati copiati in bb; stanno, in bb, nelle posizioni da inf a k-1
- restano: aa[i], ..., aa[med] che sono ordinati e maggiori o uguali dell'ultimo elemento messo in bb: bb[k-1] ; il loro posto definitivo è: aa[k], ..., aa[sup] (visto che, degli elementi di aa che ci interessano, gli altri sono in bb da inf a k-1); allora spostiamoli direttamente lí, prima di ricopiare bb su aa.

Quindi, una versione ottimizzata di merge (ottimizzata nel senso che evita copiature inutili), si ha sostituendo la parte che segue il punto (*) con:

```
if  i = med   then     {    j ← k          tanto  il  vecchio  j  non serve più a nulla
                            per aux = i, ..., med : { aa[ j ] ← aa[aux] ;  j++ }
con questo abbiamo eseguito lo spostamento richiesto dal caso 2; ora si ricopia bb su aa:
 per aux = inf, ..., k-1 :  aa[aux] ← bb[aux]
```

Per completezza, vediamo quale può essere la struttura di un programma C, sfruttando gli array dinamici. Limitiamo al minimo i commenti.

```
#include <stdio.h>
#include <stdlib.h>   /*  per  calloc  */

void ms ( int aa[ ], int inf, int sup );

void main( )
{
    int n;      /*  la dimensione   dell'array da ordinare  */
    int * a ;   /*  l'array  */
    int k ;

    scanf ("%d", &n);

    if (n < 1)  printf ( "ERRORE");
    else {
         a =  calloc (n, sizeof(int));
               /* qui bisognerebbe controllare che  a  non ÿ NUL .... */

         for (k=0; k<n; k++)  scanf ("%d", a+k);

         printf ( "array da ordinare: \n\n ");
         for (k=0; k<n; k++)  printf ( "%d  ", *(a+k));

         ms(a, 0, n-1);

       printf ( "array ordinato: \n\n ");
       for (k=0; k<n; k++)  printf ( "%d  ", *(a+k));
     }
}


void merge ( int aa[ ], int inf, int med, int sup);

void ms( int aa[ ], int inf, int sup)
{
   int med;
   if (inf < sup)  {  med = (inf + sup) / 2 ;
                ms( aa, inf, med );
                ms( aa, med+1, sup );
                merge( aa, inf, med ,sup );
               }
 }
```

```
void merge ( int aa[ ], int inf, int med, int sup )
{
    /* seguiamo la versione ottimizzata */
    int * bb ;     /*  array ausiliario */
    int i, j, k, aux ;

    bb =  calloc (sup - inf + 1, sizeof(int));
          /* qui bisognerebbe controllare che  bb non ÿ NUL ....
             notare che chiediamo solo la quantita` di memoria che serve
             quindi l'indice  k  parte da 0 e non da  inf e ci sono altre
             piccole modifiche alla gestione degli indici
          */
    i = inf;   j = med+1 ;    k = 0 ;

    while ( (i <= med ) && (j <= sup))
        { if ( aa[i] <= aa[j] )   { bb[ k ] = aa[i] ; i++ ; }
          else                    { bb[ k ] = aa[j] ; j++ ; }
          k++;
        }
    if  ( i <= med )  {
          j = k+inf;
          for (aux = i; aux <= med;  aux ++) { aa[j] = aa[aux];    j++; }
      }

    for (aux = 0; aux < k ;  aux ++)  aa[ aux+inf ] = bb[ aux ];

    free(bb) ;  /* visto che non serve piu` .....
                notare che la memoria occupata dall'array puntato da bb
                NON viene rilasciata all'uscita della procedura, ma solo a
                conclusione di tutto
             */
}
```

2.6 Ricorsione indiretta e incrociata

Quelli visti sono tutti esempi di *ricorsione diretta*: una funzione o procedura contiene, nel corpo, una o più chiamate a sè stessa. Si può anche avere una *ricorsione indiretta*:

> la procedura P chiama direttamente la procedura Q, che chiama direttamente la procedura R, ..., che chiama direttamente P.

E' anche possibile la *ricorsione incrociata* (si dice anche: procedure/funzioni *mutuamente ricorsive*); nel caso più semplice: P chiama Q e Q chiama P.

Un esempio di tutto questo si ha considerando la "valutazione delle espressioni" (qualcosa verrà vista a lezione o durante le esercitazioni).

2.7 Quando usare la ricorsione

Difficile dare una risposta netta. Prima di tutto, bisogna familiarizzarsi con la tecnica. Fatto questo si vede che in molte situazioni l'approccio ricorsivo risulta chiaro e semplice. Due sono i casi che vedremo più in dettaglio:

- algoritmi per lavorare su tipi di dato definiti induttivamente; in particolare vedremo il caso delle liste e degli alberi;

- algoritmi che utilizzano la tecnica del “divide et impera”; due esempi significativi di questa tecnica sono proprio ricerca binaria e merge sort.

Abbiamo visto che l’idea della ricerca binaria si può facilmente tradurre in *stile iterativo* (cioè usando istruzioni di ripetizione, while, repeat, for); decisamente meno facile è la formulazione iterativa dell’idea di merge sort. Analogamente, si vedrà che per lavorare sulle liste è abbastanza indifferente usare uno stile iterativo o uno ricorsivo, mentre lavorando su alberi è in generale più facile usare lo stile ricorsivo.

Va detto che esiste una tecnica standard per eliminare la ricorsione, usando uno stack: in pratica si fa “a mano” quello che l’implementazione fa automaticamente. Quindi non c’è nulla che si fa con la ricorsione che non possa essere fatto senza. Tuttavia, in molti casi la trasformazione per eliminare la ricorsione è faticosa e produce un algoritmo che è molto meno chiaro di quello ricorsivo.

Una delle obiezioni che si fa alla ricorsione è che il meccanismo può risultare “costoso” in termini di tempo di esecuzione; del resto anche in assenza di ricorsione, il meccanismo

dichiarazione di funzione/procedura -- chiamata della funzione/ procedura

ha un certo costo. Questo punto si chiarirà meglio parlando di *complessità degli algoritmi*. Qualcosa si può comunque anticipare.

Nei linguaggi di programmazione moderni, tra cui il C, le procedure / funzioni sono considerate uno degli strumenti principali per programmare in modo sicuro e flessibile; quindi la questione: procedure sí, procedure no, è risolta con un sí deciso. Gli eventuali “costi”, non rilevanti in assoluto, sono più che bilanciati dai vantaggi per il programmatore. Se poi si confronta, in termini di soldi, il costo del lavoro umano con quello del lavoro della macchina, allora diventa evidente che conviene far lavorare la macchina. In questa ottica, tutte le volte che risulta comodo e naturale usare la ricorsione, conviene usarla.

Se l’algoritmo che stiamo progettando, una volta diventato programma, deve essere usato molte volte o è una componente critica di un sistema, bisogna valutarne in qualche modo le prestazioni (ad esempio stimandone la complessità). In qualche caso si vede che l’approccio ricorsivo è troppo inefficiente (in realtà, in genere, non è colpa della ricorsione, ma dell’idea su cui si basa l’algoritmo); un esempio è il calcolo dei coefficienti binomiali: la funzione ricorsiva che abbiamo discusso precedentemente ha una complessità troppo elevata. A questo punto si cerca di capire cosa non funziona e di rimediare, magari cambiando strategia. Per i coefficienti binomiali, vedremo (non in queste note) proprio come, analizzando le ragioni dell’inefficienza della soluzione ricorsiva, si arriva ad una soluzione molto migliore.

3

Tipi, tipi di dato, strutture dati

Si tratta di concetti importanti, ma un po' sfuggenti; spesso si trovano definizioni discordanti.

Per il concetto di *tipo* e per quello di *tipo di dato* (che abbrevieremo in **tdd**) esiste una formalizzazione nel contesto algebrico che vediamo, limitandoci però alle prime definizioni.

Per il concetto di *struttura dati* va ancora peggio: non esiste una formalizzazione e c'è la massima confusione; ci ritorneremo dopo aver precisato gli altri concetti.

Abbiamo visto diversi esempi di problemi che coinvolgono i numeri interi; il C, il Pascal e quasi tutti i linguaggi di programmazione forniscono gli interi. Abbiamo anche visto che il C ed il Pascal hanno istruzioni (if-then-else, while,...) che prevedono una *condizione*. Tali condizioni, da un punto di vista matematico, sono (o corrispondono a) formule logiche; in Pascal sono caratterizzate come tali, anche se si usa il nome *espressioni booleane*; alla base c'è la possibilità di considerare nel linguaggio dei valori di verità (vero e falso) e di combinarli con operazioni tipo: and, or,...

Quando si dice "gli interi", in genere si pensa ai numeri, all'insieme **Z**. In effetti, se avessimo solo i numeri riusciremmo a fare ben poca strada. Se riflettiamo un po', (quasi) tutte le volte che usiamo dei numeri usiamo anche delle *operazioni su di essi* (somma, sottrazione, ...). Anche per quanto riguarda le condizioni, se avessimo solo a disposizione "vero" e "falso" non andremmo lontani.

In generale: **se abbiamo bisogno di un certo genere di "oggetti", abbiamo (quasi sempre) anche bisogno di "modi per operare su questi oggetti".**

Un altro punto importante è la distinzione tra: oggetti ed operazioni, da un lato, e simboli (piú in generale: linguaggio), dall'altra: il numero *tre* si puó rappresentare in molti modi: 3, 11, III, ... l'operazione di *moltiplicazione* si puó indicare con $\bullet$, oppure con *, ...

La nozione di tipo di dato, cerca appunto di precisare questa situazione; quindi prevede (almeno) tre ingredienti: "oggetti", "modi di operare su di essi", simboli. Un modo di formalizzare il tutto è quello di utilizzare la nozione di *algebra*; purtroppo, le algebre omogenee viste nel corso di Algebra non sono sufficienti; come minimo servono le algebre eterogenee con operazioni parziali.

Vediamo per prima cosa l'esempio del tipo di dato booleani, che richiede solo le algebre omogenee, poi, con altri esempi, vedremo quali aspetti richiedono un formalismo piú ricco.

3.1 Booleani

Consideriamo:

- la segnatura Σ, con: $\Sigma_0 = \{T, F\}$, $\Sigma_1 = \{ \neg \}$, $\Sigma_2 = \{ \wedge, \vee \}$, $\Sigma_k = \emptyset$, per k >2
- l'insieme B = {*vero, falso*}
- le operazioni su B: *and, or, not* (standard)
- l'interpretazione I tale che: F^B = *falso*, T^B = *vero*, $\neg^B$ = *not*, $\wedge^B$ = *and*, $\vee^B$ = *or*

  La coppia (B, I) è una Σ-algebra (secondo la definizione vista nel corso di Algebra).

  Il tdd è appunto la coppia: $< \Sigma$, (B, I) >, cioè: segnatura + algebra.

  La segnatura ci fornisce i *simboli*, cioè i nomi delle operazioni, l'insieme B ci fornisce gli *oggetti*; infine, tramite I otteniamo le *operazioni*.

  Però non è l'unico tdd che si può considerare, volendo un tdd che descriva i valori di verità ed operazioni su di essi. Possibili varianti:

- nella segnatura potremmo aggiungere o togliere simboli; per esempio, una segnatura altrettanto valida è Ω, con: $\Omega_0 = \{T\}$, $\Omega_1 = \{ \neg \}$, $\Omega_2 = \{ \vee \}$, $\Omega_k = \emptyset$, per k >2;
- nell'algebra potremmo cambiare gli oggetti; quello che conta è che siano due (almeno, e distinti);
- ..... eccetera.

Dal punto di vista della programmazione, le scelte che si operano nel definire i tipi di dato costituiscono uno degli aspetti rilevanti della fase di progetto.

Per il tipo di programmazione che ci interessa (programmazione in piccolo), il modo di procedere è: analizzando il problema e l'idea di soluzione per esso, si capisce di aver bisogno di certi oggetti e di certe operazioni su di essi; si precisa il tutto scegliendo dei nomi (e di qui nasce la segnatura), definendo per bene gli oggetti, le operazioni, la corrispondenza tra simboli ed operazioni, cioè l'interpretazione,....

Nel caso dei booleani, le cose procedono quindi in questo modo: in base a quello che voglio fare, decido, o mi rendo conto, che:

1. mi servono due valori (oggetti), decidiamo che siano *vero, falso*; quindi ho fissato l'insieme B;
2. mi servono i connettivi proposizionali *and, or, not*; se ho un po' di conoscenze di logica proposizionale so che con questi tre posso esprimere anche gli altri (anzi, basterebbe meno); a questo punto ho fissato le operazioni con arità =1;
3. restano le costanti (operazioni di arità 0): per comodità scelgo di avere 2 costanti, cioè di poter esprimere direttamente i due valori di verità; (in effetti una sola costante sarebbe sufficiente: l'altro valore si ottiene usando l'operazione *not;* se invece non avessi costanti mi troverei in difficoltà ...); a questo punto ho completato la scelta delle operazioni;
4. non resta che scegliere i nomi, cioè la segnatura.

3.2 Interi

Seguendo l'opproccio delineato sopra, vediamo di definire un tdd interi.

- Valori: *0, 1, -1, 2, -2, ...* idealmente, tutto **Z**; in pratica, nei linguaggi di programmazione, un intervallo, grosso modo da -2^k a 2^k, per qualche k. Per semplicità, qui consideriamo il caso ideale (tutto **Z**).
- Operazioni non costanti (cioè di arità >0): anche qui c'è da fare una scelta; per ora supponiamo che si servano solo: *somma, sottrazione, moltiplicazione*.
- Costanti: se facciamo una scelta analoga a quella operata nel caso dei booleani, ci troviamo con infinite costanti (una per ogni elemento di **Z**); in effetti nei linguaggi di programmazione si fa proprio la scelta corrispondente. All'estremo opposto, potremmo provare a scegliere un insieme minimale di costanti, per esempio {*0, 1*}, oppure {*-1, 1*}; infatti, con *0* ed *1* ed usando le altre operazioni riesco ad ottenere tutti gli altri valori; lo stesso usando *-1* ed *1*.
- Simboli: ad esempio: +, -, * , e poi: 0 e 1, oppure -1 e 1 (notare la differenza tra il numero *0* ed il simbolo 0).

In base a questa discussione, e decidendo di usare {*0, 1*} come insieme di costanti, abbiamo il seguente tipo di dato:

- segnatura Σ, con: $\Sigma_0 = \{0, 1\}$, $\Sigma_2 = \{ +, -, * \}$, $\Sigma_k = \emptyset$, per k? 0,1
- Σ-algebra : (**Z**, I) dove I è quella ovvia.

Per vedere quali sono i problemi che costringono ad andare oltre le algebre omogenee, supponiamo di voler completare il nostro tipo di dato con una operazione di divisione ed una operazione di uguaglianza.

Divisione intera. Non può essere modellata come operazione binaria, secondo la def. data ad Algebra, perchè non è una funzione totale da $\mathbf{Z}^2$ in **Z** (non si può dividere per *0*); infatti, in matematica viene vista come: div : $\mathbf{Z} \times \mathbf{Z}\text{-}\{0\} \rightarrow \mathbf{Z}$.

Questo porta a considerare operazioni parziali (vedere sotto)

Uguaglianza. Se vogliamo poter parlare di uguaglianza, un modo naturale è aggiungere una operazione eq : $\mathbf{Z}^2 \rightarrow$ {*vero, falso*}. Anche questo esula dallo schema delle algebre omogenee, infatti non c'è omogeneità tra argomenti (interi) e risultato (booleano).

Questi due problemi si incontrano di continuo quindi vediamo di precisare le cose, definendo la funzioni parziali e le algebre eterogenee.

4 Funzioni parziali

Riprendendo definizioni viste nel corso di Algebra, siano A e B due insiemi.

Una **relazione** (o corrispondenza) tra A e B, sia R, è un sottinsieme di A x B.

A è il *dominio* e B è il *codominio*; inoltre l'insieme { b | esiste a in A tale che (a,b) è in R} è l'*immagine* di R (anche: immagine di A secondo R).

Nella matematica classica, R è una **funzione** (o applicazione) se:

1. *univocità*: (a,b), (a, b') appartengono ad R implica b = b'
2. *totalità:* per ogni a di A, esiste un b in B tale che (a,b) è in R

allora si scrive in genere: $R(a) = b$ invece di $(a,b) \in R$.

L'insieme A, può a sua volta essere un prodotto cartesiano $A_1 \times A_2 \ldots \times A_n$; in tal caso si usa scrivere;

$(a_1, a_2, \ldots, a_n, b) \in R$ invece di $((a_1, a_2, \ldots, a_n), b) \in R$ e

$R(a_1, a_2, \ldots, a_n) = b$ invece di $R((a_1, a_2, \ldots, a_n)) = b$

In informatica si usa anche il concetto di funzione parziale.

(Una relazione tra A e B) R è una **funzione parziale** (da A in B) se soddisfa la condizione di univocità.

A questo punto, se R soddisfa anche la condizione di totalità si dice che R è una **funzione totale**.

È importante capire che, in questo contesto, 'totale" e "parziale" non sono termini opposti; ciò è conseguenza diretta delle definizioni: una funzione parziale è una relazione che gode della proprietà (1); una funzione totale è una relazione che gode della (1) e anche della (2). In altre parole: se R è una funzione totale, allora è anche una funzione parziale.

Se R è una funzione parziale da A in B:

R è *definita su a* (oppure: *in a*) se esiste un b in B tale che (a,b) è in R (anche: R(a) = b);

R è *indefinita su/in a* altrimenti.

Si dice anche: R(a) è definito / indefinito.

Alcuni usano un simbolo speciale, in genere $\perp$, e scrivono:

$R(a) = \perp$ invece di R non è definita su a

Per chiarire la situazione, consideriamo la funzione inverso nei reali e chiamiamola inv;

- nell'uso matematico, inv è la funzione: $inv : \mathbf{R} - \{0\} \rightarrow \mathbf{R}$ tale che $inv(x) = 1/x$
- nell'uso informatico, si preferisce vedere inv come la funzione parziale:
  $inv : \mathbf{R} \rightarrow \mathbf{R}$ tale che $inv(x) = 1/x$ per $x \neq 0$
  inv non è definita su 0

  Non c'è contrasto; da una parte si restringe il dominio, in modo che sia soddisfatta la condizione di totalità; dall'altra, si ammette un dominio "ampio" e si precisa per quali valori la funzione è definita.

  Notazione:

  Alcuni lasciano implicita la parte "non definita", quindi scrivono solo:

  $inv : \mathbf{R} \rightarrow \mathbf{R}$ tale che $inv(x) = 1/x$ per $x \neq 0$

  Ritornando alla divisione intera:

- nell'uso matematico, div è la funzione: $div: \mathbf{Z} \times \mathbf{Z}-\{0\} \rightarrow \mathbf{Z}$ tale che div(x,y) = parte intera di x/y.
- qui la vedremo come la funzione parziale:
  $div : \mathbf{Z}^2 \rightarrow \mathbf{Z}$ tale che div(x, y) = parte intera di x/y per $y \neq 0$
  div(x, y) è indefinito se $y = 0$

Se in informatica ci si mettesse d'accordo a non usare mai "funzione", ma solo: "funzione parziale" e "funzione totale" non ci sarebbe nessun problema. Purtroppo non è cosí. La conseguenza è che quando si incontra la parola "funzione" (da sola) bisogna capire (dal contesto o da altro) se si tratta di funzione parziale o totale.

Per quanto riguarda queste note:

- nel contesto dei tipi di dato: operazione = operazione parziale (ricordiamo che le operazioni non sono altro che funzioni);
- negli altri casi cercheremo di precisare, a meno che non sia chiaro dal contesto.

5 Algebre eterogenee (many sorted)

Vediamo solo le prime definizioni, quelle che ci servono.

Segnature. Una segnatura (eterogenea) è una coppia $\Sigma = (S, OP)$, dove

- S è un insieme i cui elementi sono detti *sorti* o *tipi*;
- OP è un insieme di *simboli di operazione*, ciascuno con la sua *funzionalità*, che ha la forma seguente:
  $s_1 \times ... \times s_n \rightarrow s$ con $s, s_1, ..., s_n \in S$, n=0
  se n=0, prende la forma $\rightarrow s$ e i simboli con questa funzionalità sono *simboli di costante*.
- S ed Op sono non vuoti e disgiunti; di solito sono finiti, al piú sono numerabili (cioè in corrispondenza biunivoca con **N**).

Notazione. Per indicare che il simbolo op ha funzionalità $s_1 \times ... \times s_n \rightarrow s$, si scrive:

$op : s_1 \times ... \times s_n \rightarrow s$ (che diventa $op : \rightarrow s$ nel caso n=0).

Inoltre: op1, op2, ... : $s_1 \times ... \times s_n \rightarrow s$ è una abbreviazione per:

$op1 : s_1 \times ... \times s_n \rightarrow s$ e $op2 : s_1 \times ... \times s_n \rightarrow s$ e .....

Esempi.

1. Una segnatura omogenea, Σ, può essere vista come segnatura eterogenea, nel modo seguente. Come insieme S delle sorti prendiamo un qualunque insieme con un solo elemento, ad esempio {1}. Allora i simboli n-ari di Σ diventano simboli con funzionalità $s_1 \times ... \times s_n \rightarrow s$, dove $s=s_1 = ...=s_n=1$.

2. Una segnatura per il tdd interi visto prima:

- S= {int, bool}
- OP = { 0, 1, +, -, *, div, =}, dove 0, 1 : $\rightarrow$ int
  +, -, * , div : int $\times$ int $\rightarrow$ int
  = : int $\times$ int $\rightarrow$ bool

Algebre (eterogenee, con operazioni parziali).

Sia $\Sigma = (S, OP)$ una segnatura, Una Σ-algebra è una coppia $\mathbf{A} = (A, OP^A)$, dove:

- $A = \{A_s \mid s \in S\}$ è una famiglia di insiemi indiciata su S; cioè: per ogni sorte in S abbiamo un insieme corrispondente; notare che s?s' non implica A_s ? $A_{s'}$
- $OP^A = \{ op^A \mid op \in OP \}$, dove,
  se $op : s_1 \times ... \times s_n \rightarrow s$, allora op^A è una operazione (parziale) $op^A : A_{s1} \times ... \times A_{sn} \rightarrow A_s$
  nel caso n=0 si intende che op^A è un elemento di A_s e viene detto *(operazione) costante*.

Nota. Rispetto alle definizioni viste ad Algebra, è "sparita" l'interpretazione I. Solo apparentemente, perchè è implicita nella corrispondenza tra op ed op^A.

Esempi.

3. Se Σ è la segnatura dell'esempio 2, una Σ-algebra è data da:

- $A = \{A_{int}, A_{bool}\}$, con $A_{int} = \mathbf{Z}$ e $A_{bool} = \{vero, falso\}$
- OP^A è l'insieme che contiene

  0^A e 1^A che sono zero ed uno, rispettivamente

  $+^A, *^A, -^A$ che sono la somma, il prodotto, la sottrazione

  $div^A : \mathbf{Z} \times \mathbf{Z} \rightarrow \mathbf{Z}$ tale che div(x, y) = divisione intera di x per y, se y?0; indefinito altrimenti

  $=^A : \mathbf{Z} \times \mathbf{Z} \rightarrow \{vero, falso\}$ che è l'uguaglianza.

4. Consideriamo la segnatura $\Sigma = (S, OP)$, con S = {int, bool} e OP che contiene i seguenti simboli:

T , F: $\rightarrow$ bool

$\neg$: bool $\rightarrow$ bool

$\vee$, $\wedge$: bool $\times$ bool $\rightarrow$ bool

0, 1 : $\rightarrow$ int

+, -, * , div : int $\times$ int $\rightarrow$ int

= : int $\times$ int $\rightarrow$ bool

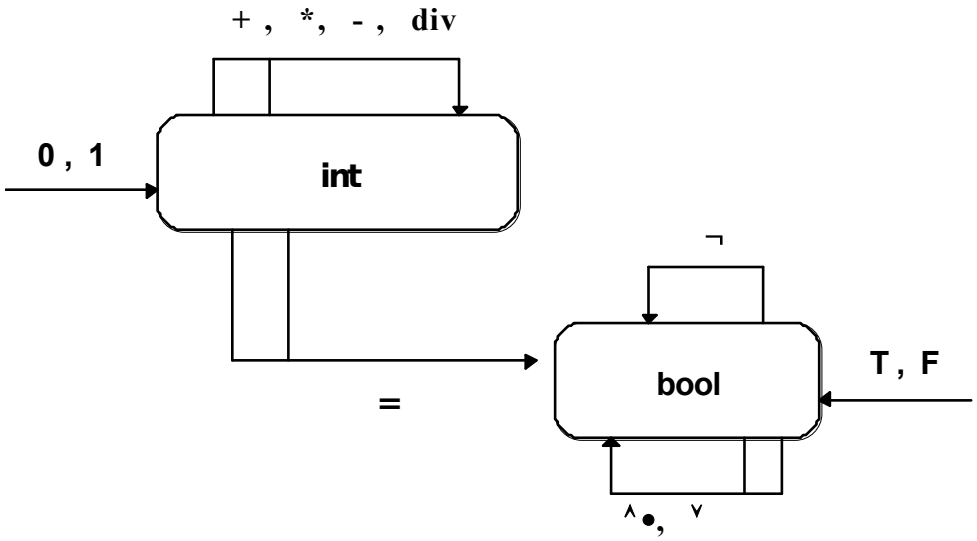
Questa segnatura ci fornisce simboli per scrivere *espressioni* sia di tipo intero che di tipo booleano; ad esempio: (1+1)*(1+1+1), espressione intera, (1+1 = 1*1) $\wedge$ (0 = 1-1), espressione booleana. Aggiungendo variabili, otteniamo qualcosa di piú interessante, as esempio: x = y+1.

Non abbiamo definito cosa sono le espressioni; per il momento accontentiamoci dell'intuizione; la formalizzazione si ha definendo i *termini* (con variabili) su una segnatura (definizione che estende quella vista ad Algebra).

Un algebra per questa segnatura si ottiene aggiungendo operazioni per T, F, $\neg$, $\vee$, $\wedge$ all'agebra vista nell'es. 3.

Segnature in forma grafica.

Spesso è comodo dare una rappresentazione grafica delle segnature: un tondo, o rettangolo per ogni sorte, archi etichettati per i simboli delle operazioni. Ad esempio la segnatura dell'es. 4 si rappresenta cosí:



Si perde un po' di informazione: l'ordine degli argomenti.

Tipi di dato e tipi. In conclusione, per quanto ci riguarda:

un **tdd** è una coppia: < segnatura, algebra sulla segnatura >, dove la segnatura, e quindi l'algebra, è eterogenea e le operazioni sono parziali; **tipo**, invece, è sinonimo di sorte.

Nel caso dei booleani, abbiamo una segnatura con una sola sorte; quindi un tdd con un solo tipo; facile confonderli. Nel caso degli interi, la segnatura dell'esempio 2 e l'algebra dell'esempio 3 ci forniscono un tdd "interi" con due tipi: int e bool.

Venendo ai linguaggi di programmazione: quando si dice che un linguaggio ha il tipo "interi", in realtà si vuol dire che ha un tdd "interi", mentre quando si dice che una variabile è di tipo "intero", si intende dire che è di sorte "intero" (cioè può assumere solo valori che stanno nell'insieme che corrisponde alla sorte "intero").

Spesso si distingue tra **tipi di dato *concreti*** e **tipi di dato *astratti***. La distinzione è chiara nel contesto della formalizzazione algebrica: i tipi di dato concreti sono quelli che consideriamo qui (in altri termini: qui tdd = tipo di dato concreto); i tipi di dato astratti sono classi di algebre, in genere definite specificando delle proprietà che devono possedere .....

Purtroppo, non tutti sono d'accordo, quindi si trovano definizioni diverse. Ritorneremo su questo dopo aver visto ulteriori esempi e parlato di implementazione di un tipo di dato.

Presentazione semplificata dei tipi di dato.

Spesso i tdd si descrivono in modo semplificato. invece di definire separatamente segnatura ed algebra, si mescolano un po' sintassi e semantica, in particolare si confondono simboli di operazione ed operazioni. Ad esempio, l'algebra dell'esempio 3 viene descritta come segue.

- sorti: int, bool
- sostegni: $A_{int} = \mathbf{Z}$, $A_{bool} = \{vero, falso\}$
- operazioni 0, 1 : $\rightarrow$ int -- corrispondono a zero ed uno

  +, -, * : int $\times$ int $\rightarrow$ int -- corrispondono a somma, sottraz., moltplicaz.

  div : int $\times$ int $\rightarrow$ int -- div(x,y) = divisione intera di x per y, se y?0 (indefinita altrimenti)

  = : int $\times$ int $\rightarrow$ bool -- corrisponde all'uguaglianza

6 Le successioni

Le successioni sono fra gli oggetti che si incontrano piú di frequente in programmazione; un caso particolare di successioni sono le stringhe. Forniscono un esempio interessante di tdd.

Per essere precisi: ci sono diversi tipi di dato centrati sull'idea di successione. Qui, prima precisiamo insiemi e notazioni e descriviamo alcune operazioni interessanti sulle successioni, poi vediamo due tdd notevoli, basati sulle successioni: gli staks e le code.

6.1 Successioni ed operazioni su di esse

Sia Elem un insieme non vuoto (anche infinito, come **N** (i naturali), **Z** (gli interi), **Q** (i razionali), ma numerabile (cioè in corrispondenza biunivoca con **N**)).

Consideriamo l'insieme Succ_Elem delle successioni (finite) su Elem. Considereremo sempre successioni finite, quindi non stiamo a ripeterlo ogni volta.

Scriveremo le successioni nella forma $<a_1, ..., a_n>$ con $n =0$, $a_k \in$ Elem, per $1=k=n$;
n è la *lunghezza* della successione; per $n=0$, ci si riduce a $< >$, cioè alla *successione vuota* (priva di elementi).

Le operazioni che si considerano di solito sulle successioni sono scelte tra le seguenti (come vedremo, non ha senso prenderle tutte):

1. le costanti: $<a_1, ..., a_n>$ con $n=0$ ed a_j in Elem
2. mk_succ mk_succ : Elem $\rightarrow$ Succ_Elem mk_succ(a) = <a>
3. concatenazione: conc : Succ_Elem $\times$ Succ_Elem $\rightarrow$ Succ_Elem
   conc($<a_1, ..., a_n>$, $<b_1, ..., b_p>$) = $<a_1, ..., a_n, b_1, ..., b_p>$
   spesso si usa la notazione infissa:
   $<a_1, ..., a_n> \bullet <b_1, ..., b_p>$ invece di conc($<a_1, ..., a_n>$, $<b_1, ..., b_p>$)
4. aggiunta di un elemento in testa: conc_t : Elem $\times$ Succ_Elem $\rightarrow$ Succ_Elem
   conc_t(a, $<b_1, ..., b_p>$) = $<a, b_1, ..., b_p>$
5. aggiunta di un elemento in coda: conc_c : Elem $\times$ Succ_Elem $\rightarrow$ Succ_Elem
   conc_c(a, $<b_1, ..., b_p>$) = $<b_1, ..., b_p, a>$
6. testa: testa : Succ_Elem $\rightarrow$ Elem
   testa($<a_1, ..., a_n>$) = a_1 per $n >0$ testa non è definita su $< >$
7. coda coda : Succ_Elem $\rightarrow$ Succ_Elem
   coda($<a_1, ..., a_n>$) = $<a_2, ..., a_n>$ per $n >0$; non è definita su $< >$
8. selelzione sel : Succ_Elem $\times$ **N** $\rightarrow$ Elem
   sel ($<a_1, ..., a_n>$, k) = a_k per $k =n$; non è definita altrimenti
9. vuota? is_empty : Succ_Elem $\rightarrow$ {vero, falso} is_empty(s) = vero sse s = < >
10. lunghezza: lunghezza : Succ_Elem $\rightarrow$ **N** lunghezza($<a_1, ..., a_n>$) = n
11. uguaglianza eq : Succ_Elem $\times$ Succ_Elem $\rightarrow$ {vero, falso}
    eq($<a_1, ..., a_n>$, $<b_1, ..., b_p>$) = vero se ($n=p$ e $a_k = b_k$, per $1=k=n$)
    falso altrimenti
    di solito, si usa il simbolo = e notazione infissa
12. Se Elem è un insieme ordinato, si può definire un ordine sulle successione; molto usato è l'*ordine lessicografico* less : Succ_Elem $\times$ Succ_Elem $\rightarrow$ {vero, falso} vedere oltre.

Note.

1. Le operazioni viste sopra non sono tutte indipendenti; ad esempio:

mk_succ(a) = <a>

mk_succ(a) = conc_t(a, < >) = conc_c(a, < >)

testa(s) = sel(s, 1)

sel(s, k) = testa(coda(....(coda(s) ...) con k-1 operazioni coda

eccetera ...

Inoltre non sono tutte ugualmente significative; alcune sono state messe quasi solo per esercizio.

2. E` facile vedere che la concatenazione è associativa: s1 • (s2 • s3) = (s1 • s2) • s3 ed ha un elemento neutro (identità), la successione vuota: < > • s = s • < > = s.

3. Le operazioni viste sopra si dividono in tre categorie:

- *costruttori*: le costanti, mk_succ, la concatenezione, l'aggiunta di un elemento; sono le operazioni che permettono di costruire successioni a partire da elementi o da successioni piú piccole
- *distruttori* (o *selettori*): testa, coda, sel; permettono di scomporre le successioni, di isolarne gli elementi
- operazioni che forniscono delle *proprietà*: lunghezza, is_empty, eq, less.

4. Ordine lessicografico. L'ordine lessicografico è quello dei dizionari, per cui:

latte < latteria, cane < gatto, ramo < remo, armadio < armatura,....

E` un po' complicato da definire formalmente; prima definiamo l'ordine stretto, che indichiamo con **<**, poi l'ordine debole, che indichiamo con =, mentre con < indichiamo l'ordine stretto in Elem. Allora in Succ_Elem, l'ordine stretto è dato da:

x **<** y se vale (1) oppure (2)

(1) x è un prefisso proprio di y ; cioè esiste z non vuoto t.c. y = x • z

(es. x = latte , y = latteria , z = ria)

(2) esistono u, w, z in Succ_Elem ed a,b in Elem, con a<b, tali che

x = u • <a> • w e y = u • <b> • z

(esempio: x = armadio , y = armatura , u = arma , a = d , b = t , ...)

Finalmente: x **=** y se (x **<** y) oppure (x = y) e

less(x, y) = x **=** y.

Per esercizio: verificare che è una buona definizione, cioè che si hanno le proprietà

riflessiva: x **=** x

transitiva: (x **=** y) e (y **=** z) implica x **=** z

antisimmetrica: (x **=** y) e (y **=** x) implica x = y

Le successioni considerate finora hanno lunghezza finita, ma illimitata; in molte applicazioni questo non interessa e/o non ha senso (ad esempio se vogliamo modellare un buffer di I/O). Quindi si considerano anche successioni con lunghezza limitata a priori.

6.1.1 Successioni di lunghezza limitata (a priori)

Fissato un intero K, K > 0: Succ_Elem_K = insieme delle successioni su Elem di lunghezza = K.

Le operazioni subiscono lievi modifiche (qui ci concentriamo solo sulle differenze):

1. le costanti: $<a_1, ..., a_n>$ con 0 = n = K
2. mk_succ non cambia, perchè K > 0 (a parte sostituire Succ_Elem con Succ_Elem_K)

3. conc : Succ_Elem_K $\times$ Succ_Elem_K $\rightarrow$ Succ_Elem_K
   conc(<a_1, ..., a_n>, <b_1, ..., b_p>) = <a_1, ..., a_n, b_1, ..., b_p> se n+p = K
   indefinita altrimenti
4. conc_t : Elem $\times$ Succ_Elem_K $\rightarrow$ Succ_Elem_K
   conc_t(a, <b_1, ..., b_p>) = <a, b_1, ..., b_p> se n+1 = K, indefinita altrimenti
5. conc_c analogo a conc_t
6. testa non cambia (a parte sostituire Succ_Elem con Succ_Elem_K)
7. coda non cambia (a parte sostituire .....)
8. sel non cambia (a parte sostituire .....)
   volendo si può riscriverla come sel : Succ_Elem_K $\times$ {0, ..., K} $\rightarrow$ Elem
9. vuota? non cambia (a parte sostituire .....)
10. lunghezza: non cambia (a parte sostituire .....)
    volendo si può riscriverla come lunghezza : Succ_Elem_K $\rightarrow$ {0, ..., K}
11. uguaglianza non cambia (a parte sostituire Succ_Elem con Succ_Elem_K)
12. ordine lessicografico: non cambia (a parte sostituire .....)

Nota.

E' utile sapere che se Elem ha n elementi, le successioni di lunghezza i sono n^i (questo si può verificare facilmente per induzione aritmetica su i).

Inoltre si vede che gli elementi di Succ_Elem_K sono $(n^{K+1} - 1) / (n - 1)$; vedere sotto per la dimostrazione. Anche per n e k abbastanza piccoli, " - 1" al numeratore è trascurabile e il tutto si può approssimare con $n^{K+1} / (n - 1)$.

Per n=2 si ottiene 2^{K+1} ; quindi: tutte le stringhe binarie di lunghezza minore o uguale a K sono appena il doppio di quelle lunghe esattamente K.

Dimostrazione della formula.

Indichiamo con S_K il numero di elementi in Succ_Elem_K (cioè la sua cardinalità). Poiché le stringhe di lunghezza i sono n^i, abbiamo:

$S_K = n^0 + n^1 + n^2 + ... + n^K = 1 + n + n^2 + ... + n^K$ e

$S_{K+1} = 1 + n + n^2 + ... + n^K + n^{K+1}$

si vede subito che $S_{K+1} = S_K + n^{K+1}$ ma anche: $S_{K+1} = 1 + n * S_K$

quindi: $S_K + n^{K+1} = 1 + n * S_K$ da cui si ottiene $S_K = (n^{K+1} - 1) / (n - 1)$.

6.2 Le stringhe

Discutiamo solo gli oggetti, non i tipi di dato che si costruiscono con essi.

Da un punto di vista matematico, le stringhe non sono altro che successioni (finite); quindi abbiamo già dato.

Da un punto di vista pratico, si parla di stringhe piuttosto che di successioni quando si vuole, ed è possibile, usare una notazione piú comoda e sintetica: abc invece di <a, b, c>.

L' esempio piú familiare di stringhe sono le "parole". La parola algoritmo è una successione finita di lettere (gli elementi); con la notazione di sopra, si scriverebbe <a, l, g, o, r, i, t, m, o>.

Sempre nel genere "parole" abbiamo la rappresentazione dei numeri: successioni di cifre.

Il punto chiave per usare la notazione sintetica è la possibilità di distinguere le componenti (gli elementi), anche se non usiamo le virgole (o altro) per separarli.

Se i nostri elementi sono *simboli atomici*, come a 3 % , possiamo scriverli uno a fianco all'altro, senza usare *separatori* ed è facile, ad es. data la stringa a3% capire che è composta da tre elementi. Se

invece gli elementi non sono atomici, ad esempio sono a loro volta stringhe, la cosa può diventare problematica. Basta pensare a successioni di numeri, scritti in notazione decimale; allora: scrivendo <17, 8, 44> si capisce che è una successione di tre elementi, diversa da < 178, 44>, oppure da <178, 4, 4>, mentre se scriviamo 17844 non si capisce piú niente.

Naturalmente non c'è nulla di magico nella nostra notazione, l'unica condizione (che non avevamo precisato) è: le due parentesi acute e la virgola non compaiono nei simboli di Elem (per capire: se Elem contenesse: sia 3 che 3,3 come interpretare <3,3,3> ?).

Si può osservare che se usassimo uno stile orientale per scrivere le stringhe potremmo risolvere il problema delle successioni di numeri:

<17, 8, 44> diventerebbe 17 mentre <178, 4, 4> diventerebbe 178
8 4
44 4

Però non siamo abituati ad usare questa notazione.

Un ultimo esempio di stringhe: i programmi. Facendo l'esempio del C, un programma (ma anche una espressione, una istruzione,...) è una stringa; gli elementi, i simboli, sono i *simboli del linguaggio C* :

le lettere, le cifre, il punto, il punto e virgola, le parentesi, ... che sono simboli atomici

== && while if ..... che sono simboli non atomici.

Ricapitolando: le stringhe sono semplicemente successioni, scritte senza separatori.

Per le stringhe si usano notazioni e termini particolari:

- l'insieme Elem viene detto *alfabeto* (o anche vocabolario);
- l'insieme Succ_Elem si indica con Elem*
- la stringa vuota si indica con ε (epsilon), o con Λ (lambda maiuscolo), o con λ (lambda minuscolo)
- l'insieme delle stringhe di lunghezza (esattamente) k si indica con $Elem^k$ (in effetti, per k>0 è in corrispondenza biunivoca col prodotto cartesiano Elem $\times$ ..... $\times$ Elem); quindi si ha:

$Elem^* = \cup \{Elem^k \mid k=0\}$.

6.3 Gli stack

L'analogia piú ovvia è la pila di piatti: quando serve un piatto lo si prende dalla cima della pila, quando si aggiunge un piatto lo si poggia in cima alla pila; quindi si aggiunge e si toglie sempre da un lato e l'ultimo piatto posato è il primo ad essere usato; si dice che si segue una politica LIFO (Last In First Out). Gli staks sono molto usati nella programmazione a basso livello; quando si usa un linguaggio evoluto che permette procedure/funzioni ricorsive (e le implementa in modo efficiente) come il C è difficile che serva usarli esplicitamente (mentre sono alla base dell'implementazione della ricorsione, quindi si usano implicitamente).

Vediamo prima il **tdd stack senza limite di lunghezza**.

- Sorti: elem, stack, bool.
- Insiemi: Elem, Stack, Bool; dove: Bool = {vero, falso}, Elem = un insieme (non stiamo a precisare); Stack = Succ_Elem come definito precedentemente.
- Operazioni:

simbolo	funzionalità	intepretazione
empty	: → stack	lo stack vuoto: < >
is_empty	: stack → bool	è l'operazione vista precedentemente: is_empty(<a_1, ..., a_n>) = vero sse n=0
top	: stack → elem	è l'operazione testa vista precedentemente: top(<a_1, ..., a_n>) = a_1 indefinito su < >

pop : stack → stack è l'operazione coda vista precedentemente:
pop($<a_1, ..., a_n>$) = $<a_2, ..., a_n>$ indefinito su < >

push : elem × stack → stack è l'operazione aggiungi-in-testa vista precedentemente:
push(a, $<a_1, ..., a_n>$) = $<a, a_1, ..., a_n>$

Note.

1. Abbiamo usato empty, pop,...invece di vuoto, coda,... perchè sono i nomi piú usati in letteratura.

2. Abbiamo distinto tra simbolo, funzionalità, interpretazione, ma poi abbiamo usato top, pop, push sia per il simbolo che come nome dell'operazione.

Variante: stack di lunghezza = K (per qualche K intero >0).

Si definisce in analogia a quanto fatto per le successioni. Oltre a modificare l'interpretazione di push, per tener conto dei limiti sulla lunghezza, di solito si aggiunge l'operazione lunghezza (come nelle successioni), quindi una sorte nat cui corrisponde l'insieme **N**.

6.4 Le code

L'analogia è quella delle "file di attesa": chi arriva si mette in coda, in fondo; il primo della fila è il primo ad essere "servito". Come si vede confrontando la definizione delle operazioni, l'unica differenza tra code e stack sta nel fatto che negli stack si aggiunge e si toglie dalla stessa parte, politica LIFO (Last In First Out), mentre nelle code gli elementi vengono aggiunti da una parte e tolti dall'altra, per cui si ha una politica FIFO (First In First Out). A differenza degli staks, anche usando un linguaggio ad alto livello, può capitare di dover usare delle code; in particolare le utilizzaremo quando parleremo di *visite in ampiezza* degli alberi e dei grafi.

Il tdd code a lunghezza illimitata:

- Sorti: elem, queue, bool;
- Insiemi: Elem, Queue, Bool; dove: Bool ed Elem come sopra; Queue = Succ_Elem
- Operazioni:

simbolo	funzionalità	intepretazione
empty :	→ queue	la coda vuota: < >
is_empty :	queue → bool	è l'operazione già vista: is_empty($<a_1, ..., a_n>$) = vero sse n=0
testa :	queue → elem	è l'operazione testa già vista: testa($<a_1, ..., a_n>$) = a_1 indefinito su < >
resto :	queue → queue	è l'operazione coda: resto($<a_1, ..., a_n>$) = $<a_2, ..., a_n>$ indefinito su < >
in_coda :	elem × queue → queue	è l'operazione aggiungi in fondo: in-coda(a, $<a_1, ..., a_n>$) = $<a_1, ..., a_n, a>$

Analogamente agli stack, si definiscono anche le **code a lunghezza limitata**.

7

Implementazione delle successioni

Una successione è un oggetto matematico; sulla carta lo rappresentiamo con la notazione vista sopra, $<a_1, ..., a_n>$, oppure in qualche altro modo. Per ragionare "a livello del problema" va benissimo, ma quando poi si deve passare al livello di linguaggio C, Pascal,... bisogna trovare un modo per rappresentare le successioni. I linguaggi imperativi offrono 3 modi principali: gli array, i file e le liste dinamiche.

Per ora lasciamo stare i file e vediamo cosa si può fare con gli array e le liste dianamiche.

Gli **array** permettono di implementare solo **successioni di lunghezza limitata** (a meno che il linguaggio non offra, come il C, *array dinamici*).

Le **liste dinamiche** permettono di **implementare successioni** sia **di lunghezza limitata** che **di lunghezza illimitata.**

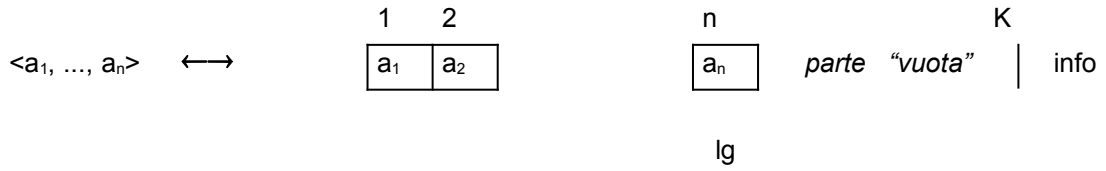
7.1 Implementazione con array

7.1.1 Implementazione degli oggetti

Se K è la lunghezza massima, una successione si può rappresentare con una coppia: (info, lg), dove

info è un array [1 .. K] of elem e lg è la lunghezza effettiva della successione, 0 = lg = K.

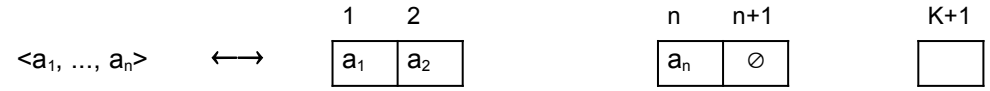
Schematicamente:



La coppia si può poi realizzare come record (struct) a due campi.

Un'implementazione alternativa è quella che si usa in C per le stringhe (= successioni di caratteri). Serve un elemento speciale, diverso da tutti quelli in Elem, indichiamolo con ⊘ (corrisponde al carattere nul che in C si indica con '/0'). Allora si usa un array di lunghezza K+1 e si indica la fine della successione con ⊘.

Schematicamente:



Correttezza dell'implementazione.

Abbiamo visto due modi per rappresentare, o implementare, le successioni; sono corretti? Per rispondere bisogna precisare cosa si intende per *correttezza di una implementazione* di un insieme di oggetti. Non vediamo una definizione formale, ma semplicemente dei *criteri di correttezza*, tra l'altro molto semplici.

Per concretezza ci riferiamo alle successioni di caratteri, ma il discorso vale in generale, anche per oggetti che non sono successioni.

- Per prima cosa, tutti gli oggetti devono essere rappresentabili. L' implementazione array+lunghezza permette di rappresentare, fissato K, tutte le successioni di lunghezza = K; non è una rappresentazione valida per tutte le successioni. La rappresentazione stile C è valida anch'essa solo per successioni di lunghezza massima fissata, ma in più richiede che ci sia nel linguaggio un tipo che contiene i caratteri che ci interessano ed anche almeno un elemento un piú.

  Notiamo che non è necessario che la rappresentazione sia univoca: non c'è niente di male se un oggetto ha tante rappresentazioni. Nel caso array + lunghezza, quello che c'e' nell'array nelle posizioni da lunghezza+1 a K non interessa; quindi, ad esempio, ci sono tante coppie (a,0) che corrispondono alla successione vuota. (Naturalmente, quando si implementano le operazioni, bisogna tenere conto di questo fatto.)

- Due oggetti distinti non devono essere identificati nella rappresentazione. questo è chiaro: un'implementazione degli interi che usasse un solo bit: 0 per i negativi, 1 per i positivi, sarebbe molto compatta, ma poco utile.

7.1.1 Implementazione delle operazioni come procedure / funzioni

Per concretezza, fissiamo Elem e sia Elem = Caratteri (quindi sia in Pascal che in C il tipo corrispondente è char). Sia sempre K la lunghezza massima delle successioni.

Ci riferiamo all'implementazione degli oggetti con array + lunghezza.

Allora: **const** K = ... *ci basta sapere che è maggiore di 0*
type Succ = **record** info **:** array [1 .. K] of char
lg **:** integer *assumerà valori tra 0 e K*
end

Vediamo l'implementazione delle operazioni, in ordine di difficoltà.

Implementazione di is_empty : Succ_Elem → {vero, falso}
is_empty(s) = vero se s = < >
falso altrimenti

Poiché ad s corrisponde una coppia (array, lunghezza), si ha che is_empy(s) = (lunghezza = 0).

Quindi

```
function    imp_is_empty (x : Succ) : boolean
                { return (x.lg = 0) }
```

Implementazione di testa : Succ_Elem → Elem
testa (< >) indefinita
testa(<a_1, ..., a_n>) = a_1 per n >0

L'implementazione è ovvia, sempre usando una function (questo perché Elem è un tipo di base, char; se fosse un tipo strutturato ci sarebbero problemi -- vedere sotto: implementazione di mk-succ), tranne che per la questione di come gestire il caso "indefinito".

Prima di tutto, osserviamo che "volere la testa di una successione vuota" non è una bella cosa; molto probabilemente corrisponde a un errore. E` importante che gli errori non passino sotto silenzio, ma vengano segnalati. Inoltre, non è la funzione/procedura che implementa l'operazione testa che può gestire l'errore. E` il programma chiamante che "sa cosa fare"; quindi nella funzione o procedura dobbiamo solo segnalare l'errore con un messaggio di errore, o qualcosa di equivalente, ed in modo "pulito".

Ci sono vari modi, ne vediamo 3.

1° modo: usare un valore speciale.

Supponiamo di sapere che le successioni "buone", cioè quelle che ci interessano, non contengano mai il simbolo # (ad esempio, perché sono successioni di sole lettere). Allora:

```
function imp1_testa (x: succ) : char
        {  if x.lg = 0  then  return ('#')  else  return (x.info[1])  }
```

Notare che non va bene scegliere come simbolo speciale qualcosa come -1, perché la funzione imp_testa deve sempre restituire un char (in effetti, in C si può usare questo trucco).

Non sempre è possibile trovare un valore speciale; un metodo che funziona sempre usa una variabile ausiliaria.

2° modo: usare una variabile booleana ausiliaria e implementare l'operazione con una procedura (invece di una funzione).

```
procedure   imp2_testa (     x : Succ              parametro IN
                             indef : boolean       parametro OUT
                              res : char           parametro OUT      )
            {    if  x.lg = 0  then  indef ← true
                 else  {  indef ← false;   res ← x.info[1] }
            }
```

Quando nel programma vogliamo la testa della successione s, chiamiamo imp_testa (s, aux, val), con aux e val variabili del tipo giusto, e poi controlliamo il valore di aux: se e` falso, sappiamo che in val troviamo la testa di s; se è vero, dobbiamo gestire la situazione di errore (come, dipende da quello che stiamo cercando di fare nel programma).

3° modo: non far niente, cioè scaricare tutta le responsabilità sul programma chiamante. Scriviamo la funzione imp_testa per implementare l'operazione testa supponendo che l'argomento non sia vuoto; quando, nel programma, vogliamo la testa della successione s, prima controlliamo se s è vuota usando la funzione imp_is_empty .....

Implementazione di mk_succ : Elem → Succ_Elem, dove mk_succ(a) = <a>.

Qui il problema è il seguente: in matematica, qualunque insieme può essere codominio di una funzione; in altre parole, non ci sono restrizioni sul tipo dei risultati delle funzioni. Nei linguaggi di programmazione spesso le restrizioni ci sono; in Pascal sono abbastanza severe: il risultato di una funzione non può essere un tipo strutturato, cioè un array, un record, ...; in C non è possibile avere risultati che sono array (ma è possibile avere risultati di tipo struct). In conclusione:

1. Se il linguaggio lo permette, usiamo una funzione.

```
function imp1_mk_succ ( ch : char ) :  Succ
            {     aux: variabile ausiliaria di tipo  Succ
                  aux.info[1] ← ch
                  aux.lg ← 1
                  return(aux)
            }
```

Per capire l'uso di questa funzione, vediamo uno schema di programma:

```
dichiarazioni    tra le altre cose:    2 variabili   cc di tipo carattere   s   di tipo Succ
                                       la funzione   imp1_mk_succ
istruzioni       .....................
                 leggi (cc)
                 s  ←  imp1_mk_succ(cc)          ora:   s = <cc>
                 .....................
```

2. Se non è possibile usare una funzione, la soluzione classica è utilizzare una procedura (questo è già stato visto, parlando di "Simulazione di funzioni usando procedure").

```
procedure   imp2_mk_succ (     ch : char          parametro IN
                               res : Succ         parametro OUT )
            {  res.info[1] ← ch  ;   res.lg ← 1  }
```

Per capire l'uso di questa procedura, vediamo uno schema di programma:

dichiarazioni tra le altre cose: 2 variabili cc di tipo carattere s di tipo Succ
la procedura imp2_mk_succ

istruzioni .....
leggi (cc)
imp2_mk_succ(cc, s); *ora: s = <cc>*
.....

Implementazione di coda : Succ_Elem → Succ_Elem
coda(< >) indefinita
coda(<a_1, ..., a_n>) = <a_2, ..., a_n> per n >0 (cioè < > se n = 1)

L'operazione coda presenta i due problemi visti prima, gestione dell'indefinito e problema del tipo del risultato, che appunto si risolvono con le tecniche viste, e un problema nuovo. Se in un programma abbiamo una variabile s di tipo Succ, ci sono due motivi per volere la sua coda:

1. vogliamo una nuova successione cs uguale alla coda di s;
2. vogliamo modificare s togliendo il 1o elemento.

Notare che in matematica il problema non si pone: coda(s) è sempre una nuova successione; il problema nasce nel contesto della programmazione imperativa, dove è centrale l'idea di "oggetti" il cui valore cambia durante l'esecuzione; il caso 2. di sopra è del tutto analogo a:
x ← x-1, con x variabile intera.

In certe situazioni è possibile definire una sola funzione/procedura in grado di gestire entrambe i casi (variando gli argomenti della chiamata). Ora vediamo una tecnica generale, che funziona sempre; quindi anche per gestire l'indefinito scegliamo il modo che funziona sempre: la variabile booleana.

Molto semplicemente definiamo due implementazioni: una che restituisce una nuova sequenza ed una che modifica l'argomento.

```
procedure  imp1_coda (   x : Succ              parametro IN
                         indef : boolean  parametro OUT
                         res : Succ            parametro OUT  )
           var  k :  integer
           {    if  x.lg = 0  then  indef ← true
                else  {  indef ← false;
                         res.lg ← x.lg-1
                         per  k = 2,...., x.lg : res.info[k-1] ← x.info[k]
           }

procedure  imp2_coda (   x : Succ              parametro IN-OUT
                         indef : boolean  parametro OUT  )
           var  k :  integer
           {    if  x.lg = 0  then  indef ← true
                else     {    indef ← false;
                              per  k = 2,...., x.lg  :  x.info[k-1] ← x.info[k]
                              x.lg ← x.lg-1
                         }
           }
```

Notare che imp1_coda si può anche scrivere come funzione in un linguaggio che permette di restituire tipi strutturati come risultati, mentre imp2_coda ha proprio le caratteristiche di procedura.

A questo punto, siamo in grado di scrivere le implementazioni di tutte le altre operazioni non costanti: lo lasciamo **per esercizio**. Restano solo le costanti.

Implementazione delle (operazioni) costanti.

Nel caso degli interi, non ci sono problemi: il linguaggio ci permette di esprimerle direttamente; se ci serve il numero diciassette scriviamo semplicemente 17.

Nel caso dei booleani, in Pascal è la stessa cosa (ci sono le costanti true e false), in C c'è una codifica standard: 0 per falso, 1 (oppure qualunque intero non zero) per vero.

Nel caso delle successioni, abbiamo visto che:

$<a_1, ..., a_n>$ corrisponde ad una coppia (array, lunghezza), con lunghezza = n e array che contiene a_1 nella prima posizione, a_2 nella seconda,.....

Non sempre i linguaggi permettono di scrivere direttamente dei valori che sono degli array o dei record. Qualcosa si può fare, ma non tutto. Allora quello che si fa in genere è: definire l'implementazione di poche costanti, di base, e costruire le altre a partire da queste, usando i costruttori, cioè operazioni come mk_succ, conc,.....

Nel caso delle successioni, la scelta piú frequente è quella di fornire l'implementazione solo della costante successione vuota: < > . Questa corrisponde ad una coppia (array, lunghezza), con lunghezza = 0 e array che contiene ... quello che capita.

Per capire il modo in cui realizza l'implementazione, è opportuno capire come useremo la costante in un programma; i casi sono essenzialmente due:

1. abbiamo una variabile s di tipo succ e vogliamo assegnarle il valore < > (a livello di algoritmo astratto scriveremmo semplicemente s ← < > ;)
2. abbiamo una variabile s di tipo succ e ci chiediamo se il suo valore è < >.

Per 2. abbiamo l'operazione is_empty che è fatta apposta.

Resta quindi la 1. Un metodo "sempre applicabile" è di usare una procedura:

```
procedure   imp_empty ( x : Succ        parametro OUT )
                { x.lg ← 0 }
```

Quindi, quello che, a livello di algoritmo astratto era semplicemente s ← < > diventa una chiamata di procedura: imp_empty(s).

Nota.

Molte delle procedure/funzioni viste (ad esempio imp_empty) si riducono ad una o due istruzioni. Ci si può chiedere se abbia senso scrivere delle procedure cosí semplici. Nel caso di imp_empty lo scopo è di realizzare l'assegnazione s ← < >; allora: ha senso definire una procedura quando basterebbe scrivere: s.lg ← 0 ? Sembrerebbe di no; tuttavia ....

Qui bisogna distinguere tra programmi "corti", ad esempio una o due pagine di codice, sviluppati da una sola persona, e programmi "lunghi", sviluppati da tante persone, ciascuna incaricata di una parte. Fintanto che si può tenere tutto sott'occhio, come nel caso di due pagine di codice, non si vede l'interesse per procedure come imp-empty, ma anche di molte altre procedure. Il fatto è che in questo caso non c'è una reale necessita` di isolare le varie parti del programma.

Al contrario, quando si deve portare avanti un progetto complesso si deve operare in un altro modo. Quando abbiamo discusso merge sort, abbiamo visto brevemente la fase di "merge"; poi, una volta capito che si poteva fare, abbiamo fatto tutto il resto senza preoccuparci di *come* veramente sarebbe stata realizzata la fase di merge; ci bastava sapere *cosa* faceva merge. Nello sviluppo di un progetto complesso si estende questo approccio; quindi:

- si divide il problema, e dunque la sua soluzione, in parti sufficientemente autonome;
- per ciascuna parte si precisa che *cosa si aspetta* dalle altre e che *cosa fornirà* alle altre, senza precisare *come* le cose saranno fatte;

- si realizzano le varie parti attraverso procedure o insiemi di procedure (questo in C o Pascal; in altri linguaggi si usano anche “moduli”, “classi”, “oggetti”, ...).

Ora supponiamo che nel progetto servano le successioni. Spesso non è chiaro all’inizio: quanto serviranno, con quali operazioni, quale implementazione sia la migliore,.... Allora: ci si mette d’accordo sul fatto che si userà un tipo di dato Successioni; man mano che servono si precisano le operazioni,.... Ciascuno sviluppa la sua parte di codice usando le procedure/funzioni su cui ci si è messi d’accordo,.... Quando è il momento si precisa l’implementazione delle successioni e delle operazioni. Fatto questo, NON c’è bisogno di rivedere le varie parti per sistemare le cose riguardo alle successioni: sono già a posto ! Se poi, in un secondo momento, si decide di cambiare l’implementazione delle successioni, si deve riscrivere SOLO la parte di programma in cui sono definite le successioni, NON le parti che le usano.

Il C fornisce uno metodo abbastanza semplice, per separare le informazioni che tutti devono sapere (i nomi dei tipi ed i prototipi delle funzioni e, tramite commenti, una descrizione di quello che fanno), da quelle che non è necessario conoscere (in particolare: la struttura interna dei tipi e l’implementazione delle funzioni). Questo aspetto del C verrà visto nel corso; la questione generale, invece, verrà affrontata nel corso di *Linguaggi di programmazione* ed in corsi successivi.

Un esercizio per “mettere tutto assieme” (e chiarire il discorso delle costanti). Vogliamo una procedura che:

legge n, c1,...cn

se n>K, lo segnala ed esce

se n=K, carica c1,...cn in una variabile di tipo Succ

In pratica questa procedura è una mk_succ generalizzata.

Per realizzare la procedura si può procedere in due modi:

1. usando le funzioni/procedure che implementano < >, mk_succ, conc,...
2. ignorandole e lavorando direttamente sull’implementazione degli oggetti successioni

Qui non discutiamo quale delle due sia la strada migliore; la questione è legata al discorso di sopra.

L’esercizio è: provare a realizzare la procedura nei due modi.

Correttezza dell’implementazione delle operazioni.

Ci sono delle difficoltà a formulare dei criteri generali di correttezza per l’implementazione di operazioni; questo è dovuto al fatto che, come abbiamo visto, ci sono tante varianti: una operazione si può implementare con una funzione o con una procedura; ci sono vari modi di gestire le situazioni di risultato non definito; infine, non va dimenticato il fatto che l’implementazione degli oggetti non è univoca.

Con gli strumenti che abbiamo a disposizione, quello che possiamo fare è solo affrontare la questione caso per caso.

Per un esempio, consideriamo l’implementazione della funzione eq (l’uguaglianza) che non abbiamo discusso. Poichè il risultato è un booleano, useremo una funzione

function imp_eq (x, y : Succ) : boolean

Supponiamo di avere due variabili imp_s1 ed imp_s2, con valori, rispettivamente, s1 ed s2 (o meglio: l’implementazione di s1 ed s2). Quello che vogliamo è:

eq(s1, s2) = b (dove b = vero, falso) sse imp_eq(imp_s1, imp_s2) = imp_b

dove imp_b è l’implementazione di b.

In particolare, se ad esempio imp_s1.lg = imp_s2.lg = 0, il risultato deve essere “vero”, anche se imp_s1.info e imp_s2.info sono diversi come array !

7.2 Implementazione con liste dinamiche

7.2.1 Implementazione degli oggetti

Le liste dinamiche si costruiscono “mettendo in fila” una sequenza di record allocati dinamicamente. Ciascuno dei record contiene un campo puntatore che gli consente di agganciarsi al successivo della fila.

Per fare una lista di oggetti di tipo elem abbiamo bisogno di definire un ulteriore tipo che possiamo pensare come “cella della lista atta a contenere un valore di tipo elem”:

```
Type   Cell =  record  info : elem
                       next : puntatore_a_Cell
               end
       Succ  = puntatore_a_Cell
```

Se consideriamo una variabile s di tipo Succ possiamo avere tre casi:

- s è indefinita → in questo caso non rappresenta niente, va inizializzata
- s vale **null** → in questo caso rappresenta la successione vuota
- s vale l’indirizzo di una cella esistente → in questo caso rappresenta la successione che incomincia da quella cella.

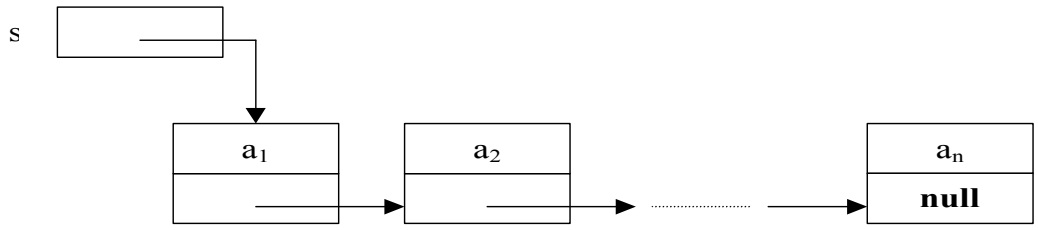
Nel terzo caso, il primo elemento della successione è quello contenuto nel campo info della cella puntata da s.

Come facciamo a trovare gli altri elementi della successione e a sapere dove finisce la successione? È necessario usare il campo next per “agganciare” ogni cella alla successiva della lista:

Nel campo next di ogni cella si mette un puntatore (indirizzo) alla cella successiva.

Nel campo next dell’ultima cella si mette il valore **null.**

In questo modo è possibile passare dalla prima cella alla seconda, dalla seconda alla terza e così via. Quando si trova la cella con il campo next a **null** si capisce che la lista è finita.



7.2.2 Implementazione delle operazioni come procedure / funzioni

Come nel caso precedente assumiamo che il tipo Elem equivalga a char. Le implementazioni per altri tipi sono analoghe.

Implementazione di Is_Empty : Succ_Elem → {vero, falso}
Is_Empty(s) = vero se s = < >
falso altrimenti

Dal paragrafo precedente sappiamo che la successione vuota ~e rappresentata con un valore **null.** Quindi:

```
function    imp_is_empty (x : Succ) : boolean
            { return (x = null) }
```

Implementazione di testa : Succ_Elem → Elem
testa (< >) indefinita
testa(<a_1, ..., a_n>) = a_1 per n >0

Per quanto riguarda la gestione delle eccezioni (successione vuota) vale il discorso fatto per le successioni a lunghezza limitata. Qui assumiamo di disporre di una costante NULL_CHAR che corrisponde all'elemento indefinito (primo modo visto in precedenza). Allora:

```
function    imp_testa (x : Succ) : char
            {   if  (x = null) then return (NULL_CHAR)
                else return (x->info)
            }
```

Implementazione di mk_succ : Elem → Succ_Elem, dove mk_succ(a) = <a>.

Come nel caso delle successioni limitate si può implementare come funzione o come procedura. In C sono possibili entrambe le implementazioni:

Primo modo:

```
Function    imp1_mk_succ ( el: char ) :  Succ
            {   aux: variabile ausiliaria di tipo  Succ
                new(aux)        è necessario allocare una nuova cella
                aux->info ← el
                aux->next ← null
                return(aux)
            }
```

Attenzione: anche se la variabile aux diventa indefinita quando si esce dalla procedura, la cella allocata continua ad esistere. Siccome la funzione restituisce l'indirizzo di quella cella, il risultato della funzione corrisponde appunto ad una lista di un unico elemento che incomincia e finisce in quella cella.

Secondo modo:

```
procedure   imp2_mk_succ (    el : char              parametro IN
                              res : Succ        parametro OUT )
            {   new(res)
                res->info ← ch
                res->next ← null  }
```

Attenzione: in C bisogna stare attenti a come si dichiara il parametro res. Dato che è in modalità OUT, bisognerà dichiarare `Succ * res` (cioè res è un puntatore al tipo Succ) e poi

Nel codice bisognerà mettere:

```
*res = malloc( sizeof ( Cell ) );
*res->info = ch;
*res->next = null
```

Implementazione di coda : Succ_Elem → Succ_Elem
coda(< >) indefinita
coda(<a_1, ..., a_n>) = <a_2, ..., a_n> per n >0 (cioè < > se n = 1)

Valgono le considerazioni fatte nel caso dele successioni limitate per il valore indefinito e per il fatto di modificare o meno la successione in input. Vediamo solo un'implementazione mediante una procedura che:

- se l'imput è vuoto restituisce errore in un booleano
- altrimenti modifica l'input eliminando il primo elemento

```
procedure   imp_coda ( x : Succ                parametro IN-OUT
                       indef : boolean  parametro OUT  )
            var  y : Succ
            {    if  x = null  then  indef ← true
                 else     {    y ← x
                               x ← x->next
                               free(y)       è necessario liberare la cella corrispondente
                                             alla testa che "buttiamo via"
                          }
            }
```

Notare che l'operazione è molto più semplice che nel caso degli array perché non dobbiamo scandire tutta la lista per fare arretrare gli elementi di un posto: basta "staccare" l'elemento in testa e spostare avanti il puntatore.

Implementazione della costante vuota: empty : → Succ_Elem dove empty() = < >

```
function    imp1_empty (  ) : Succ
            {  return ( null ) }
```

oppure come procedura

```
procedure   imp2_empty ( x : Succ   parametro OUT )
            {  x ← null  }
```

7.3 Confronto tra le due implementazioni

Ovviamente l'implementazione con liste ha il vantaggio di essere utilizzabile per liste di lunghezza arbitraria, mentre quella con array è limitata alle liste di lunghezza limitata a priori.

Nota: In C questo non è del tutto vero perché è possibile un'implementazione che gestisce liste di lunghezza arbitraria mediante array dinamici e una tecnica "a blocchi". Queste cose non le vedremo.

Questo comunque non significa che l'implementazione con liste sia sempre da preferire, si sono vantaggi e svantaggi, secondo le operazioni che si vogliono fare. Vediamone alcune:

- **Aggiunta di un elemento in testa:** con le liste dinamiche è più efficiente perché basta generare una nuova cella e "agganciarla" all'inizio della lista. Con gli array bisogna spostare tutti gli elementi già presenti avanti di un posto.
- **Aggiunta di un elemento in coda:** con gli array è più facile perche basta aumentare la lunghezza di uno e scrivere il nuovo elemento nella cella corrispondente. Con le liste bisogna scandire tutta la lista per trovare la cella finale e aggiungere lì la nuova cella.
- **Coda:** se si genera una nuova successione il problema è pressoché equivalente in entrambi i casi, con le liste c'è in più la noia di allocare tutte le celle della lista nuova, mentre con gli array si alloca in un

colpo solo. Se invece si modifica la successione esistente l'implementazione con liste è più efficace perché basta sganciare il primo elemento invece di spostare tutti gli altri indietro di un posto (visto sopra).

- **Lunghezza:** con l'array è immediata (scritta nel record) mentre è necessario scandire tutta una lista per contarne gli elementi.
- **Selezione del k-esimo elemento:** con gli array è immediato accedere alla cella con indice k, mentre con le liste bisogna scandire dall'inizio contando fino a k.
- **Scansione:** questa è un'operazione generale che può intervenire nell'implementazione di diverse operazioni del tipo di dato. Si tratta in pratica di navigare sulla lista muovendosi da un elemento al successivo o al precedente. Con gli array si può fare liberamente. Con le liste è solo possibile la scansione in avanti, non è possibile muoversi all'indietro.

Alcuni di questi problemi per le liste sono rimediabili facilmente. In particolare, pa parte gli ultimi due, gli altri si rimediano cambiando l'implementazione del tipo Succ:

```
Type   Succ = record
                 first :   puntatore_a_Cell   punta alla prima cella della lista
                 last :    puntatore_a_Cell   punta all'ultima cella della lista
                 lg   :    integer            mantiene la lunghezza della lista
               end
```

Ovviamente l'implementazione delle varie operazioni dovrà essere fatta in modo da assegnare in modo consistente i valori ai tre campi in corrispondenza di ogni operazione.

Esercizio: implementare le primitive conc_c, conc, lunghezza nei due casi (Succ definito come puntatore e Succ definito come record).

Il problema della scansione può essere risolto cambiando il tipo Cell. Si aggiunge in ogni cella un altro campo puntatore che punta alla cella precedente: anche qui è necessario cambiare l'implementazione di conseguenza. In questo caso si parla di **liste doppiamente linkate.**

Il problema di selezionare elementi in posizione arbitraria è intrinseco delle strutture dinamiche e non si può rimediare: è sempre necessaria la scansione.

8 Tipi di dato astratti e concreti, strutture dati

8.1 Ricapitolando ...

Per noi: **tdd = tipo di dato = tipo di dato concreto**.

Un **tdd concreto** è una coppia: < segnatura, algebra sulla segnatura >, dove la segnatura, e quindi l'algebra, è eterogenea e le operazioni sono parziali.

Un **tdd astratto** è una coppia < segnatura, classe di algebre sulla segnatura >. Tale classe è definita, in genere, specificando un elenco di proprietà che devonno essere soddisfatte dalle algebre della classe. Questo argomento verrà affrontato in altri corsi, qui solo due parole.

Nel corso di algebra è stato visto il concetto di isomorfismo; questo si estende al caso dellle nostre algebre. Bene: due algebre isomorfe, sono, da un certo punto di vista, indistinguibili; quindi se una certa algebra A va bene per descrivere, ad esempio, gli stack e B è isomorfa ad A, allora anche B va bene per gli stack. Ecco un primo modo di definire un tipo di dato astratto: la classe formata da un'algebra A e da tutte quelle isomorfe ad A. Si può andare oltre: anche algebre non isomorfe possono essere "abbastanza simili per quelli che sono i nostri scopi"; ma noi ci fermiamo qui.

All'interno dei tipi di dato concreti, è utile distingure tra

- tipi di dato **a livello di problema**, o di soluzione ad alto livello del problema, ad esempio: interi, reali, schede anagrafiche, archivi, successioni, alberi, grafi,....
- tipi di dato **a livello di linguaggio** di programmazione; ci sono quelli di base, cha approssimano gli interi, i reali,..., e quelli strutturati e costruiti dal programmatore: record, array, files, liste,... (Ricordiamo che non c'è un tipo di dato array, ma tanti tipi di dato, analogamente per i record, i file,...).

  In effetti, i tipi di dato a livello di linguaggio di programmazione raramente vengono presentati come tali, cioè specificando segnatura ed algebra (anche perchè vengono fuori della algebre un po' complicate), e questo porta a chiamarli in altro modo.

  In particolare, alcuni usano:

- "tipo di dato astratto" per i tipi di dato concreti a livello di problema;
- "tipo di dato concreto" per i tipi di dato concreti a livello di linguaggio.

Infine: il termine **stuttura dati** si usa in genere, in modo informale e quindi generico, per indicare *gli elementi* dei tipi di dato concreti a livello di liguaggio, quando sono (appunto) strutturati, come nel casi degli array, dei record, delle liste, dei files,....

8.2 Implementazione di un tipo di dato (solo uno schema)

Schema:

- a livello di problema e algoritmo, abbiamo un tdd
- a livello di programma, abbiamo l'implementazione del tdd (nel linguaggio di programmazione che usiamo)

  Ad esempio, per risolvere un certo problema usiamo un algoritmo dove servono due code limitate di numeri razionali, c1 e c2.

  Quindi, l'algoritmo usa il tdd coda con 4 sorti: razionali (che precisa la sorte elem), bool, naturali (perchè usaremo anche l'operazione lunghezza) code.

  Nel programma C che realizza l'algoritmo avremo (se usiamo un solo file):

- un tipo per ogni sorte: `float` per razionali, `int` per naturali, `int` per bool (a meno di non definire un tipo `Bool` con un `typedef`) ed un tipo `Code` per code, definito in qualche modo

  (ad esempio: `typedef struct { .....} Code;`).
- la dichiarazione delle funzioni (o procedure) che implementano le operazioni per le code;
- due variabili `c_1` e `c_2` di tipo `Code` per le due code c1 e c2.

  Alternativamente, il programma può essere organizzato usando più files:

- un file header `imp_code.h`
- un file con il codice per le code `imp_code.c`
- un file con il programma (che include `imp_code.h`).

9 Definizioni induttive

Le definizioni induttive sono un modo di definire insiemi; come caso particolare ci sono le definizioni induttive di relazioni e funzioni (che sono degli insiemi).

Vedremo alcuni esempi, ma non daremo la definizione di "definizione induttiva"; formalizzare le cose non è molto difficile, ma richiede almeno qualche ora di lezione, e non abbiamo il tempo per farlo (quelli che senza

definizioni precise non possono vivere, possono farsi prestare le dispense del corso di Programmazione 1995/96).

9.1 Definizioni induttive "semplici"

Esempio 1. Espressioni costruite usando: lettere, +, *, parentesi tonde.

Se proviamo a definire "a parole" come sono fatte le espressioni aritmetiche solite (quelle delle scuole medie) ci accorgiamo che non è facile. Si ricorre allora a definizioni induttive.

Sia Lettere l'insieme delle lettere e Alfa = Lettere $\cup$ {+, *, (,) }; vogliamo definire Exp, l'insieme delle nostre espressioni. Una possibile def induttiva è la seguente

Def_1

Exp è l'insieme di stringhe su Alfa definito [induttivamente] da:

base	1)	se	$x \in$ Lettere	allora $x \in$ Exp
passo	2)	se	exp1, exp2 $\in$ Exp,	allora exp1 + exp2 $\in$ Exp
	3)	se	exp1, exp2 $\in$ Exp,	allora exp1 * exp2 $\in$ Exp
	4)	se	exp $\in$ Exp,	allora (exp) $\in$ Exp

Note.

- Qui sopra abbiamo dato la *forma* della def. induttiva, il *significato* lo dobbiamo ancora spiegare.
- Qui ed in seguito, **gli spazi sono irrilevanti**; quindi b + a*c è uguale a b+a *c
- Qui ed in seguito, **l'ordine con cui si scrivono le clausole è irrilevante**; anche se c'è l'uso di mettere prima la base e poi (in ordine arbitrario) le clausole del passo.
- Ripetiamo che questa è solo una delle possibili def; vedremo meglio più avanti.
- Alle volte si scrive più concisamente:
  1) Lettere $\subseteq$ Exp
  2) exp1, exp2 $\in$ Exp $\Rightarrow$ exp1 + exp2 $\in$ Exp eccetera
- Qui Lettere è un insieme finito, ma nella base si possono anche avere insiemi infiniti.

Il significato intuitivo di Def_1 è:

> Exp è il più piccolo insieme di stringhe che contiene le lettere (cioè contiene gli oggetti specificati nella clausola di base) e gli oggetti che si ottengono applicando un numero finito di volte le clausole del passo, cioè le regole 2), 3), 4), agli oggetti già costruiti.

Più precisamente: l'insieme definito da Def_1 è $Exp = \cup \{ Exp_k \mid k = 0 \}$ dove

Exp_0 = Lettere [quindi: Exp_0 è quello definito dalla base]

Exp_{k+1} = { exp1 + exp2 | exp1, exp2 $\in$ ($Exp_0 \cup ... \cup Exp_k$) } $\cup$
{ exp1 * exp2 | exp1, exp2 $\in$ ($Exp_0 \cup ... \cup Exp_k$) } $\cup$
{ (exp) | exp $\in$ ($Exp_0 \cup ... \cup Exp_k$) }
[il passo ci dice come Exp_{k+1} viene costruito a partire da Exp_0 .... Exp_k]

E` abbastanza evidente, dalla definizione, che $Exp_0 \subset Exp_1 \subset Exp_2 \subset Exp_3$ ... (qui l'inclusione è stretta, ma non è detto che sia sempre così).

In effetti c'è una def. alternativa per gli insiemi Exp_k che è forse meno intuitiva della precedente, ma più comoda e sintetica (nel seguito, useremo questo "tipo" di definizione):

Exp_0 = Lettere [come sopra]

Exp_{k+1} = Exp_k $\cup$ { exp1 + exp2 | exp1, exp2 $\in$ Exp_k }
$\cup$ { exp1 * exp2 | exp1, exp2 $\in$ Exp_k }
$\cup$ { (exp) | exp $\in$ Exp_k }

Quindi: Exp_0 = { a, b, c, ...}

$Exp_1 = Exp_0 \cup$ { a+a, a+b, ..., b+a, b+b,.....
a*a, a*b, ...
(a), ..., (d), ... }

$Exp_2 = Exp_1 \cup$ { a + a+a, a + a+b ....., a + c*d ,..... b + d*c ..... a+a + a+b,.....
a * a+a, a * a+b ....., a * c*d ,..... b * d*c ..... a+a * a+b,.....
(a+a), (a+b)....., (c*d) ,..... }

Ambiguità.

Ci sono stringhe, come a * b + c che si possono ottenere in due modi:

a * b + c oppure a * b + c

Stringhe più lunghe, esempio z * b + d * d + e + w, si possono ottenere in tanti modi.

Questo fenomeno va sotto il nome di *ambiguità*. L'ambiguità può creare problemi quando cerchiamo di interpretare le stringhe, per esempio per valutarle: se interpretiamo * come prodotto e + come somma, a*b+c costituisce un problema (a meno di non imporre delle regole di precedenza). In altri casi il problema non c'è: anche a + b + c è ambigua, ma se interpretiamo il + come somma, l'associatività della somma ci salva.

Giocando con le definizioni induttive si può ridurre il problema dell'ambiguità, ad esempio usando le parentesi in modo più mirato. Se pensiamo all'interpretazione standard: * indica il prodotto e + la somma, una definizione che elimina il problema delle stringhe tipo a*b+c è la seguente.

Def_2

Exp è l'insieme di stringhe su Alfa definito [induttivamente] da:

base 1) se $x \in$ Lettere allora $x \in$ Exp

passo 2) se exp1, exp2 $\in$ Exp, allora (exp1 + exp2) $\in$ Exp

3) se exp1, exp2 $\in$ Exp, allora exp1 * exp2 $\in$ Exp

L'insieme definito è $Exp = \cup \{ Exp_k \mid k = 0 \}$ dove:

Exp_0 = Lettere

$Exp_{k+1} = Exp_k \cup \{ (exp1 + exp2) \mid exp1, exp2 \in Exp_k \}$
$\cup \{ exp1 * exp2 \mid exp1, exp2 \in Exp_k \}$

Sintassi e semantica.

Gli insiemi definiti da Def_1 e Def_2 sono diversi. Per non fare confusioni, chiamiamo Exp_1 ed Exp_2 i due insiemi. Allora Exp_2 è strettamente contenuto in Exp_1; non ci sono più stringhe della forma: (a), (b),....; non ci sono più stringhe come a*b+c (ma ci sono ancora stringhe del tipo a*b*c che non ci preoccupano perché il prodotto è associativo); non si può piú scrivere: a+b e a+b+c ma si deve scrivere (a+b) e (a+(b+c)), oppure ((a+b)+c).

Però, in base all'interpretazione che abbiamo detto, i due insiemi sono "equivalenti come potere espressivo": tutto ciò che riusciamo ad esprimere usando le stringhe di Exp_1 lo possiamo esprimere usando quelle di Exp_2. In più, come già detto, in Exp_2 non ci sono più ambiguità pericolose.

E' molto importante non far confusione tra i due aspetti, cioè tra *sintassi* (come si scrive) e *semantica* (cosa vuol dire). Dal punto di vista sintattico, (b+c) e b+c sono diversi; in base alla semantica che abbiamo in mente, in cui le parentesi servono solo a raggruppare, le due esperessioni sono del tutto equivalenti. Se però cambiassimo l'interpretazione delle parentesi, ad esempio dicendo che (exp) si interpreta come "exp al cubo", le due stringhe di prima sarebbero diverse anche semanticamente.

Detto questo, bisogna anche dire che, il più delle volte, la definizione sintattica è guidata da considerazioni semantiche: in base a quello che voglio esprimere, preciso il modo in cui si scrive. Questo è precisamente quello che abbiamo fatto quando abbiamo dato la Def_2, scegliendo un modo furbo di mettere le parentesi.

Un altro esempio per chiarire questo punto, dove Cifre = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} ∪ { + }

Def_3

Exp_3 è l'insieme di stringhe su Cifre ∪ { + } definito [induttivamente] da:

base 1) se x ∈ Cifre allora x ∈ Exp_3

passo 2) se exp1, exp2 ∈ Exp_3, allora exp1 + exp2 ∈ Exp_3

Exp_3 contiene 7+3 ma non contiene 10, infatti non c'è nessun modo di formare la stringa "10" usando le clausole 1) e 2). D'altra parte, se interpretiamo le cifre in modo standard ed il simbolo + come somma, la stringa 7+3 ci fornisce effettivamente il modo di esprimere il numero dieci.

Per avere stringhe come 10 ci vuole una definizione più complicata: prima dobbiamo definire i numeri e poi le espressioni.

Def_4_a

Num è l'insieme di stringhe su Cifre definito [induttivamente] da:

base 1) se x ∈ Cifre allora x ∈ Num

passo 2) se num ∈ Num, x ∈ Cifre e x? 0 allora x num ∈ Num

(x num è la concatenazione di x con num)

Def_4_b

Exp_4 è l'insieme di stringhe su Cifre ∪ { + } definito [induttivamente] da:

base 1) se x ∈ Num allora x ∈ Exp_4

passo 2) se exp1, exp2 ∈ Exp_4, allora exp1 + exp2 ∈ Exp_4

Ci sono altri casi in cui capita di dover definire più insiemi; non sempre si possono definire separatamente come sopra: prima Num e poi Exp usando Num. Spesso è necessario definire i diversi insiemi "in parallelo"; si hanno quindi le definizioni induttive multiple.

9.2 Definizioni induttive multiple

Un esempio importante viene sempre dalle espressioni. In matematica e nei linguaggi di programmazione ci sono delle convenzioni, o *regole di precedenza*; il caso più familiare è proprio quello di a*b+c che si interpreta come se fosse scritto (a*b)+c; si dice anche che "il * ha precedenza sul +", oppure che "il * lega più strettamente del +". È possibile "inserire" regole di questo tipo dentro le definizioni induttive, ma per fare questo bisogna definire diversi insiemi. Vediamo proprio come inserire nelle espressioni con lettere e +, *, la precedenza del * sul +. La definizione che diamo è la formulazione in stile induttivo di quella che usa il Pascal per le espressioni aritmetiche (semplificandola).

A partire da Lettere (l'insieme delle <lettere>), si definiscono, in parallelo, tre insiemi: Fatt (l'insieme dei <fattori>), Term (l'insieme dei <termini>), Exp (l'insieme delle nostre <espressioni>).

Non cerchiamo di chiarire come si arriva a concepire una simile definizione, semplicemente la scriviamo e spieghamo cosa significa.

Ricordiamo che Alfa = Lettere ∪ {+, *, (,) }.

Def_5

Fatt, Term, Exp sono gli insiemi di stringhe su Alfa definiti [induttivamente] da:

1) Lettere ⊂ Fatt [una lettera è un fattore]
2) exp ∈ Exp ⇒ (exp) ∈ Fatt [una espressione tra parentesi è un fattore]
3) Fatt ⊂ Term [un fattore è un termine]
4) term ∈ Term, fatt ∈ Fatt ⇒ term * fatt ∈ Term
   [il prodotto di un termine ed un fattore è un termine]
5) Term ⊂ Exp [un termine è una espressione]
6) exp ∈ Exp, term ∈ Term ⇒ exp + term ∈ Exp

[la somma di una espressione ed un termine è una espressione]

Note.

- Nella 4) si poteva anche rovesciare l'ordine: mettere prima il fattore e poi il termine; il punto chiave è: si ammette solo la forma term * fatt oppure solo la forma fatt * term ma non tutte e due; analogamente per la clausola 6).
- Ricordiamo che Lettere $\subset$ Fatt è equivalente a lett $\in$ Lettere $\Rightarrow$ lett $\in$ Fatt
- Dove sono finiti **base e passo** ? In ogni clausola c'è una parte destra (a destra di $\Rightarrow$, oppure di $\subset$) ed una parte sinistra (a sinistra di $\Rightarrow$, oppure di $\subset$). Una clausola è di base, se "quello che c'è a sinistra è completamente noto". La 1) è una clausola di base, perchè a sinistra c'è solo Lettere, che è noto; in altre parole: per usare la 1) basta conoscere l'insieme Lettere e lo conosciamo. Tutte le altre clausole fanno parte del passo, perchè a sinistra contengono uno degli insiemi che stiamo definendo. Questo discorso è del tutto generale, non lo abbiamo fatto prima, perchè negli esempi precedenti le cose erano così chiare (???) che sarebbe stata una perdita di tempo.

Notiamo che un modo di leggere Def_5 (andando da destra verso sinistra) è:

i) un <fattore> è una <lettera> oppure una <espressione> tra parentesi

ii) un <termine> è un <fattore> oppure un prodotto: <termine> * <fattore>

iii) una <espressione> è un <termine> oppure una somma: <espressione> + <termine>

Mettendo tutto assieme e ricordando che si era detto di costruire i tre insiemi in parallelo, dovrebbe essere abbastanza chiaro che

Exp = $\cup$ { Exp_k | k =0 } Term = $\cup$ { $Term_k$ | k =0 } Fatt = $\cup$ { $Fatt_k$ | k =0 } dove:

Exp_0 = Lettere

$Term_0 = Fatt_0 = \varnothing$ [infatti non ci sono clausole di base per Term e Fatt]

$Fatt_{k+1}$ = $Fatt_k \cup \{ (e) \mid e \in Exp_k \}$

$Term_{k+1}$ = $Fatt_k \cup Term_k \cup \{ t * f \mid t \in Term_k , f \in Fatt_k \}$

Exp_{k+1} = $Term_k \cup Exp_k \cup \{e + t \mid e \in Exp_k , t \in Term_k \}$

Notare che nella def. di $Term_{k+1}$ la parte $Fatt_k$ viene dalla clausola 3), la parte $Term_k$ viene dall'idea di accumulare sempre quello che si è già costruito nei passi precedenti, mentre il resto viene dalla clausola 4). Discorso analogo per Exp_{k+1}.

Per i curiosi.

Diamo un'idea del perché la Def_5 ingloba la regola di precedenza del * sul +, discutendo il solito esempio: a*b+c. Usiamo la def induttiva "a ritroso" per vedere se a*b+c è una espressione e, in tal caso, come è stata costruita.

a*b+c $\in$ Exp sse ([1] oppure [2]) dove:

[1] a*b+c è ottenuta usando la regola 5); cioè: a*b+c $\in$ Term

[2] a*b+c è ottenuta usando la regola 6); cioè: a*b $\in$ Exp e c $\in$ Term

E` facile vedere che effettivamente a*b $\in$ Exp e c $\in$ Term, quindi la [2] è vera.

La [2] dice: a*b+c si ottiene come somma di a*b con c, cioè che è equivalente a (a*b)+c.

Vediamo se la strada [1] è percorribile.

a*b+c $\in$ Term sse ([3] oppure [4]) dove:

[3] a*b+c $\in$ Fatt

[4] a $\in$ Term e b+c $\in$ Fatt

E` facile vedere che: la [3] non è possibile; ma anche la [4] non è possibile, perchè non è possibile che b+c $\in$ Fatt. Quindi la [1] non è possibile e la stringa a*b+c si può costruire solo in un modo, quello che corrisponde a dire che è una somma.

9.3 Definizioni induttive usando "pattern matching"

La Def_5 segue lo stile "classico" delle definizioni induttive; però è piuttosto pesante, da leggere e da scrivere. Per questo motivo spesso si preferisce uno stile più sintetico, detto per *pattern matching*, che ricorda molto (per chi lo conosce) lo stile BNF.

Sottolineamo che le definizioni per pattern matching sono solo un modo alternativo di scrivere delle definizioni induttive; una volta spiegata la corrispondenza fra i due stili, non c'è niente da aggiungere; in particolare non bisogna spiegare quali insiemi definiscono (e riprendere la tiritera delle successioni di insiemi).

Una strada per arrivare allo stile per pattern matching è riprendere il modo di leggere la Def_5, proposto precedentemente:

i) un <fattore> è una <lettera> oppure una <espressione> tra parentesi

ii) un <termine> è un <fattore> oppure un prodotto: <termine> * <fattore>

iii) una <espressione> è un <termine> oppure una somma: <espressione> + <termine>

Si tratta solo di sintetizzare e schematizzare questo modo di vedere le cose.

Def_6

Fatt, Term, Exp sono gli insiemi di stringhe su Alfa definiti [per pattern matching] da:

i) fatt **::=** lett **|** (exp)

ii) term **::=** fatt **|** term1 * fatt

iii) exp **::=** term **|** exp1 + term

dove lett è un generico elemento di Lettere, fatt è un generico elemento di Fatt, term e term1 sono generici elementi di Term,.....

Note.

- La barra **|** si legge "oppure" e separa le varie *alternative*, mentre **::=** si legge "è una delle cose seguenti".
- L'ordine tra le alternative è irrilevante; come pure irrilevante è l'ordine in cui si scrivono le clausole: le righe i), ii), iii) si possono mettere in un ordine qualunque.
- Inoltre, ogni alternativa va presa singolarmente: scrivere

  ii) term **::=** fatt **|** term1 * fatt

  è equivalente a scrivere

  iia) term **::=** fatt

  iib) term **::=** term1 * fatt

  quindi non c'è nessun legame tra il "fatt" di iia) e quello di iib).
- All'interno di una stessa alternativa, invece, se usiamo lo stesso nome intendiamo lo stesso oggetto; ad esempio, scrivendo

  term **::=** fatt * fatt

  intendiamo dire che bisogna prendere due copie dello stesso fattore e metterci un * in mezzo.

Vediamo ancora un paio di esempi:

Come primo esempio, aggiungiamo a quanto sopra delle espressioni booleane semplici (con tante parentesi per evitare ambiguità)

Il nuovo alfabeto è Beta = Alfa ∪ { = , and, or, not }

Def_7

Fatt, Term, Exp, Bexp sono gli insiemi di stringhe su Beta definiti [per pattern matching] da:

i) fatt **::=** lett **|** (exp)

ii) term **::=** fatt **|** term1 * fatt

iii) exp **::=** term **|** exp1 + term

iv) bexp **::=** (exp1 = exp2) **|** (bexp1 and bexp2) **|** (bexp1 or bexp2) **|** (not bexp1)

dove lett è un generico elemento di Lettere, fatt è un generico elemento di Fatt ....

Ultimo esempio: supponiamo di voler definire tutte le stringhe su {1, 2, 3} del tipo ww, cioè formate concatenando due stringhe uguali (es. 11, 123123, 3131,....). Chiamando Rip l'insieme di queste stringhe, lo possiamo definire facilmente

Def_8

Rip è l'insieme di stringhe su {1, 2, 3} definito [per pattern matching] da:

rip **::=** w w dove w è un generico elemento di {1, 2, 3} *

9.4 Definizioni induttive di funzioni

Se A è un insieme definito induttivamente, succede spesso che per definire una funzione f: A → B (per qualche B) si possa seguire passo-passo la definizione induttiva di A.

Vediamo subito un esempio.

Esempio 1

Consideriamo l'insieme Exp di espressioni definite, per pattern matching da:

exp **::=** lett **|** (exp1 # exp2) **|** (exp1 * exp2) dove lett è un generico elemento di Lettere

Abbiamo usato il simbolo # invece del solito + per evitare confusioni in quanto segue ed abbiamo messo tante parentesi per evitare ogni forma di ambiguità.

Vogliamo definire la funzione (totale)

conta_per : Exp → **N** tale che conta_per(exp) = il numero di simboli * presenti in exp.

E` facile definire conta_per seguendo le definizione induttiva di Exp. In genere si scrive:

conta_per : Exp → **N** è la funzione (totale) data da:

conta_per(lett) = 0 se lett ∈ Lettere

conta_per((exp1#exp2)) = conta_per(exp1) + conta_per(exp2)

conta_per((exp1*exp2)) = 1 + conta_per(exp1) + conta_per(exp2)

Notare che il + a destra dell'= indica la somma in **N**.

A prima vista sembra un nuovo tipo di definizione, ma non è cosí; per capirlo, basta guardare la funzione come un insieme, scrivere (x, y) ∈ f invece di f(x) = y e introdurre un po' di nomi:

conta_per è il sottinsieme di Exp × **N** definito induttivamente da:

base: lett ∈ Lettere ⇒ (lett, 0) ∈ conta_per

passo: (n1, exp1), (n2, exp2) ∈ conta_per ⇒ (n1+n2, (exp1#exp2)) ∈ conta_per

(n1, exp1), (n2, exp2) ∈ conta_per ⇒ (1+n1+n2, (exp1*exp2)) ∈ conta_per

Però, è scomodo vedere le cose in questo modo e non ci siamo abituati, quindi ritorniamo alla prima definizione di conta_per. I principi su cui si basa sono i seguenti:

1. preso exp in Exp, exp può avere solo 3 forme: lett oppure (exp1#exp2) oppure (exp1*exp2)
2. per ciascuna di queste possibilità scriviamo una clausola per conta_per; notare che per ogni espressione exp c'è solo una clausola che si applica ad exp; da questo segue l'univocità;
3. per quanto riguarda la totalità: è chiara nel caso di base; negli altri due casi, per sapere il valore di conta_per(exp) dobbiamo prima calcolare il valore di conta_per su exp1 ed exp2; non si cade in un circolo vizioso, perchè exp1 ed exp2 sono "più piccole" di esp; iterando il procedimento si arriva alle singole lettere;

4. per quanto riguarda la correttezza: è abbastanza chiaro che la funzione conta veramente il numero di simboli *

Questo ragionamento informale si può precisare sotto forma di dimostrazione per induzione aritmetica generalizzata sulla lunghezza delle espressioni, viste come stringhe.

Usiamo |exp| per indicare la lunghezza di exp.

Tesi: per ogni exp di Exp: a) esiste uno ed un solo n in N tale che conta_per(exp) = n

b) n è il numero di simboli * nella stringa exp

(Traccia di) Dimostrazione per induzione su |exp|

base: |exp| = 1; quindi exp deve essere una lettera e la tesi è ovvia

passo: |exp| >1. Allora ci sono solo due casi.

1° caso: exp = (exp1#exp2) ed exp1, exp2 sono univocamente determinate. Ovviamente, |exp1| < |exp| ed |exp2| < |exp|. Aplicando ad exp1 l'ipotesi induttiva, otteniamo che esiste ed è unico n1 tale che ..... ed analogamente per exp2; allora di nuovo la tesi è ovvia

2° caso: exp = (exp1*exp2). Analogo al precedente.

Quindi la definizione data è una *buona definizione* nel senso che definisce effettivamente una funzione che conta nel modo corretto il numero di simboli *.

Vediamo ora un caso più complicato, facendo una piccola modifica all'esempio precedente.

Esempio 2

Consideriamo l'insieme Exp di espressioni definite, per pattern matching da:

exp **::=** lett **|** exp1 # exp2 **|** exp1 * exp2 dove lett è un generico elemento di Lettere

Rispetto all'esempio 1, abbiamo tolto le parentesi; quindi abbiamo stringhe ambigue.

Vogliamo sempre definire la funzione (totale)

conta_per : Exp → **N** tale che conta_per(exp) = il numero di simboli * presenti in exp.

Seguendo le definizione induttiva di Exp:

conta_per : Exp → **N** è la funzione (totale) data da:

conta_per(lett) = 0 se lett ∈ Lettere

conta_per(exp1#exp2) = conta_per(exp1) + conta_per(exp2)

conta_per(exp1*exp2) = 1 + conta_per(exp1) + conta_per(exp2)

Ragionando come sopra, è facile vedere che;

per ogni exp, esiste (almeno) un n t.c. conta_per(exp) = n

Il problema è l'univocità. Infatti, se prendiamo exp = a * b # c ci sono due clausole della definizione di conta_per che si applicano ad exp. In effetti, in questo caso non succedono guai (però è un po' noioso dimostrarlo). In generale, però, il pericolo esiste e bisogna stare attenti.

Un esempio di pasticcio, si ha prendendo:

Lettere = {a, b, c}; Exp come nell'esempio 2 e la seguente definizione:

calc : Exp → **N** data da

calc(a)= 2

calc(b)= 3

calc(c)= 5

calc(exp1 # exp2) = calc(exp1) + calc(exp2)

calc(exp1 * exp2) = calc(exp1) × calc(exp2)

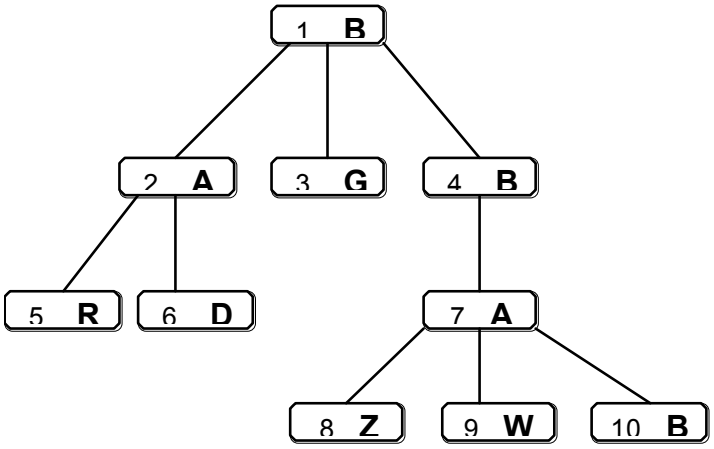
E` facile vedere che calc non è una funzione; basta considerare la solita a*b#c.

10

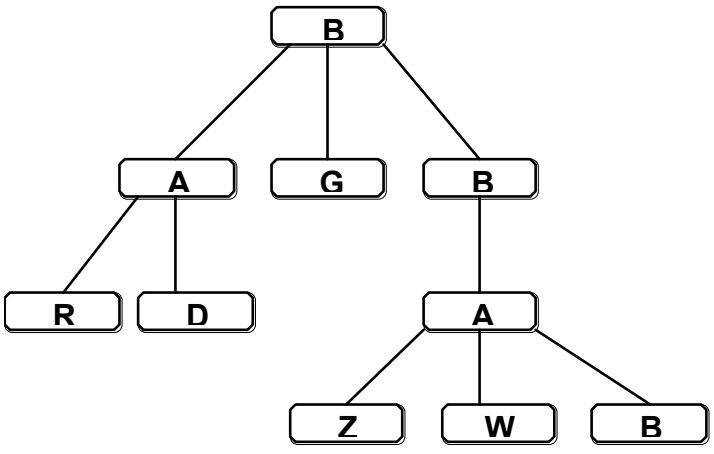
Alberi

10.1 Alberi come “disegni”

In prima battuta, gli alberi sono dei “disegni’ come quello che segue



In questo disegno distinguiamo: i **nodi** (cioè i rettangoli numerati), gli **archi** (cioè i segmenti che uniscono i nodi), le **etichette** (le lettere dentro i nodi). In un albero i nodi sono tutti distinti (mentre due nodi diversi possono avere la stessa etichetta), ecco perchè li abbiamo numerati. Spesso però, per semplicità, la distinzione tra i nodi viene lasciata implicita e l’abero viene disegnato come segue.



Le etichette non sono un ingrediente essenziale; dal punto di vista matematico si studiano anche, e soprattutto, gli alberi senza etichette; tuttavia nelle applicazioni sono presenti.

Guardando dall’alto verso il basso, si ha la nozione **padre** e **figlio** e quindi di **fratello**: se due nodi sono collegati da un arco, quello che sta sopra è il padre, quello che sta sotto è il figlio; ad esempio, i nodi 8 ,9, 10 sono tutti figli del nodo 7, e quindi sono fratelli. I nodi senza figli (qui: i nodi 5, 6, 3, 8, 9, 10) sono detti **foglie**; tutti gli altri nodi sono detti **nodi interni** dell’albero. In un albero c’è sempre un nodo, ed uno solo, senza padre, qui il nodo 1; questo nodo è detto **radice**. L’**apertura** (branching) **di un nodo** (si dice anche il grado di un nodo) è uguale al numero dei suoi figli.

Se chiudiamo per transistività la relazione padre-figlio, abbiamo la **relazione antenato-discendente**; ad esempio, il nodo 4 è un antenato del nodo 9 e quindi il nodo 9 è un discendente del nodo 4; tutti i nodi non radice sono discendenti della radice.

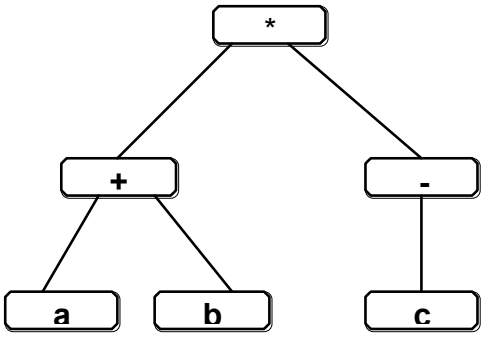
Una successione di nodi, $<n_0, ..., n_k>$ tali che n_j è padre di n_{j+1} è un **cammino** sull'albero, di lunghezza k, da n_0 ad n_k ; la **lunghezza** di un cammino è quindi uguale al numero di archi percorsi; l'**altezza** dell'albero è il massimo delle lunghezza dei cammini sull'albero; quindi è sempre la lunghezza di un cammino dalla radice ad una delle foglie. Nell'esempio (ed usando solo i numeri): <1, 2, 6> e <4, 7, 10> e <7> sono cammini (ma non sono i soli); l'altezza è 3.

Il **livello** (si dice anche profondità) di un nodo è definito come segue: la radice ha livello 0, i suoi figli livello 1, e così via; quindi il nodo 8 è a livello 3. In effetti: l'altezza di un albero è uguale al massimo tra i livelli delle sue foglie.

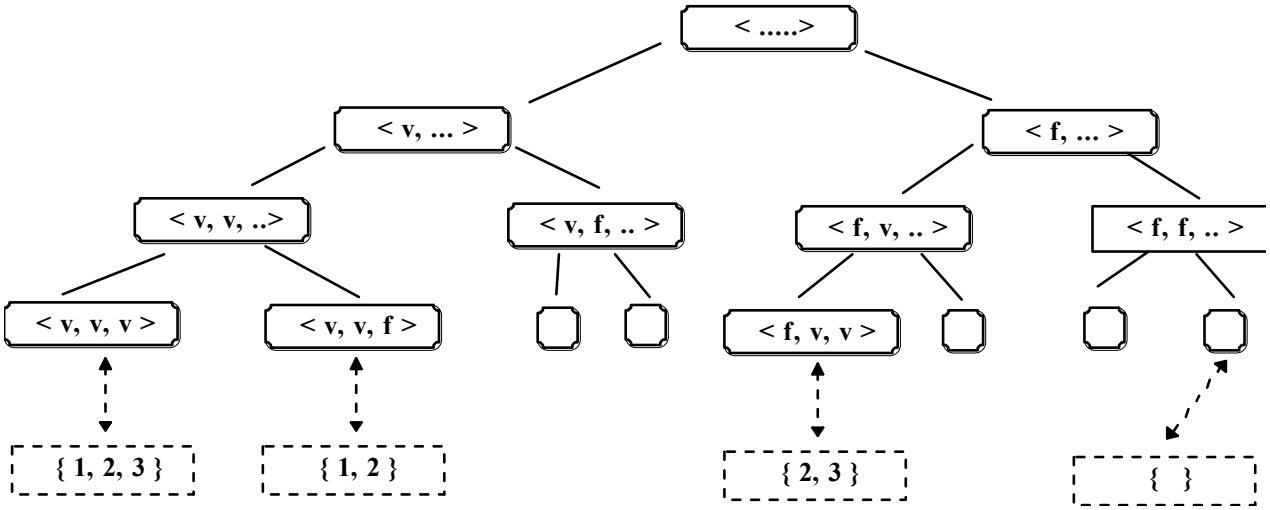
Infine: ogni nodo di un albero individua un **sottoalbero**, formato dal nodo stesso e da tutti i sui discendenti; nell'esempio i sottoalberi sono 10 (perchè i nodi sono 10): c'è tutto l'albero, l'albero formato dal nodo 2 e dai sui figli, quello formato dal solo nodo 3,.....; notare che i nodi 1, 2, 3, 4 non formano un sottoalbero.

Gli alberi sono utili per modellare molte situazioni; vediamo qualche esempio.

- La struttura di un file system (se ignoriamo eventuali "link", o alias) è rappresentabile con un albero, dove la foglie corrispondono ai files (e alle directory vuote) e gli altri nodi a directory.
- Le espressioni si possono rappresentare sotto forma di alberi, che mettono bene in luce la loro struttura; ad esempio, l'espressione (a + b) * (- c) si può rappresentare con l'albero che segue; notare che nella rappresentazione ad albero le parentesi non sono necessarie.



- La gerarchia di rettangoli vista parlando di visibilità delle dichiarazioni si può rappresentare con un albero; per un programma C contenuto in un solo file: la radice corrisponde al file, le funzioni (incluso il main) ai suoi figli; se una funzione contiene blocchi con dichiarazioni allora darà origine ad un sottoalbero.
- Ad un gioco come gli scacchi si può associare un albero; la radice rappresenta la situazione iniziale della scacchiera; per ogni possibile prima mossa del 'bianco" c'è un figlio, che corrisponde alla situazione della scacchiera dopo quella mossa; ciascuno di questi nodi ha tanti figli quante sono le possibili mosse del "nero" (a partire dalla configurazione corrispondente), e così via; le foglie corrispondono alle situazioni di "scacco matto" (o di "stallo", se è possibile). Una partita corrisponde ad un cammino dalla radice ad una foglia.
- Prendiamo il problema, già visto, di generare tutti i sottinsiemi di {1, ..., K} e ricordiamo che questi sottinsiemi sono in corrispondenza biunivoca con le successioni di "vero" e "falso" di lunghezza K (vero in posizione j indica che il numero j è nel sottisieme). Possiamo organizzare il procedimento usando un "albero di decisione": la radice corrisponde a non aver deciso nulla; ha due figli: quello di sinistra corrisponde alla scelta di "prendere l'elemento 1", quello di destra alla scelta di non prenderlo; ciascuno di essi ha due figli: quello di sinistra corrisponde alla scelta di "prendere l'elemento 2", quello di destra alla scelta di non prenderlo; e così via; infine, le foglie corrispondono ai sottinsiemi. Per chiarezza, facciamo il disegno, nel caso K=3; il disegno è incompleto per motivi di spazio; le caselle tratteggiate, non fanno parte dell'albero, ma mostrano la corrispondenza tra foglie e sottinsiemi; v = vero, f = falso.



Quelli visti sono solo alcuni esempi di uso degli alberi; comunque sono sufficienti a chiarire un punto importante: il "tipo di alberi" che ci interessa, cambia a seconda delle situazioni.

Le varianti principali di alberi si hanno al variare delle caratteristiche seguenti e alle loro combinazioni:

- apertura dei nodi: arbitraria, ma sempre finita (es file system); sempre = K, per K fissato (es degli scacchi); sempre = K, tranne che per le foglie (esempio dei sottinsiemi);
- ordine tra i figli: significativo (es dei sottinsiemi o delle espressioni, quando ci sono operazioni non commutative); non significativo (gioco degli scacchi);
- presenza e ruolo dell'albero vuoto.

Poichè ci sono tante varianti, ci sono anche tanti tipi di dato definiti attorno all'idea di albero. Ne vedremo alcuni (tries, alberi di ricerca, ma non in queste note), però prima concentriamoci sulla definizione degli oggetti, cioè degli alberi.

Il fatto è che i disegni aiutano l'intuizione, ma si prestano poco a ragionamenti rigorosi. Quindi dobbiamo definire gli alberi indipendentemente dai disegni; per fare questo, le definizioni induttive risultano particolarmente comode. Poichè ci sono tante varietà di alberi, ci sono tante definizioni induttive; ne vedremo solo alcune.

Un punto comune a tutte: **si suppone di avere un insieme, NODI, da cui prendere i nodi**; per non correre il rischio di restare senza nodi, supponiamo che NODI sia infinito, ma numerabile, cioè in corrispondenza biunivoca con **N**.

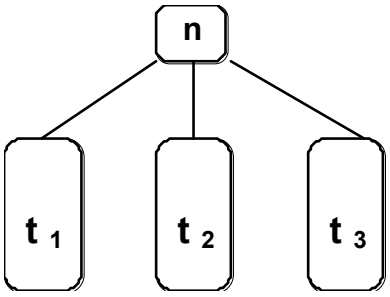
10.2 Alberi ad apertura illimitata con ordine tra i figli (e senza albero vuoto)

Primo tentativo di definizione.

- base: se n è un nodo, allora n è un albero;
- passo: se n è un nodo e $t_1, ..., t_k$ per (qualche) k>0, sono alberi,
  allora la coppia (n, $< t_1, ..., t_k >$) è un albero.

Notare che $< t_1, ..., t_k >$ è una successione, quindi abbiamo un ordine tra i figli.

Grafiicamente, se k =3, il passo corrisponde a:



Qui e nel seguito usiamo la seguente convenzione: i nodi sono quadrati, o tondi, o rettangoli messi "in orizzontale", mentre i sottoalberi sono "rettangoli in verticale".

C'`è un problema in questa definizione: potremmo avere tanti nodi uguali; ad esempio, in base alla definizione potremmo avere l'albero (n, < (n, <n>), n>). Invece è importante che i nodi di un albero siano tutti distinti (per non fare confusione). Inoltre la definizione non precisa chi è la radice, chi è padre e chi è figlio,..... Infine la notazione non è omogenea: l'albero fatto da un solo nodo coincide col nodo, gli altri sono coppie; meglio uniformare, usando sempre delle coppie.

Allora definiamo, induttivamente, non solo gli alberi, ma anche, per ciascun albero t: l'insieme dei suoi nodi, la sua radice e la relazione padre-figli.

Def_1

L'insieme degli alberi su NODI, che indichiamo con Tree(NODI), e per ciascun albero t: l'insieme dei suoi nodi, Nodi(t), la radice, root(t), la relazione padre-figli sono definiti induttivamente da:

base: se $n \in NODI$ allora $(n, < >) \in Tree(NODI)$
inoltre $n = root((n, < >))$ $Nodi((n, < >)) = \{n\}$

passo se $n \in NODI$; $t_j \in Tree(NODI)$, per $j = 1, ..., k$, per qualche $k>0$,
e $Nodi(t_i) \cap Nodi(t_j) = \varnothing$, per i?j, e $n \notin Nodi(t_1) \cup ... \cup Nodi(t_k)$
allora $(n, < t_1 , ..., t_k >) \in Tree(NODI)$ inoltre, se usiamo t per indicare $(n, < t_1 , ..., t_k >)$:
$root(t) = n$; $Nodi(t) = \{n\} \cup Nodi(t_1) \cup ... \cup Nodi(t_k)$;
i fligli di n sono, nell'ordine: $root(t_1),..., root(t_k)$

Notare che basta definire chi sono i figli della (nuova) radice; la relazione padre-figlio all'interno di $t_1, ..., t_k$ è, induttivamente, definita dalla stessa regola.

A questo punto si può precisare la nozione di **altezza** di un albero.

Def. La funzione $h: Tree(NODI) \rightarrow \mathbf{N}$ è data da:
$h((n, <>)) = 0$
$h((n, < t_1 , ..., t_k >)) = 1 + max\{ h(t_1), ..., h(t_k) \}$

Un po' più complicato risulta definire la nozione di **livello**. Il livello, infatti, dipende sia dal nodo che dall'albero; quindi la funzione livello è

$liv : NODI \times Tree(NODI) \rightarrow \mathbf{N} \cup \{-1\}$

Dato un nodo n ed un albero t, ci sono solo due casi: n non compare in t (allora il risultato dovrebbe essere indefinito; ma qui, per comodità, è -1) oppure compare una ed una sola volta in t; in questo caso, compare: o come radice, oppure all'interno di uno dei sottoalberi. Dunque la definizione induttiva di liv è:

$liv(n, (n, <>)) = 0$
$liv(n, (n', <>)) = -1$ se n ? n'
$liv (n, (n, < t_1 , ..., t_k >)) = 0$
$liv (n, (n', < t_1 , ..., t_k >)) = d$ se n ? n' dove:

se definiamo: $d_j = liv(n, t_j)$ per $j = 1, ..., k$

allora: $d = -1$ se $d_1 = ... = d_k = -1$

$d = 1 + \max\{d_1, ..., d_k\}$ altrimenti.

Infine, per definire **i cammini** si usa le relazione padre-figli data dalla definizione iduttiva:

un cammino di lunghezza k nell'albero t, con k=0, è una successione di k+1 nodi $<n_0, ..., n_k>$, tali che: $n_0, ..., n_k$ appartengono a Nodi(t) ed inoltre n_i è il padre di n_{i+1} in t, per i = 0, ..., k-1.

Nelle definizioni precedenti non c'è nessun riferimento alle **etichette**; in effetti mancano; se le vogliamo, dobbiamo aggiungerle. E` molto semplice:

sia Etichette l'insieme delle nostre etichette; un **albero etichettato** è una coppia (t, lab) dove lab (abbreviazione di label, etichetta) è una funzione da Nodi(t) in Etichette (che dunque associa un etichetta ad ogni nodo di t).

10.3 Alberi ad apertura illimitata senza ordine tra i figli (e senza albero vuoto)

Basta modificare la definizione precedente in un sol punto, precisamente: nel passo, invece di usare una successione $< t_1 , ..., t_k >$, usiamo l'insieme $\{ t_1 , ..., t_k \}$; cioè:

passo se .... *tutto come sopra*

allora $(n, \{ t_1 , ..., t_k \}) \in$ Tree(NODI) inoltre ..... *come sopra* .....
i fligli di n sono: $root(t_1), ..., root(t_k)$

Quindi, usando sempre gli stessi nomi: Tree(NODI), ...

Def_2

L'insieme degli alberi su NODI, che indichiamo con Tree(NODI), e per ciascun albero t: l'insieme dei suoi nodi, Nodi(t), la radice, root(t), la relazione padre-figli sono definiti induttivamente da:

base: se $n \in$ NODI allora $(n, \{ \}) \in$ Tree(NODI) qui: $\{ \}$ equivale a $\varnothing$
inoltre $n = root((n, \{ \}))$ $Nodi((n, \{ \})) = \{n\}$

passo se $n \in$ NODI ; $t_j \in$ Tree(NODI), per j = 1, ..., k, per qualche k>0,

e $Nodi(t_i) \cap Nodi(t_j) = \varnothing$, per i?j, e $n \notin Nodi(t_1) \cup ... \cup Nodi(t_k)$

allora $(n, \{ t_1 , ..., t_k \}) \in$ Tree(NODI) inoltre, se usiamo t per indicare $(n, \{ t_1 , ..., t_k \})$:

$root(t) = n$; $Nodi(t) = \{n\} \cup Nodi(t_1) \cup ... \cup Nodi(t_k)$;

i fligli di n sono: $root(t_1), ..., root(t_k)$ *e non c'è ordine tra di essi*.

Per altezza, livello di un nodo, cammini, etichette tutto come sopra, con gli ovvi cambiamenti.

10.4 Alberi ad apertura limitata (e senza albero vuoto)

A questo punto dovrebbe essere chiaro che basta modificare di poco le definizioni precedenti.

Sia nel caso di figli ordinati, che di figli "disordinati", fissato un K>0

- se vogliamo apertura = K, nel passo aggiungiamo la condizione : k = K
- se vogliamo apertura = K (per tutti i nodi non foglie), nel passo scriviamo:
  se $n \in$ NODI ; $t_j \in$ Tree(NODI), per j = 1, ..., **K** ..... *tutto il resto come sopra (cambiando k in K).*

Gli alberi con apertura = K (per tutti i nodi non foglie), con o senza ordine tra i figli, dove le foglie sono tutte allo stesso livello, vengono detti anche **alberi completi**, oppure **alberi perfettamente bilanciati** (vedere sotto per un disegno con K=3).

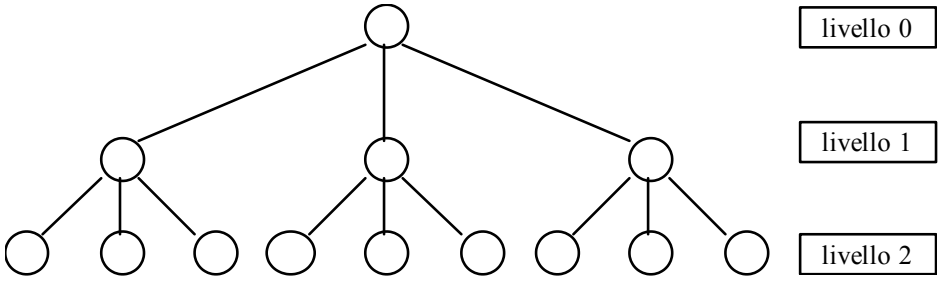
Per questi alberi perfettamente bilanciati c'è un **legame tra altezza e numero di nodi** (questo ci servirà spesso in seguito, parlando di complessità degli algoritmi).

Sia t uno di questi alberi, con altezza h; allora:

1. t ha esattamente K^h foglie; quindi $h = \log_K$(numero delle foglie)
2. t ha esattamente $(K^h - 1) / (K - 1)$ nodi interni

La 1. si intuisce disegnando un albero (vedi sotto) e si dimostra facilmente per induzione su h; si arriva alla 2. usando la tecnica già vista parlando di successioni (in effetti non è un caso: c'è una relazione tra alberi e successioni).

Per capire come si arriva alla 1 basta considerare un piccolo esempio, con K=3



A livello 0 c'è un solo nodo, a livello 1 ce ne sono 3, a livello 2 ce ne sono 3*3,..... Ricordiamo che l'altezza è uguale al livello delle foglie.

Notare che anche con K=2, basta arrivare ad altezza 10 per avere piú di 1000 foglie!

Sempre nel caso K=2, le formule di sopra ci dicono che le foglie sono una in più di tutti gli altri nodi, quindi da sole costituiscono la metà di tutto l'albero. Con K>2 il fenomeno è ancora piú macroscopico; basta guardare il disegno di sopra: 4 nodi interni e 9 foglie.

10.5 Alberi posizionali ed albero vuoto

Gli alberi posizionali costituiscono un raffinamento degli alberi con ordine tra i figli. Il caso più frequente è quello degli alberi binari.

In un **albero binario**, si distingue tra "figlio sinistro" e "figlio destro"; quindi per ogni nodo ci sono 4 possibilità: non ha figli, ha solo il figlio sinistro, ha solo il figlio destro, ha due figli.

Se prendiamo la Def_1, anche aggiungendo la condizione k = 2, riusciamo a catturare solo tre di queste possibilità; infatti nel casi di un unico figlio non si riesce a distinguere tra destra e sinistra.

La soluzione più elegante utilizza l'albero vuoto, che indicheremo con $\perp$ Quest'albero non ha nodi, e quindi non ha radice, ed ha altezza indefinita.

Supponiamo sempre di partire avendo un insieme NODI dei nodi.

Def_3

L'insieme degli alberi binari su NODI, che indichiamo con BinTree(NODI), e per ciascun albero t: l'insieme dei suoi nodi, Nodi(t), la radice, root(t), la relazione padre-figli sono definiti induttivamente da:

base: $\perp \in$ BinTree(NODI) inoltre Nodi(($\perp$) = $\varnothing$

passo se $n \in$ NODI ; $t_1 , t_2 \in$ BinTree(NODI)

e $Nodi(t_1) \cap Nodi(t_2) = \varnothing$ e $n \notin Nodi(t_1) \cup Nodi(t_2)$

allora ($n, < t_1 , t_2 >$) $\in$ BinTree(NODI) inoltre, se usiamo t per indicare ($n, < t_1 , t_2 >$) :

$root(t) = n$; $Nodi(t) = \{n\} \cup Nodi(t_1) \cup Nodi(t_2)$;

se $t_1 ? \perp$ allora $root(t_1)$ è il figlio sinistro di n; se $t_2 ? \perp$ allora $root(t_2)$ è il figlio destro di n.

A questo punto è chiaro il ruolo dell'albero vuoto nella costruzione: $\perp$ in posizione 1 (e/o 2), indica che manca il figlio di sinistra (e/o il figlio di destra).

Questa definizione si estende facilmente al caso in cui il numero di figli è = K, per un qualunque K fissato (allora si parla di alberi K-ri), ed al caso in cui il numero di figli è arbitrario (ma finito); l'unica differenza è che non si parla più di figlio destro e sinistro, ma di 1° figlio, 2° figlio,..... **per esercizio**.

Notare che gli alberi K-ri **pieni** (tutti i nodi non foglie hanno esattamente K figli) coincidono con gli alberi con ordine tra i figli ed apertura = K; gli alberi alberi K-ri **completi** (o **perfettamente bilanciati**), che sono pieni e con le foglie tutte allo stesso livello, coincidono con gli alberi con ordine tra i figli ed apertura = K completi.

10.6 Operazioni sugli alberi

Dipendono dal tipo di albero; qui solo alcune operazioni di uso generale; maggiori dettagli per le operazioni di alcuni tdd che vedremo (tries ed alberi di ricerca, non in queste note). Usiamo t, t1, ... per indicare alberi; n, n1,.... per indicare nodi.

Costruttori:

1. empty_tree — è la costante albero vuoto
2. mk_tree — mk_tree(n, t1, ...,tk) = l'albero con radice in n e sottoalberi t1, ..., tk ; il risultato varia a seconda del tipo di alberi considerati

Selettori:

3. root — root(t) è il nodo radice di t
4. padre — padre(t, n) è il padre del nodo n in t
5. figlio — figlio(t, n, j) è il figlio j-mo di n in t; ha senso solo nel caso di alberi ordinati o posizionali
6. sottoalbero — sottoalbero(t, n, j) è il sottoalbero di t che ha la radice nel j-mo figlio di n; anche questa solo nel caso di alberi posizionali

Proprietà:

7. is_empty_tree — is_empty_tree(t) = t è vuoto; ha senso solo se è previsto avere l'abero vuoto;
8. is_leaf — is_leaf(t,n) = n è una foglia dell'albero t (cioè n non ha figli);
9. n_figli — n_figli(t,n) = numero di figli (non vuoti) di n in t

Come nel caso delle successioni, alcune operazioni sono parziali; ad esempio, se tra gli alberi che consideriamo c'è l'abero vuoto, allora l'operazione root è parziale, perchè root($\perp$) non è definito.

11 Visite degli alberi

La visita di un albero è un'operazione di attraversamento della struttura dell'albero. Si tratta di "visitare" tutti i nodi di un albero nel senso di fare in modo che un algoritmo faccia una qualche operazione su ciascun nodo.

L'idea di visita è generale e indipendente dalle operazioni particolari che si fanno su ciascun nodo. Al limite si potrebbero visitare tutti i nodi senza farci niente, ad esempio semplicemente per contare quanti sono.

Ci sono comunque molte applicazioni sensate, ad esempio:

- scrivere in output il contenuto (etichetta, informazione) presente in ciascun nodo dell'albero;
- ricercare il nodo contenente una data informazione;
- confrontare se due alberi contengono le stesse informazioni;
- fare una copia dell'albero;

eccetera...

Ci sono tanti tipi di alberi, ma l'idea di visita si applica a tutti e lo schema generale, ed i primi raffinamenti di questo, si possono dare riferendosi ad un albero generico. Supponiamo inizialmente che sia non vuoto.

Idea:

Visitare l'albero = visitare i suoi nodi.

Partendo dalla radice raggiungiamo gli altri nodi muovendo lungo gli archi dell'albero; quindi visitiamo un nodo solo dopo aver visitato il padre (non sono ammessi i "salti").

Schema generale (su un abero t che non è vuoto):

A) "visita" la radice di t (vedi sotto)

B) scegli un nodo v tra quelli visitati che abbia figli non visitati

se non esistono nodi con queste caratteristiche la visita è terminata

C) scegli alcuni dei figli di v non visitati e visitali; poi riparti da B)

Qui non precisiamo in cosa consista la "visita" di un nodo; per esempio, può consistere nello stampare l'etichetta, nel modificarla se soddisfa ad una condizione,.....

I vari tipi di visita si differenziano per l'ordine di attraversamento dei nodi, che dipende dai criteri di scelta usati in B) e C). Le visite piu' usate sono:

- DFS (depth first search, o visita in profondità),
- BFS (breadth first search, o visita in ampiezza, o per livelli).

11.1 Schemi di visita

11.1.1 Visita BFS (in ampiezza)

La sigla BFS sta per *breadth-first search.* Significa adottare una strategia che prima di scendere sui figli di un dato nodo ne visita tutti i fratelli, in modo che i livelli alti dell'albero vengono visitati prima di scendere ai livelli inferiori.

Idea: visito prima la radice (nodo a livello 0),
poi tutti i figli della radice (nodi a livello 1),
poi tutti i figli dei figli della radice (livello 2),
e cosi' via...

Vediamo come si realizza facendo le scelte opportune per B) e C).

Criteri di scelta:

B) scelgo il primo (in ordine di tempo) nodo visitato con figli non visitati.

C) scelgo tutti i suoi figli per visitarli.

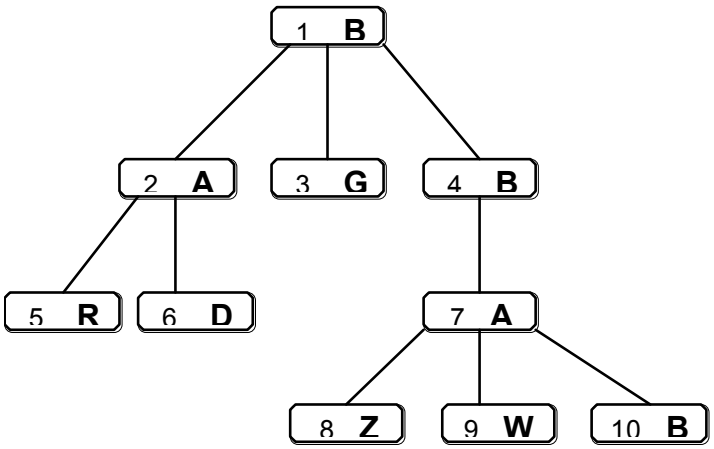
Esempio:

Con una BFS dell'albero in Fig. 1, si visitano: per primo il nodo 1; poi i nodi 2, 3, 4 (in un qualche ordine - vedere sotto); poi i nodi 5, 6, 7 (in un qualche ordine); poi i nodi 8, 9, 10 (in un qualche ordine).

Quindi: in una BFS, non posso visitare un nodo (ad esempio: 5) se prima non ho visitato tutti i nodi dei livelli superiori (cioè: 1, 2, 3, 4); ma a parità di livello (es: 2, 3, 4) l'ordine di visita non è imposto.

Quando poi si precisa un algoritmo per la BFS, bisogna appunto precisare quest'ordine.

Fig. 1



11.1.2 Visita DFS (in profondità)

La sigla DFS sta per *depth-first search.* Significa adottare una strategia che prima di visitare i fratelli di un nodo scende in profondità sui suoi discendenti, in modo da esaurire tutto un sottoalbero prima di passare al sottoalbero vicino.

Idea:

- scendo in profondità fino a esaurire un ramo
- quando arrivo in fondo al ramo (ho visitato la foglia) e quindi non posso piu' proseguire, torno indietro all'ultimo nodo visitato da cui si dipartano rami non ancora esplorati,
- riprendo la discesa da lí sui rami non ancora visitati.

Criteri di scelta per B) e C):

B) scelgo l'ultimo (in ordine di tempo) nodo visitato che abbia qualche figlio non visitato.

C) scelgo solo un figlio (qui c'è libertà) tra quelli non visitati.

Esempio:

Usiamo sempre la Fig. 1. Una possibile visita DFS, visita i nodi nell'ordine: 1, 2, 5, 6, 3, 4, 7, 8, 9, 10.

Un'altra posiibile DFS è: 1, 4, 7, 10, 9, 8, 3, 2, 6, 5. Supponendo che l'albero in figura sia un albero dove i figli di un nodo sono ordinati (e che l'ordine, da sinistra vesro destra, sia proprio quello dato dalla figura), allora la prima visita DFS corrisponde al criterio addizionale: dovendo scegliere tra due figli non visitati (o tra due rami non ancora esplorati) scelgo quello più a sinistra; l'altra DFS corrisponde a scegliere il più a destra.

11.2 Algoritmi di visita

Per concretezza, facciamo subito alcune scelte sul tipo di alberi e sul tipo di visita (si potrebbe invece continuare a sviluppare e precisare lo schema generale e poi ricavare da questo i vari casi particolari).

Consideriamo, come alberi:

- alberi ad apertura limitata = k, con ordine tra i figli (senza albero vuoto);
- alberi (posizionali) k-ri (e quindi con ordine tra i figli e albero vuoto).

  e, come ulteriore semplificazione, vediamo il caso k =3.

  Come visite consideriamo:

- la BFS, visitando i figli di un nodo nell'ordine in cui sono
- la DFS, scegliendo sempre il nodo non visitato più a sinistra

Quindi abbiamo 4 casi da discutere (due visite per due tipi di alberi)

Nel precisare gli algoritmi di visita sfrutteremo le definizioni induttive degli alberi; per comodità le ripetiamo, semplificandole (cioè omettendo i discorsi su: unicità dei nodi, radice.....).

NODI è l'insieme dei nodi.

Def_1 (alberi ad apertura = 3, con ordine tra i figli e senza albero vuoto)

L'insieme degli alberi Tree_3(NODI), è definito induttivamente da:

base: se $n \in$ NODI allora $(n, < >) \in$ Tree_3(NODI)

passo se $n \in$ NODI ; t_1, t_2, $t_3 \in$ Tree_2(NODI)

allora $(n, < t_1 >)$, $(n, < t_1 , t_2 >)$, $(n, < t_1 , t_2 , t_3 >) \in$ Tree_3(NODI)

Def_2 (alberi ternari)

L'insieme TerTree(NODI) è definito induttivamente da:

base: $\perp \in$ TerTree(NODI)

passo se $n \in$ NODI ; t_1 , t_2 , $t_3 \in$ TerTree(NODI)

allora $(n, < t_1 , t_2, t_3 >) \in$ TerTree(NODI)

11.2.1 Visita BFS di alberi ad apertura = 3, con ordine tra i figli

L'unica cosa non banale dell'algoritmo consiste nel ricordarsi via via che si avenza quali sono i nodi già visitati che hanno figli non visitati. Dato che quando visitiamo un nodo ci arriviamo "dall'alto" siamo sicuri che i suoi figli non sono ancora stati visitati.

L'idea chiave è quella di utilizzare un coda per mantenere i nodi già visitati che hanno figli da visitare. Metteremo un nodo in coda se questo ha figli nel momento in cui lo visitiamo. Quindi:

- La coda viene inizializzata mettendoci la radice quando la visitiamo
- A ogni iterazione togliamo un nodo dalla coda e visitiamo i suoi figli
- L'algoritmo si ferma quando la coda è vuota.

Tornando alla Fig. 1, vediamo un esempio:

- visitiamo la radice, il nodo 1; dato che ha figli, lo mettiamo in coda
- preleviamo il primo elemento dalla coda, sempre 1
  visitiamo uno alla volta i figli (nell'ordine: 2, 3, 4); dopo averli visitati li mettiamo in coda se hanno figli
  quindi ora la coda è: < 2, 4 > (e 2 è il primo elemento della coda)
- preleviamo il primo elemento dalla coda: 2
  visitiamo i figli (nell'ordine: 5, 6) e non li mettiamo in coda, perchè non hanno figli
  quindi ora la coda è: < 4 >
- preleviamo il primo elemento dalla coda: 4
  visitiamoil il figlio (7) e lo mettiamo in coda
  quindi ora la coda è: < 7 >
- eccetera,... fino a che la coda non è vuota.

  Nell'esempio, la coda entra veramente in gioco solo per ottenere di visitare prima i figli del nodo 2 e poi quelli del nodo 4; nel caso di alberi "più densi" (con tanti figli) il ruolo della coda diventa più evidente.

A questo punto possiamo scrivere, ad alto livello, una prima versione dell'algoritmo. Utilizziamo le primitive sugli alberi viste sopra più il tdd **coda_di_nodi** con le primitive seguenti:

- empty() genera la coda vuota
- is_empty(q) dice se q é una coda vuota
- enqueue(n,q) aggiunge il nodo n nella coda q (è come la primitiva in_coda già vista)

- dequeue(q) toglie il primo nodo dalla coda q e lo restituisce come valore (in pratica è una combinazione delle primitive testa e resto già viste).

Inoltre supponiamo di avere una procedura **visita(n)** che dato un nodo n esegue le operazioni relative alla sua visita (queste operazioni come detto dipendono dall'applicazione ma non sono importanti per l'algoritmo di visita).

```
procedura bfs_1 ( tt : albero  --   parametro IN)
   var  aux  :  coda_di_nodi
        x , y : nodi
        i : integer
   { aux ← empty()
     x  ← root(tt)
     visita(x)
     if n_figli(tt,x) > 0 then enqueue(x,aux)
     while  not is_empty(aux)  do
        {                  x ← dequeue(aux)
                           per i = 1,…..,n_figli(tt,x) :
                              {    y ← figlio(tt,x,I)
                                   visita(y)
                                   if n_figli(tt,y) > 0 then enqueue( y , aux )
                              }
        }
   }
```

Osservando l'algoritmo, si nota che risulta più semplice mettere in coda i nodi prima di visitarli:

```
procedura bfs_2 ( tt : albero  --   parametro IN)
   var  aux  :  coda_di_nodi
        x : nodi
        i : integer
   {    aux ← empty()
        enqueue( root(tt) , aux )
        while  not is_empty(aux)  do
           {               x ← dequeue(aux)
                           visita(x)
                           per i = 1,…..,n_figli(tt,x) :  enqueue( figlio(tt,x,i) , aux )
           }
   }
```

Osservando l'algoritmo bfs_2, si nota che è del tutto equivalente considerare una coda di alberi, invece di una coda di nodi (e vedremo che questo semplifica l'implementazione, quindi faremo così anche dopo). Basta usare una **coda_di_alberi** invece che una **coda_di_nodi** (con le stesse primitive di prima) e la primitiva sugli alberi **sottoalbero(t,n,i)** che è analoga a **figlio(t,n,i)** ma invece di restituire il figlio i-esimo restituisce tutto il sottoalbero che ha come radice il figlio i-esimo.

```
procedura bfs_2_bis ( tt : albero  --    parametro IN)
   var   aux  : coda_di_alberi
         tx : albero
         i : integer
   { aux ← empty()
     enqueue(tt)
     while  not empty(aux)  do
         {              tx ← dequeue(aux)
                        visita ( root(tx) )
                        per i = 1,…..,n_figli( tt , root(tx) ) :  enqueue( sottoalbero(tt,root(tx),i) , aux )

         }
   }
```

11.2.2 Visita BFS di alberi ternari

Si tratta semplicemente di adattare lo schema precedente al diverso tipo di alberi. Un modo molto semplice consiste nell'usare l'algoritmo bfs_2_bis (ma potremmo anche usare bfs_2), aggiungendo in testa un controllo, per il caso che tt sia vuoto, e adattando il resto. Ricordiamo che in questi alberi, se un nodo ha in posizione j un sottoalbero che è vuoto, allora il figlio j-mo è assente.

Usiamo una **coda_di_alberi** come prima. Dato che i figli sono solo tre usiamo le primitive **sottoalbero_sx, sottoalbero_ctr, sottoalbero_dx** per ottenere rispettivamente i sottoalberi sinistro, centrale e destro di un dato nodo.

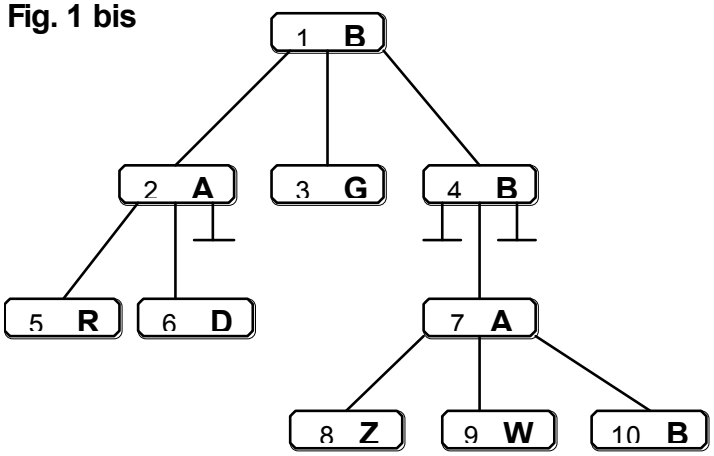
```
procedura bfs_3 ( tt : albero_ternario      --    parametro IN)
   var   aux  : coda_di_alberi
         tx, ty : albero_ternario
   { if   not is_empty_tree(tt)   then
         {    aux ← empty()
              enqueue( tt , aux )
              while  not empty(aux)  do
              {              tx ← dequeue(aux)
                             visita ( root(tx) )
                             ty ← sottoalbero_sx( tt , root(tx) )
                             if not is_empty_tree(ty) then enqueue( ty, aux )
                             ty ← sottoalbero_ctr( tt , root(tx) )
                             if not is_empty_tree(ty) then enqueue( ty , aux )
                             ty ← sottoalbero_dx( tt , root(tx) )
                             if not is_empty_tree(ty) then enqueue( ty , aux )
              }
         }
   }
```

11.2.3 Visita DFS di alberi ternari

Dopo quello che abbiamo visto, per la DFS il caso più semplice è quello degli alberi ternari.

Basta prendere l'utimo algoritmo visto, bfs_3, usare uno stack al posto della coda e rovesciare l'ordine in cui si inseriscono i fligli (prima quello di destra, poi quello di centro, poi quello di sinistra). Vediamo perchè su un esempio. Prendiamo la figura 1, con una lieve modifica: aggiungiamo sottoalberi vuoti al nodo 2 e al nodo 4, per precisare quali figli mancano; per i nodi foglia non c'è bisogno: mancano tutti.

Fig. 1 bis



Avendo scelto di andare prima a sinistra: i nodi vengono visitati nell'ordine seguente:

1, 2, 5, 6, 3, 4, 7, 8, 9, 10.

Usando uno stack e chiamando tj l'albero con radice = nodo j si procederebbe così:

aux:	operazioni successive
vuoto	metto t1 in testa
< t1 >	tolgo t1, visito la radice, e metto prima t4, poi t3, poi t2, così t2 si trova in testa
< t2, t3, t4 >	tolgo t2, visito la radice, e metto prima t6, poi t5
< t5, t6, t3, t4 >	tolgo t5, visito la radice, e non metto nulla perchè non ha figli
< t6, t3, t4 >	tolgo t6, visito la radice, e non metto nulla perchè non ha figli
< t3, t4 >	tolgo t3, visito la radice, e non metto nulla perchè non ha figli
< t4 >	eccetera

Usiamo uno **stack_di_alberi** con le solite primitive: **push** aggiunge un nodo in testa allo stack, **top** restituisce il nodo in testa allo stack (senza toglierlo dallo stack), **pop** toglie dallo stack il nodo in testa.

Quindi abbiamo l'algoritmo (ricavato da bfs_3):

```
procedura dfs_1 ( tt : albero_ternario      --   parametro IN)
 var      aux  : stack_di_alberi
          tx, ty : albero_ternario
   { if   not is_empty_tree(tt)   then
          {    aux ← empty()
               push( tt , aux )
               while  not empty(aux)  do
               {              tx ← top(aux)
                              pop(aux)
                              visita ( root(tx) )
                              ty ← sottoalbero_dx( tt , root(tx) )
                              if not is_empty_tree(ty) then push( ty , aux )
                              ty ← sottoalbero_ctr( tt , root(tx) )
                              if not is_empty_tree(ty) then push( ty , aux )
                              ty ← sottoalbero_sx( tt , root(tx) )
                              if not is_empty_tree(ty) then push( ty , aux )
               }
          }
   }
```

11.2.4 Implementazione ricorsiva della DFS per alberi ternari

C'è un altro modo di implementare la visita DFS: usando la ricorsione. Vediamolo prima nel caso degli alberi ternari.

Guardando la figura 1bis (ed usando sempre la convenzione di chiamare tj l'albero con radice nel nodo j), c'è un altro modo di descrivere come procede la visita DFS:

la visita di tutto l'albero, cioè di t1, consiste in :

- visitare la radice
- visitare il sottoalbero t2
- visitare il sottoalbero t3
- visitare il sottoalbero t4

ed inoltre: la visita dei sottoalberi, procede in modo analogo (di qui la ricorsione).

Generalizzando, la visita DFS di un albero t (anche vuoto) consiste in

se t non è vuoto, allora

- visita della radice
- visita DFS del sottoalbero di sinistra (anche se è vuoto)
- visita DFS del sottoalbero di centro (anche se è vuoto)
- visita DFS del sottoalbero di destra (anche se è vuoto)

(se t è vuoto non si fa niente)

Quindi abbiamo il seguente algoritmo ricorsivo:

```
procedura dfs_2 ( tt : albero_ternario      --   parametro IN)
   { if   not is_empty_tree( tt )  then
         {    visita ( root(tt) )
                  dfs_2( sottoalbero_sx( tt , root(tt) ) )
                  dfs_2( sottoalbero_ctr( tt , root(tt) ) )
                  dfs_2( sottoalbero_dx( tt , root(tt) ) )
              }
   }
```

Nota. Non è possibile fare l'analogo nel caso della visita in ampiezza; cioè non è possibile evitare l'uso esplicito della coda utilizzando una procedura ricorsiva.

11.2.5 Visita DFS di alberi ad apertura arbitraria, con ordine tra i figli

È facile generalizzare quanto fatto per gli alberi ternari, basta ricordare che non si ammette la presenza dell'albero vuoto e usare un contatore per ciclare sui figli (sottoalberi) di un dato nodo.

Algoritmo ricorsivo.

```
procedura dfs_3 ( tt : albero  --   parametro IN)
     var i : integer
     {    visita ( root(tt) )
          if  not is_leaf( tt , root(tt) ) then
               per i=1,……, n_figli( root(tt) ) : dfs( sottoalbero( tt, root(tt), i ) )
          non si fa niente se non ci sono sottoalberi
     }
```

Algoritmo non ricorsivo, usando uno stack.

```
procedura dfs_4 ( tt : albero  --   parametro IN)
   var   aux : stack_di_alberi
         tx : albero
         i : integer
   { aux ← empty()
     push(tt, aux)
     while   not is_empty(aux)  do
         {              tx ← top(aux)
                        pop(aux)
                        visita( root(tx) )
                             i ← n_figli( root(tx) )
                        while i > 0 do
                                 {    push ( sottoalbero(tt,root(tx),i) , aux )
                                      i --
                                 }
         }
   }
```

11.3 Visite pre-order, in-order, post-order per alberi binari

Nel caso degli alberi binari (posizionali con albero vuoto) spesso ha senso utilizzare versioni della DFS in cui l'ordine rispettivo di visita della radice e dei suoi sottoalberi cambia. Ci sono essenzialmente tre versioni:

- **pre-order:** quella vista finora: si visita prima la radice, poi il sottoalbero sinistro, infine quello destro;
- **in-order:** si visita prima il sottoalbero sinistro, poi la radice, infine il sottoalbero destro;
- **post-order:** si visitano prima i due sottoalberi (sinistro poi destro) e infine la radice.

Il fatto di propendere per una versione piuttosto che un'altra dipende dall'applicazione che stiamo considerando. Vediamo un esempio significatovo in cui è possibile utilizzare tutte e tre le versioni della visita secondo le esigenze.

Notazioni alternative per le espressioni algebriche

Consideriamo alberi che rappresentano espressioni algebriche con simboli di operazioni che hanno al più 2 argomenti (anche se queste espressioni si rappresentano correttamente con alberi non posizionali, purchè ci sia un ordine tra i figli, usiamo gli alberi posizionali perchè sono più comodi). Le nostre espressioni possono essere quindi composte da simboli di tre classi:

- **costanti:** ad esempio lettere o numeri che rappresentano valori sui quali possono agire gli operatori
- **operatori unari:** cioè che agiscono su un solo valore, ad esempio logaritmo, seno, coseno.
- **operatori binari:** cioè che agiscono su due valori, presi nell'ordine, ad esempio le quattro operazioni.

La regola base per rappresentare espressioni con alberi binari è la seguente:

- ogni nodo dell'albero contiene un simbolo di costante o di operatore;
- i nodi contenenti costanti sono foglie;
- un nodo contenente un operatore unario ha solo il figlio destro e il sottoalbero corrispondente contiene l'espressione a cui deve essere applicato l'operatore;
- un nodo contenente un operatore binario ha due figli corrispondenti nell'ordine alle espressioni a cui deve essere applicato l'operatore.

Un esempio è dato dalla figura che segue (dove indichiamo l'albero vuoto solo dove serve).

L'albero rappresenta l'espressione (a + b) * log c in cui abbiamo:

- un operatore * che viene usato al livello più esterno per moltiplicare il risultato dell'espressione tra le parentesi alla sua sinistra per il risultato dell'espressione alla sua destra;
- nella parentesi sinistra abbiamo un operatore + che viene applicato alle due costanti a e b;
- a destra abbiamo un operatore log che viene applicato alla costante c.

In pratica, un albero si ottiene a partire dalla notazione usuale (cosiddetta infissa) tramite la seguente regola induttiva (definita in modo intuitivo):

- (base): se l'espressione contiene solo una costante si ha un albero costituito da un solo nodo;
- (passo): altrimenti si cerca l'operatore "che agisce per ultimo" e si mette come radice; i sottoalberi sinistro (se esiste) e destro si costruiscono ricorsivamente, rispettivamente a partire dall'espressione che precede (se esiste) e da quella che segue tale operatore.

Trovare "l'operatore che agisce per ultimo" non è sempre facile, come abbiamo già visto, a meno di conoscere le regole di precedenza degli operatori moltiplicativi su quelli additivi o di adottare una noiosissima notazione piena di parentesi. Comunque, una volta che siamo riusciti a costruire l'albero, questo ci fornisce una interpretazione univoca per l'espressione (infatti le parentesi spariscono, ma l'interpretazione corrisponde).

In realtà, le espressioni si possono scrivere uilizzando tre notazioni alternative, due delle quali non necessitano di parentesi perché sono intrisecamente non ambigue:

- **Notazione infissa:** è quella a cui siamo abituati: ogni operatore unario precede l'espressione al cui valore deve essere applicato, mentre un operatore binario sta in mezzo tra le due espressioni a cui devono essere applicato. Se ignoriamo le regole di precedenza degli operatori moltiplicativi su quelli additivi, per avere una notazione non ambigua abbiamo visto che è necessario usare parentesi. Ad esempio, se non avessimo le parentesi sarebbe impossibile discriminare tra le due espressioni (a + b) * log c e a + (b * log c).
- **Notazione polacca prefissa:** ogni operatore sia unario che binario precede i suoi operandi che sono scritti in fila dopo di esso. In questo caso l'espressione dell'albero in figura si scrive:

  * + a b log c

  Per noi è molto più difficile da leggere perché possiamo trovare diversi operatori in fila e ad occhio non è facile capire i vari termini e fattori che compongono l'espressione. Per un calcolatore (ad esempio, un programma interprete), è molto più facile dell'infissa perché è una notazione univoca dove non è necessario l'uso di parentesi.
- **Notazione polacca postfissa:** analoga alla precedente, ma in questo caso gli operandi precedono l'operatore. L'espressione dell'esempio si scrive:

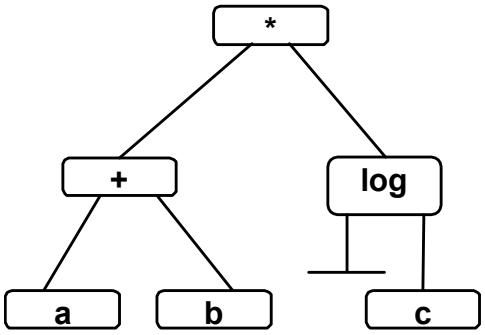
  a b + c log *

Partendo dall'albero che rappresenta l'operazione è possibile generare tutte e tre le notazioni per una stessa espressione, usando le diverse varianti della visita DFS. Assumiamo in questo caso che la procedura visita(nodo) abbia come effetto di stampare il simbolo che etichetta il nodo visitato:

visita pre-order ——————— notazione polacca prefissa

visita post-order ——————— notazione polacca postfissa

visita in-order ——————— notazione polacca infissa



Attenzione: la stringa che si ottiene dall'attraversamento in-order dell'albero non è la stessa da cui eravamo partiti: mancano le parentesi. Quindi la semantica dell'aspressione, che è univocamente determinata nell'albero, può andare persa nel generare la notazione infissa. Infatti nel nostro caso avremmo a + b * log c che non è la stessa cosa di (a + b) * log c come espressione algebrica. È piuttosto facile rimediare generando "molte" parentesi che conservino la struttura presente nell'albero. Non è molto facile invece generare la notazione con il minimo di parentesi necessarie alla quale siamo abituati.

Nota: nel caso più generale di alberi che rappresentano termini su una segnatura, la notazione infissa può non avere senso, ma restano le altre due. Con un po' di pazienza si possono aggiungere anche a queste le parentesi e le virgole (come in : f(a, g(b, c), d)).

L'implementazione delle visite è molto facile a partire dall'algoritmo ricorsive già visto per la DFS sinistra, che coincide con **pre-order**:

```
procedura preorder ( tt : albero_binario  --  parametro IN)
        {  if  not is_empty_tree ( tt )  then
                {   scrivi ( etichetta di root(tt) )
                    preorder( sottoalbero_sx ( tt, root( tt ) )
                    preorder( sottoalbero_dx ( tt, root( tt ) )
                }
        }
```

Per ottenere le visite **post-order** e **in-order** basta scambiare l'ordine delle tre istruzioni nel corpo dell'if:

```
procedura postorder ( tt : albero_binario      --  parametro IN)
        {  if  not is_empty_tree ( tt )  then
                    {   postorder(sottoalbero_sx ( tt, root( tt ) )
                        postoder(sottoalbero_dx ( tt, root( tt ) )
                        scrivi ( etichetta di root(tt) )
                    }
        }

procedura inorder ( tt : albero_binario    --  parametro IN)
        {  if  not is_empty_tree ( tt )  then
                    {   inorder(sottoalbero_sx ( tt, root( tt ) )
                        scrivi ( etichetta di root(tt) )
                        inorder(sottoalbero_dx ( tt, root( tt ) )
                    }
        }
```

Per aggiungere le parentesi nel corso della visita in-order si può fare così :

```
    procedura inorder ( tt : albero_binario      --  parametro IN)
        {  if  not is_empty_tree ( tt )  then
           then     {   scrivi ( " ( " )
                        inorder(sottoalbero_sx ( tt, root( tt ) )
                        scrivi ( etichetta di root(tt) )
                        inorder(sottoalbero_dx ( tt, root( tt ) )
                        scrivi ( " ) " )
                    }
        }
```

La notazione ottenuta è non ambigua ma piuttosto pesante, l'espressione dell'esempio verrebbe scritta così:

(((a) + (b)) * (log (c))).

Per riuscire a mettere meno parentesi, bisogna lavorare un po': **per esercizio**.

12 Implementazione degli alberi

Sono possibili diverse implementazioni, ciascuna delle quali può essere più o meno conveniente, in termini di spazio e/o di efficienza delle operazioni, secondo il tipo di alberi che si vogliono rappresentare e secondo l'applicazione. Ne vediamo solo alcune, cercando di confrontarle e discuterne vantaggi e svantaggi.

12.1 Alberi con apertura ≤ K -- Implementazione con record e puntatori ai figli

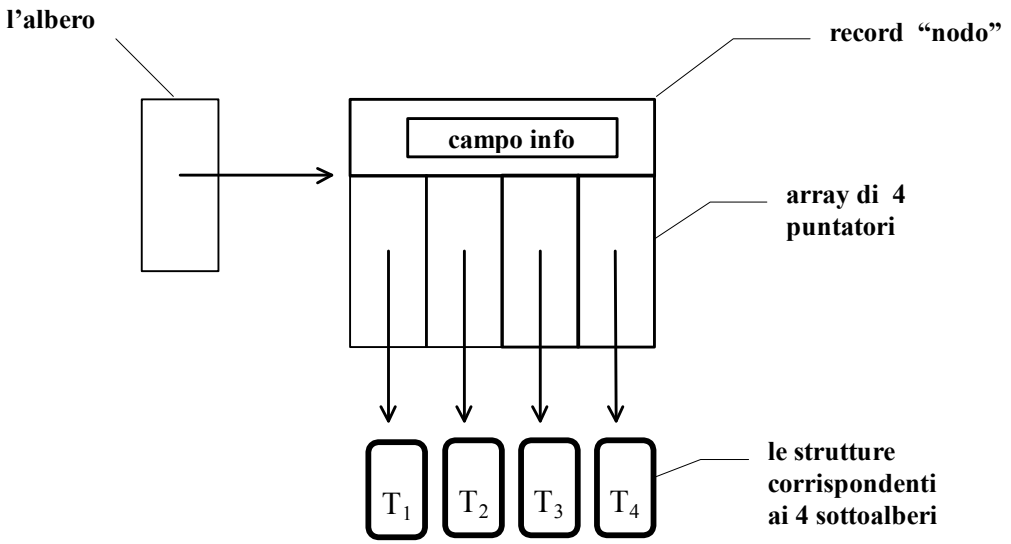
12.1.1 Implementazione degli oggetti

Questa implementazione non richiede di conoscere il numero massimo di nodi. È una estensione dell'idea di liste dinamiche realizzate con record e puntatori:

- Un nodo si rappresenta con un record a due campi: il campo info che corrisponde all'informazione contenuta nel nodo ed un campo children che è un array di K puntatori_a_nodo, che puntano ai figli, oppure sono nulli (se manca il figlio corrispondente).
- Un albero corrisponde ad un puntatore_a_nodo (che punta al nodo radice).

```
Type  nodo   =  record
                   Info:     Elem                  cioè il tipo che serve
                   Children:    array [1..k] of puntatore_a_nodo
                end
      tree    =  puntatore_a_nodo
```

Graficamente, con K=4:



Se K è piccolo e l'albero è posizionale, invece di un campo array di puntatori, si possono usare K campi puntatore con nomi che corrispondano alle posizioni (esempio: **left** e **right** negli **alberi binari)**,.

È naturale associare al puntatore nullo l'albero nullo, per cui in questa implementazione è automatico prevedere la presenza dell'albero nullo. Se non si vuole ammettere l'albero nullo, allora un puntatore nullo verrà interpretato come un figlio assente. L'unica differenza tra gli alberi posizionali (che ammettono l'albero nullo) e quelli non posizionali (che non lo ammettono) in questa implementazione è che nei posizionali possiamo avere misti nell'array puntatori nulli e non nulli (cioè potremmo avere il primo figlio, il secondo no, il

terzo sì e così via…..), mentre nei non posizionali metteremo tutti i figli presenti nelle prime posizioni dell'array e appena troviamo un valore nullo significa che da lì in poi non ci sono più figli.

Notare che anche se negli alberi non c'è un ordine tra i figli, l'implementazione impone un ordine (arbitrario).

Questa implementazione è una delle più usate, perchè è anche molto naturale.

Il difetto principale è che mentre è facile "scendere", cioè attraversare l'albero da padre a figlio, non è possibile risalire. Se l'applicazione richiede di poter risalire, bisogna aggiungere ulteriori campi puntatore nel record: ad esempio, un puntatore "all'indietro" che punta al padre (simile a come abbiamo fatto nelle liste doppie).

Un altro difetto è lo spreco di spazio; infatti tutti i nodi contengono K campi puntatore, compresi quelli che hanno meno di K figli ed in particolare le foglie che non ne hanno affatto. Nella situazione migliore, che è quella di un albero binario completo, i campi puntatore inutilizzati sono pari al doppio delle foglie, ossia due di più di quelli utilizzati, che sono pari al doppio dei nodi interni. Se K è abbastanza grande (in realtà basta che sia >2) i campi puntatore inutilizzati sono molti di più di quelli utilizzati. Se in più l'albero non è completo e magari neppure pieno la situazione peggiora ulteriormente.

12.1.2 Implementazione delle operazioni

Alcune si ottengono banalmente generalizzando quelle che abbiamo visto per le liste:

- Empty_tree() restituisce semplicemente il valore **null**.
- Is_empty_tree(t) controlla se il valore della variabile t è **null** oppure no.
- Root(t) restituisce il record puntato da t (che è un nodo) se t è non nullo, segnala errore altrimenti (vale tutta la discussione fatta per l'operazione *testa* sulle successioni).
- Is_leaf(n) controlla semplicemente se tutti i puntatori nel record sono nulli o no.
- N_figli(n) conta quenti **sono** i puntatori non nulli nel nodo (se l'albero non è posizionale conta solo finché non ne trova uno nullo).

  Vediamo le altre con un po' più di dettaglio.

  Implementazione di Figlio : Tree x Node x Integer → Node

  In questa implementazione con puntatori in realtà il parametro di tipo tree è superfluo (a meno che non si voglia controllare preventivamente se il nodo appartiene effettivamente all'albero, ma è un'operazione costosa). Infatti il nodo contiene già in sé le informazioni per accedere ai suoi figli.

  Quindi potremmo scrivere una funzione:

  function child(n : node, i: integer) : node

  che restituisce semplicemente il record puntato dall'i-esimo campo dell'array dei figli nel record n.

  Tuttavia questo figlio potrebbe non esistere:

- Nel caso di alberi non posizionali, conviene verificare che i abbia un valore inferiore a N_figli(n) e generare un errore in caso contrario (come al solito si pone il problema di come fare a generare l'errore: si potrebbe aggiungere un parametro Booleano OUT)
- Nel caso di alberi posizionali basta verificare se il puntatore i-esimo è nullo e restituire errore n questo caso.

  Per gli alberi posizionali però è ammissibile avere sottoalberi vuoti e il fatto di generare un errore quando si incontrano può essere fastidioso. Perciò può essere conveniente utilizzare l'operazione Sottoalbero invece della Figlio (vedi sotto).

  Implementazione di Sottoalbero : Tree x Node x Int → Tree

  Questa è del tutto analoga alla precedente ma invece di restituire il record del figlio corrispondente restituisce il suo indirizzo (infatti il tipo Tree è appunto un puntatore_a_nodo). Quindi anche un valore nullo è una risposta buona nel caso di alberi posizionali. La segnalazione resta nel caso di non posizionali.

Il figlio corrispondente si ottiene semplicemente applicando Root al risultato. Quindi di solito non conviene implementare sia Figlio che Sottoalbero, basta una delle due.

Osservazione: il tipo Node di questa implementazione non è molto pulito peché non contiene solo le informazioni sul singolo nodo, ma anche le informazioni su tutto il sottoalbero che ci sta sotto (tramite i puntatori). Un approccio più "pulito" consiste nel tenere il tipo Node *privato* per il tipo di dato, cioè non renderlo visibile ai moduli che usano questo tdd. In questo caso non si può mai usare come parametro di una primitiva. Come facciamo? Se consideriamo che il contenuto "ufficiale" (cioè visibile) di un nodo si debba limitare al solo campo info, allora potremmo:

- Implementare la root(t) con valore in Elem, cioè restituendo non tutto il record ma solo il contenuto del campo info
- Adottare una definizione alternativa Sottoalbero : Tree x Int → Tree con la seguente semantica:

  Sottoalbero(t,i) restituisce l'i-esimo sottoalbero della radice di t.
- Non implementare la Figlio (si usa invece Sottoalbero + Root)
- Sostituire la Is_leaf con una Is_single : Tree → Boolean che dice se un albero è formato di un solo nodo.
- Sostituire la N_figli con una N_sottoalberi : Tree → Integer che dice quanti sottoalberi ci sono sotto la radice di un albero.

Tutto funziona come prima facendo a meno del tipo nodo. La Root comunque ci consente di "guardare dentro" ai nodi.

Implementazione dei costruttori

Abbiamo già visto la Empty_tree che fornisce l'albero vuoto. Questo però va bene solo nel caso che l'albero vuoto sia ammesso. Se è inammissibile non ha senso implementarla e bisogna sostituirla con una operazione che produca un albero costituito di un solo nodo:

Init_Tree : Node → Treedefinita Init_Tree(n) = (n,<>)

Gli alberi con più nodi si costruiscono poi con la Mk_tree secondo una strategia *bottom-up,* cioè dalle foglie in su, raggruppando alberi già fatti come sottoalberi.

Problemi da risolvere:

- Come creare un nodo. Abbiamo visto che i nodi sono "mattoni" con i quali costruiamo l'albero e vanno allocati dinamicamente.
- Implementare la Init_tree
- Implementare la Mk_tree

Vediamo come si fa per gli alberi binari, per gli altri è analogo. In realtà quello che interessa l'applicazione che usa gli alberi è solo il *contenuto* di un nodo e non il record di tipo Node in sé stesso (abbiamo visto che questo tipo si può anche tenere privato, cioè "nascosto"). Nel nostro caso cioè interessa solo il campo **info**. Quindi potremmo allocare i nodi all'interno delle due operazioni implementandole come segue:

```
function Init_tree( e : Elem IN) : Tree
var t : Tree
     i : integer
{    t ← new(Node)
     t->Info ← e
     per i=1,….,K :   t->children[i] ← null
     return( t )
}
```

```
function Mk_tree( e : Elem,  T : array[1..K] of Tree ) : Tree
var t : Tree
    i : integer
{
    t ← new(Node)
    t->info ← e
    per i=1,….,K :   t->children[i] ← T[i]
    return( t )
}
```

In effetti per gli alberi che non ammettono l'albero vuoto la Mk_tree andrebbe cambiata un po' per permettere di aggiungere un numero di figli inferiore al massimo K. Si può aggiungere un parametro intero num che dice quanti sono i figli da mettere:

```
function Mk_tree( e : Elem,  T : array[1..K] of Tree,  num : integer ) : Tree
var t : Tree
    i : integer
{
    t ← new(Node)
    t->info ← e
    per i=1,….,num:     t->children[i] ← T[i]
    if  num < K  then   t->children[num+1] ← null
    return( t )
}
```

Nel caso di alberi che ammettono l'albero vuoto si vede invece che la Init_tree è superflua perché abbiamo :

Init_tree(e) equivale a Mk_tree(e, {Empty_tree(),….., Empty_tree()})

12.2 Alberi con apertura arbitraria -- "figlio sinistro -- fratello destro"

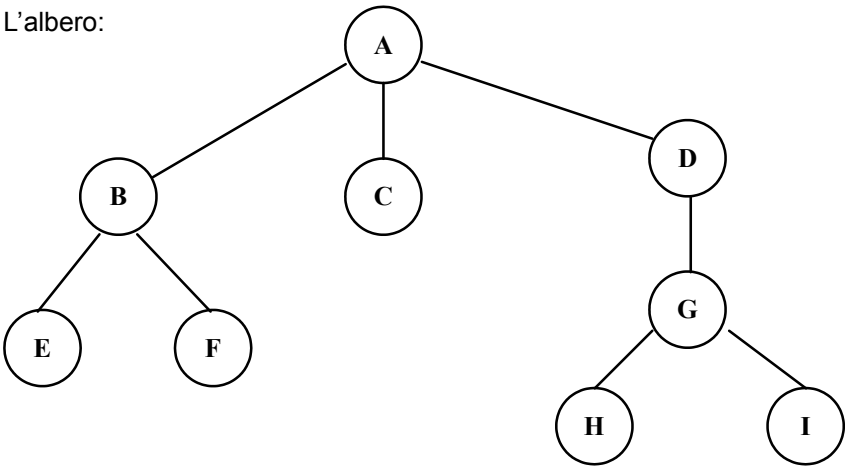
12.2.1 Implementazione degli oggetti

Permette di rappresentare alberi con un numero arbitrario di figli; si può usare anche per alberi ad apertura limitata a priori per risparmiare spazio.

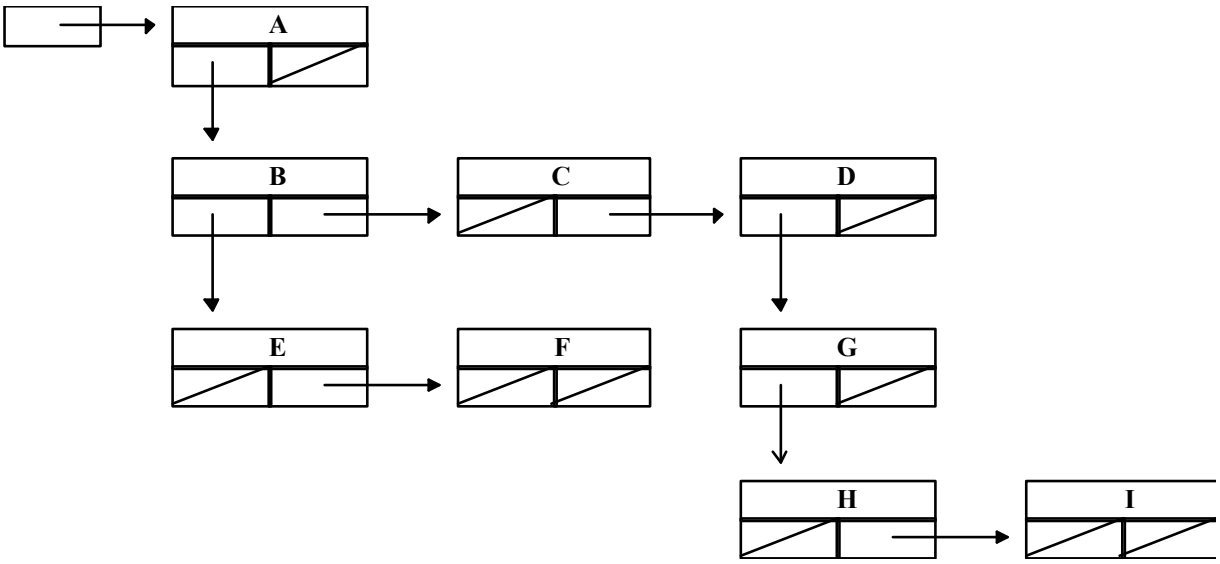
L'idea è che invece di mantenere i puntatori a tutti i figli all'interno del record che rappresenta un nodo, si tiene solo un puntatore al primo dei figli (con il puntatore "figlio sinistro" del nodo) e si concatenano i vari figli tra loro a formale una lista (con il puntatore "fratello destro" di ogni figlio).

Vediamo l'idea su un disegno.

L'albero:



viene rappresentato con la struttura seguente:



Dunque:

- il tipo albero è un puntatore_a_nodo
- il tipo nodo è un record a 3 campi: il campo info di tipo Elem, e 2 campi albero (cioè: puntatore_a_nodo); di questi il primo punta al primo dei figli (graficamente il più a sinistra), se c'è, mentre il secondo punta al fratello di destra (se c'è).

  Notare che qui, come prima, anche se negli alberi non c'è un ordine tra i figli, l'implementazione impone un ordine (arbitrario).

Vantaggi rispetto all'implementazione vista prima:

- Si possono implementare alberi con qualsiasi apertura senza decidere l'apertura massima a priori.
- I puntatori non utilizzati sono:
  - Uno per la radice, perché non ha fratelli
  - Uno per ogni nodo interno (radice compresa), perché l'ultimo dei suoi figli non ha fratelli
  - Uno per ogni foglia, perché non ha figli

  Quindi, se usata per alberi binari completi occupa lo stesso spazio della precedente (che non utilizza due puntatori per ogni foglia). In tutti gli altri casi è meno costosa.

Svantaggi:

- Non conviene per alberi posizionali perché si perde la possibilità di fare accesso diretto ai vari figli: per ottenere il figlio i-esimo bisogna scandire tutta la lista dei figli dal primo fino ad i, mentre con l'implementazione precedente si otteneva direttamente.

  Per esercizio: provare a implementare l'operazione Figlio definita come prima.
- È meno intuitiva della precedente: questo però è un problema solo di chi *implementa* il tdd, non di chi lo *usa.* Infatti chi lo usa vede solo i prototipi delle primitive, che sarano gli stessi di prima.

12.2.2 Implementazione delle operazioni

Per esercizio.

È simile a quella precedente. In più, per gestire le liste di fratelli, si utilizzando cose che si trovano nell'implementazione delle liste con puntatori.

12.3 Alberi con apertura K -- Implementazione con array

12.3.1 Implementazione degli oggetti

Come al solito quando si usano array bisogna fissare la dimensione massima degli oggetti che si adoperano (ignoriamo qui la possibilità di usare array dinamici). In questo caso si tratta del massimo numero di nodi che l'albero può avere, che viene fissato da una costante MAX.

I nodi dell'albero vengono mantenuti in un array dove ogni casella rappresenta un nodo e contiene l'informazione che ci interessa sul nodo, ossia è di tipo Elem.

Notare che una volta allocato l'array non si può modificare la sua dimensione. Quindi:

- Se vogliamo rappresentare un albero con più di MAX nodi non possiamo.
- Se vogliamo rappresentare un albero con molto meno di MAX nodi sprechiamo spazio.

Questa struttura è particolarmente comoda nel caso di alberi binari. Vediamo appunto questo caso.

L'array si riempie nel modo seguente (definizione induttiva):

- Il nodo radice si mette al primo posto;
- I figli di un nodo al posto j occupano i posti 2j e 2j+1.

Ne consegue che il padre del nodo al posto j (se j > 1) è il nodo (j div 2).

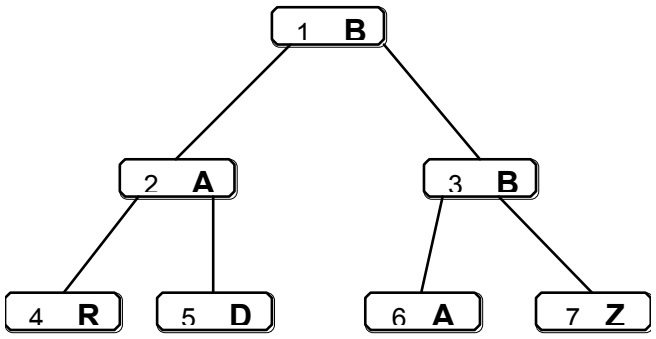
Ci serve un modo per sapere "dove finisce" l'albero, ossia quali sono i nodi foglie. Cioè, dato un nodo j, come facciamo a sapere se ha figli oppure no? Con la regola sopra possiamo solo supporre che un nodo j abbia sempre figli se 2j e 2j+1 stanno nell'array, cioè sono minori o uguali a MAX. E se invece l'albero ci serve più piccolo? Ci sono due modi:

1. Se l'albero è *completo* allora tutte le caselle sono piene fino ad un certo punto dell'array. Basta tenere un indice che indica l'ultima posizione occupata. Quindi avremo un tipo:

```
Type  Tree  =  record
                   nodi : array[1..MAX] of Elem
                   last : int
                end
```

   In questo modo, se 2 j > last sappiamo che il nodo j-esimo è una foglia, altrimenti ha i figli (entrambi perché l'albero è completo).

   Supponiamo che l'informazione contenuta nei nodi (cioè l'etichetta) sia semplicemente un carattere e vediamo un esempio:



L'albero disegnato sopra si rappresenterebbe con:

indici:	1	2	3	4	5	6	7	----------	MAX
	B	**A**	**B**	**R**	**D**	**A**	**Z**	*"niente"*	

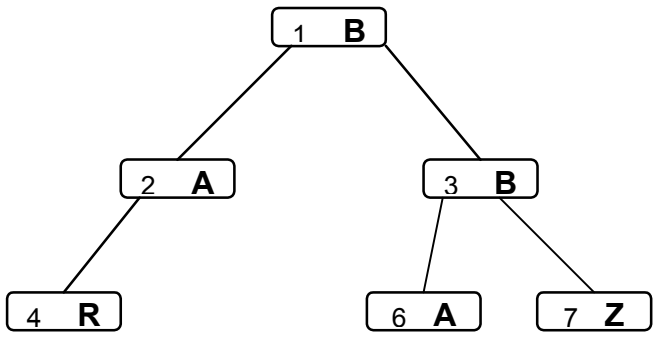
last

1. Il tipo usato per rappresentare Elem ci consente di definire un valore IND (che sta per "indefinito") che non è un valore valido per Elem. Allora possiamo usare questo valore come "tappo", ossia per marcare caselle che corrispondono a figli inesistenti. Questo vale per alberi anche non completi. In questo caso, si definisce semplicemente:

   Type Tree = array[1..MAX] of Elem

   Se a un nodo j manca il figlio sinistro [destro] si scrive IND nella cella 2j [2j+1]. In questo modo quando troviamo che un figlio di un nodo vale IND sappiamo che quel figlio è assente, quindi per quel che riguarda quel ramo l'albero finisce lì.

   Vediamo anche questo con un esempio:



L'albero disegnato sopra si rappresenterebbe con:

indici:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	----------	MAX
	B	**A**	**B**	**R**	**IND**	**A**	**Z**	**IND**	**IND**	**IND**	**IND**	**IND**	**IND**	**IND**	**IND**	*"niente"*	

Notiamo che in questo modo avremmo molte caselle a IND. Per evitare di sprecare spazio, si può combinare questa regola con la precedente:

Un nodo j non ha il figlio sinistro [destro] se e solo se 2j [2j+1] > MAX oppure la casella 2j [2j+1] vale IND.

Notiamo che non abbiamo definito nessun tipo per i nodi. In effetti, i nodi dell'albero sono le caselle dell'array. Si può definire allora:

```
Type  Node  =  integer
```

In questo caso (diversamente dalle implementazioni con puntatori) un nodo ha senso solo nel contesto di un dato albero: non possiamo ricavare informazione da un nodo solo.

Per valori di K superiori a due si fa un discorso analogo ma più complicato. Essenzialmente per K fissato bisogna capire come funziona la regola che stabilisce dove mettere i figli del nodo j (che non saranno più ai posti 2j e 2j+1 ma altrove). **Per esercizio:** provare il caso K=3.

Per il resto le cose sono analoghe (discorsi sull'albero completo, valore IND, ecc.…).

12.3.2 Implementazione delle operazioni (solo per alberi binari completi)

Vediamo l'implementazione nel caso semplice senza valore IND ma usando l'indice last per sapere in che punto dell'array finisce l'albero.

Costruttori:

Conviene pensare di allocare le variabili albero che servono semplicemente dichiarandole da programma. In questo caso i costruttori si limitano a scrivere i valori appropriati nelle variabili che vengono passate come parametri OUT.

Nota: Sarebbe possibile in alternativa un'implementazione basata su allocazione dinamica: in questo caso il tipo Tree non dovrebbe essere il record definito sopra, ma un puntatore a quel tipo di record. In questo modo, i costruttori possono essere funzioni che allocano il record, ne inizializzano i campi opportunamente e restituiscono come risultato l'indirizzo del record stesso. Provare **per esercizio**.

Dato che l'allocazione degli alberi si fa fuori dalle primitive, le operazioni saranno implementate attraverso procedure e il risultato sarà messo in un parametro OUT.

La Empty_tree come al solito è molto semplice:

```
procedure Empty_tree ( t : Tree  OUT)
{
    t.last ← 0
}
```

Dato che i nodi sono interi e la radice di un albero è sempre 1, non ha senso passare un parametro di tipo Node per costruire un albero con la Mk_tree. Quindi adottiamo la definizione altrenativa già vista in precedenza: Mk_tree : Elem x Tree x Tree → Tree cioè specifichiamo la radice attraverso il suo contenuto.

Dobbiamo tenere conto di possibili errori:

- Può accadere che lo spazio a disposizione sia insufficiente spazio per costruire l'albero (cioè l'albero risultante avrebbe più di MAX nodi).
- Può accadere che i due sottoalberi non siano alti uguali: in questo caso il risultato non sarebbe un albero completo.

Aggiungiamo il solito parametro Booleano OUT per segnalare gli errori. Quindi avremo il prototipo:

```
procedure Mk_tree ( e : Elem  IN,  t1, t2 : Tree  IN,  t : Tree  OUT,  err : Boolean  OUT)
```

L'unica parte complicata di questa procedura consiste nel copiare i nodi dei due sottoalberi t1 e t2 nelle posizioni corrette nel nuovo sottoalbero t. Per fare questo conviene scandire i sottoalberi per livelli:

- I nodi di ciascun livello l di t1 adranno copiati in t ad occupare le prime posizioni del livello l+1
- I nodi di ciascun livello l di t2 adranno copiati in t ad occupare le ultime posizioni del livello l+1

Utilizziamo due contatori annidati: lev che scandisce i livelli (il livello lev di t1 e t2 e il livello lev+1 di t) e i che scandisce i nodi all'interno di un livello. Per come è costruito l'albero, il livello lev comincia con la casella 2^{lev} e finisce alla casella 2^{lev+1}-1.

```
procedure Mk_tree ( e : Elem  IN,  t1, t2 : Tree  IN,  t : Tree  OUT,  err : Boolean  OUT)
var i, lev : integer
{   err ← False
    if  (t1.last + t2.last + 1 > MAX)  or  (t1.last ≠ t2.last)  then
    {   err ← True
        scrivi( "Errore: spazio insufficiente o sottoalberi sbilanciati" )
    }
    else
    {   t.last ← t1.last + t2.last + 1
        t.nodi[1] ← e
        lev ← 0
        while (2^(lev+1)-1) ≤ t1.last do          lev itera sui livelli dei sottoalberi
            {   per i = 2^lev ,……, 2^(lev+1)-1 :   i itera sui nodi in un livello
                {
                    t.nodi[2^lev+i] ← t1.nodi[i]
                    t.nodi[2^(lev+1)+i] ← t2.nodi[i]
                }
                lev++
            }
    }
```

Selettori:

Sono tutti molto semplici a parte Sottoalbero (che non vediamo) che ci costringe a fare un'operazione inversa a quella vista sopra per Mk_tree (provare **per esercizio**, non è facilissimo).

Nel caso della Root dobbiamo ritornare l'indice della radice, che è sempre 1. Serve il solito parametro errore per trattare il caso di albero vuoto.

```
function Root( t : Ttree,  err : Boolean) : Node
{    err ← False
     if t.last = 0 then
     {    err ← True
          scrivi( “Errore: albero vuoto” )
          return(0)
     }
     return(1)
}
```

Le

operazioni padre e figlio si implementano attraverso la regola vista prima:

```
function Padre( t : Ttree,  n : Node, err : Boolean OUT) : Node
{    err ← False
     if (n < 1) or (t.last < n) then
     {    err ← True
          scrivi( “Errore: nodo indefinito” )
          return(0)
     }
     return(n div 2)
}

function Figlio_sx( t : Ttree,  n : Node, err : Boolean OUT) : Node
{    err ← False
     if (n < 1) or (t.last < n) then
     {    err ← True
          scrivi( “Errore: albero vuoto o nodo indefinito” )
          return(0)
     }
     if   t.last < 2n     then return(0)
     else      return(2n)
}

function Figlio_dx( t : Ttree,  n : Node, err : Boolean OUT) : Node
{    err ← False
     if (n < 1) or (t.last < n) then
     {    err ← True
          scrivi( “Errore: albero vuoto o nodo indefinito” )
          return(0)
     }
     if   t.last < 2n+1       then return(0)
     else      return(2n+1)
}
```

Ci serve ancora una primitiva selettore per conoscere il valore di un nodo:

Value : Tree x Node → Elem

Qui si vede la necessità di passare anche l'albero come parametro (diversamente dall'implementazione con puntatori) in quanto il nodo da solo non ci consente di sapere il suo contenuto:

```
function Value(t : Tree  IN, n : Node  IN,  err : Boolean  OUT) : Elem
{    err ← False
     if (n < 1) or (t.last < n) then
     {    err ← True
          scrivi( "Errore: albero vuoto o nodo indefinito" )
          exit()
     }
     return(t.nodi[n])
}
```

Proprietà: sono molto facili da implementare. Lasciate **per esercizio.**

13 Tipi di dato costruiti sugli insiemi

Consideriamo un insieme (anche infinito) detto *universo* formato di elementi detti *atomi*. Consideriamo come oggetti i **sottoinsiemi finiti** di atomi dell'universo, con le operazioni più frequenti: insieme vuoto, è_vuoto?, ricerca di un elemento, inserimento di un elemento, cancellazione di un elemento, unione, intersezione, differenza, ecc..... In genere, un atomo può comparire al più una sola volta in un dato sottoisieme, nel senso che non si ammettono duplicati.

Inoltre sull'universo può essere definita una relazione d'ordine lineare < ossia dati a, b, c atomi:

- Solo una delle seguenti relazioni è vera: a < b, oppure a = b, oppure b < a (antisimmetria)
- Se a < b e b < c, allora a < c (transitività).

In questo caso si possono anche definire operazioni come minimo, massimo, successore, predecessore, ecc....

Sia Elem l'universo e Set(Elem) l'insieme di tutti i possibili sottoinsiemi di Elem (nel seguito scriviamo semplicemente Set).

Elenchiamo un insieme di operatori generale. I vari tipi di dato si otterranno prendendone solo alcuni o, eventualmente, aggungendone altri.

Costruttori:

1. Empty_Set : → Set — operazione costante, fornisce l'insieme vuoto
2. Insert : Elem x Set → Set — inserisce un nuovo elemento in un insieme

Selettori:

3. Delete : Elem x Set → Set — cancella un elemento da un insieme
4. Min : Elem x Set → Elem — fornisce l'elemento minimo di un insieme ordinato
5. Max : Elem x Set → Elem — fornisce l'elemento massimo di un insieme ordinato
6. Union : Set x Set → Set — calcola l'unione di due insiemi
7. Intersection : Set x Set → Set — calcola l'intersezione di due insiemi
8. Difference : Set x Set → Set — calcola la differenza di due insiemi

Proprietà:

9. Is_Empty_Set : Set → Boolean indica se un insieme è vuoto oppure no
10. Member : Set x Elem → Boolean indica se un elemento appartiene all'insieme oppure no

13.1 Implementazione per insiemi "piccoli"

Se l'universo è fatto di un numero piccolo di elementi (ad esempio: le lettere dell'alfabeto, i calciatori di una squadra, ...) allora è possibile utilizzare un'implementazione molto semplice ed efficiente, detta *bit-vector*. Poniamo che l'universo sia composto di N atomi e assegnamo un numero d'ordine, da 1 a N, a ciascun atomo. Allora un sottoinsieme dell'universo si può rappresentare con un array di N valori Booleani:

Type Set = array [1..N] of Boolean

Interpretiamo la cosa nel seguente modo: data una variabile s di tipo Set, mettiamo True nelle caselle di s corrispondenti ai numeri d'ordine di elementi che stanno nell'insieme che vogliamo rappresentare e False in tutte le altre. Quindi un elemento che ha come numero d'ordine i sta in s se e solo se s[i] vale True.

Assumiamo di avere due funzioni:

function Ord(e : Elem) : integer che dato un elemento ne fornisce il numero d'ordine;

function Val(i : integer) : Elem che dato un numero d'ordine restituisce l'elemento corrispondente.

In questo modo l'implementazione delle primitive è molto facile:

```
Procedure Empty_Set( s : Set OUT)
var i : integer
{   per i = 1,.....,N : s[i] ← False    }

Procedure Insert( s : Set IN-OUT, e : Elem IN)
{   s[ord[e]] ← True}

Procedure Delete( s : Set IN-OUT, e : Elem IN)
{   s[ord[e]] ← False   }
```

Per Min e Max assumiamo che valga x < y ⇔ Ord(x) < Ord(y), cioè la funzione Ord rispecchia l'ordine sull'universo. Allora:

```
Function Min( s : Set  IN,  err : Boolean  OUT) : Elem
var i : integer
{   err ← False
    per i=1,.....,N : if s[i] then return Val(i)
    err← True
}
```

Max è analoga ma si scandisce dall'ultimo elemento al primo

```
Procedure Union(s1, s2 : Set   IN,  s : Set    OUT)
var i: integer
{   per i=1,....,N : s[i] ← s1[i] or s2[i]    }
```

Intersection e Difference si implementano allo stesso modo sostituendo or con and oppure and not, rispettivamente.

```
function Is_Empty_Set( s : Set  OUT) : Boolean
var i : integer
{    per i = 1,.....,N : if s[i] then return(True)
     return(False)
}

function Member( s : Set  IN,  e : Elem  IN) : Boolean
{    return( s[Ord(e)] )   }
```

13.2 Implementazioni semplici per insiemi "grandi"

Se l'universo è troppo grande per poter adottare l'implementazione bit-vector, i modi più semplici per implementare i set sono tramite **array di Elem** oppure **liste di Elem**. I tipi utilizzati sono gli stessi del tdd successioni (cfr Sezione 6) e anche le operazioni primitive assomigliano abbastanza. Valgono i soliti discorsi:

- Con gli array si possono solo mantenere insiemi di al più K elementi con K fissato a priori, mentre con le liste non ci sono limitazioni.
- Se si aggiunge o si cancella un elemento (non in coda) con gli array è necessario spostare in avanti tutti gli elementi successivi, mentre con le liste non è necessario.
- Se si mantiene l'array ordinato è possibile cercare un elemento (operazione Member) utilizzando l'algoritmo di bisezione che è molto efficiente, mentre con le liste è necessario visitare la lista in modo sequenziale.

In entrambi i casi ci sono due varianti:

1. Mantenere ordinato l'insieme (array o lista): in questo caso gli elementi vengono inseriti nell'insieme "al posto giusto", ossia in modo che il primo della lista (o array) sia il più piccolo e così via.
2. Mantenere l'insieme disordinato: in questo caso non ci si preoccupa di come si inserscono gli elementi (per esempio, si possono mettere sempre in testa o in coda) ma costerà più fatica trovarli.

13.2.1 Implementazione con array

Il tipo è lo stesso che abbiamo usato per le successioni (vedere Sezione 6.):

```
Type   Set =   record
                   info : array [1..K] of Elem
                   lg : integer
               end
```

Vediamo solo le primitive meno banali:

Insert:

1. Con array ordinato. Supponiamo che l'operatore relazionale < si applichi direttamente agli elementi di Elem:

```
procedure Insert( s : Set  IN-OUT,  el : Elem  IN,  err : Boolean OUT)
var i ,j : integer
{    if (s.lg = K) then
     {    err ← True
          scrivi( “Errore: spazio insufficiente”)
          exit()
     }
     err ← False
     i ← 1
     while (i ≤ s.lg) and (s.info[i] < el) do
          i++                                  avanzo finché trovo elementi minori di el
     if (i>s.lg) or (s.info[i] ≠ el) then      non devo inserire el se c’é già
     {    per j=s.lg,….,i : s.info[j+1] ← s.info[j]   sposto in avanti tutti gli elementi seguenti
          s.info[i] ← el
          s.lg++
     }
}
```

1. Con array non ordinato: facciamo un’implementazione semplice che può inserire un elemento anche se è già presente (può sprecare spazio ma non compromette le altre primitive). Un’implementazione più accurata dovrebbe prima scandire tutta la lista per sincerarsi che l’elemento da inserire non ci sia già (più costosa).

```
procedure Insert( s : Set  IN-OUT,  el : Elem  IN,  err : Boolean OUT)
{    if (s.lg = K) then
     {    err ← True
          scrivi( “Errore: spazio insufficiente”)
          exit()
     }
     err ← False
     s.lg++
     s.info[s.lg] ← el
}
```

Delete:

1. Con array ordinato: supponiamo di avere a disposizione una funzione di ricerca binaria

   function RicBin(a : array [1..K] of Elem, el: Elem, inf,sup :integer) :integer

   leggermente diversa da quella definita in Sezione 2.4: invece di restituire vero/falso restituisce la posizione dell’elemento cercato nell’array. Se l’elemento non c’e’ restituisce 0 (l’implementazione di questa procedura è immediata a partire da quella già vista).

```
procedure Delete( s : Set  IN-OUT,  el : Elem  IN)
var i, j : integer
{    i ← RicBin(s.info, el, 1, s.lg)
     if (i>0) then
     {    per j=i,….,s.lg-1 : s.info[i] ← s.info[i+1]
          s.lg--
     }
}
```

1. Con array non ordinato: la ricerca binaria va sostituita con una ricerca sequenziale che scandisce la lista dall'inizio. Il resto è uguale. Pseudo-codice **per esercizio**.

Member:

1. Con array ordinato:

```
function Member(s : Set, el : Elem) : Boolean
{    return(RicBin(sinfo, el, 1, s.lg) ≠ 0)  }
```

2. Con array non ordinato: bisogna fare ricerca sequenziale. Pseudo-codice **per esercizio**.

Union:

1. Con array ordinato: si tratta di scandire in parallelo i due array in input. Si inserisce ogni elemento che si incontra avanzando ad ogni passo solo sull'array che contiene l'elemento più piccolo. Se l'elemento si trova in entrambi gli array si avanza su entrambi.

```
procedure Union( s1, s2 : Set   IN, s : Set   OUT, err : Boolean OUT)
var i, ,ii, j, k : integer
{    err ← False
     i ← 1;    j ← 1;    k ← 1
     while (i ≤ s1.lg) and (j  ≤ s2.lg) and (k ≤ K) do
     {    if(s1.info[i]<s2.info[j]) then
          {    s.info[k] ← s1.info[i]
               i++; k++
          }
          else
          {    s.info[k] ← s2.info[j]
               k++
               if (s1.info[i]=s2.info[j]) then i++
               j++
          }
     }
     per ii=i,….,s1.lg :
          if k≤ K then { s.info[k] ← s1.info[ii]; k++ }
          else  {   err ← True;  scrivi( "Errore: spazio insufficiente");exit()    }
     per ii=j,….,s2.lg :
          if k≤ K then { s.info[k] ← s1.info[ii]; k++ }
          else  {   err ← True;  scrivi( "Errore: spazio insufficiente");exit()    }
     s.lg ← k-1
}
```

La procedura è analoga a quella di Merge già vista nel contesto del MergeSort. L'unica differenza è che qui si devono evitare i duplicati. Questo implica anche che il controllo sull'errore non si può fare all'inizio (non basta verificare che s1.lg + s2.lg non superi K) perchè la dimensione del risultato non è necessariamente uguale alla somma delle dimensioni degli insiemi in input: infatti, gli elementi presenti in entrambi gli insiemi s1 e s2 vegono contati una volta sola. Questo controllo si fa sul contatore k: se supera K il ciclo while termina. I due cicli for che lo seguono fanno ancora questo controllo ed eventualmente generano segnalazione di errore.

1. Con array non ordinato: se si accettano duplicati basta copiare di seguito prima tutto s1 e poi tutto s2. Se non si accettano duplicati è più difficile e molto più inefficiente: si copia prima tutto s1 e poi si scandisce s2; a ogni passo bisogna scandire s1 per vedere se l'elemento corrente di s2 è già stato inserito oppure no. In questo caso questa implementazione è sconsigliabile.

Intersection e **Difference** si implementano in modo simile: **per esercizio**.

13.2.2 Implementazione con liste dinamiche

È molto facile da realizzare sulla base di quanto visto nella sezione precedente e in quella sulle successioni:

- **Member:** si ottiene con una ricerca sequenziale. Nel caso con lista ordinata la scansione si ferma quando incontra l'elemento cercato o un elemento più grande; nel caso non ordinato la scansione si ferma quando incontra l'elemento cercato o la fine della lista.
- **Insert:** nel caso ordinato si ottiene combinando la ricerca sequenziale (come sopra) con un inserimento simile all'inserimento in coda visto per le successioni: invece di essere in coda l'inserimento si fa "in mezzo", nel punto dove si ferma la ricerca. Nel caso non ordinato si può tranquillamente inserire in testa, come nelle successioni. Anche qui ci sarebbe il problema di controllare i duplicati.
- **Delete:** si ottiene combinando la ricerca sequenziale (come sopra) con una cancellazione simile a quella fatta per la primitiva coda (con modifica della lista) vista per le successioni. L'unica differenza è che la cancellazione invece di essere fatta all'inizio si fa in mezzo, nel punto dove si ferma la ricerca.

Pseudo-codici **per esercizio** (vedere anche Aho, Hopcroft, Ullmann – Algoritmi e Strutture Dati).

13.3 Dizionario

Il tipo di dato dizionario è un tipo Set con un insieme molto limitato di operazioni. Essenzialmente possiamo aggiungere e togliere elementi e controllare se un dato elemento appartiene al dizionario oppure no. Le primitive sono quindi le seguenti:

1. Empty_Set : → Set
2. Insert : Elem x Set → Set
3. Delete : Elem x Set → Set
4. Member : Set → Boolean

Noi vedremo il tdd "nudo", cioè privo di informazioni aggiuntive sugli elementi del dizionario. In genere, nelle applicazioni, sarà invece necessario arricchire ogni elemento x che sta nel dizionario con un insieme di informazioni utili al programma quando cerca x. Ad esempio, se stiamo progettando un archivio clienti per un professionista, gli elementi saranno ad esempio i nomi dei clienti, ma associato a ciascun nome ci saranno diverse informazioni (indirizzo, telefono, numero di pratica, ecc…..). In questo caso, il nome x di un cliente si dice *chiave,* in quanto è l'elemento che viene utilizzato per fare le ricerche nel dizionario. Per ottenere le informazioni aggiuntive sarà sufficiente modificare l'operazione Member in modo che, oltre al risultato Booleano, restituisca tali informazioni se l'elemento viene trovato.

Ovviamente questo si può implementare con array sequenziali e liste come visto sopra. Ci sono però anche implementazioni più efficienti. Ne vediamo due: una basata su alberi binari e una basata su tabelle *hash.*

13.3.1 Alberi binari di ricerca

I tipi utilizzati sono quelli già visti per gli alberi binari implementati con puntatori (si potrebbe fare anche con array, ma non lo vediamo).

```
Type   Node   =   record
                      info  :   Elem
                      left  :   puntatore_a_Node
                      right :   puntatore_a_Node
                   end
       Set =   puntatore_a_Node
```

Assumiamo come al solito che sia definita una relazione d'ordine totale < su Elem. La regola in cui gli elemti vengono inseriti nell'albero è la seguente:

Tutti gli elementi del sottoalbero di sinistra di un nodo n sono minori di n

Tutti gli elementi del sottoalbero di destra di un nodo n sono maggori di n

Assumendo che un albero sia costruito secondo questa regola, l'operazione Member si implementa facilmente come procedura ricorsiva:

```
function Member( x : Elem, s : Set) : Boolean
{
    if  s = null  then return(False)
    if  x = s->info then return(True)
    if  x < s->info then return( Member( x,s->left ) )
    else return( Member( x,s->right ) )
}
```

La procedura di inserimento si implementa in modo simile. L'idea fondamentale è di inserire un nuovo elemento sempre come foglia dell'albero. Si scende lungo l'albero secondo lo stesso percorso seguito dalla Member. Se l'elemento da inserire viene trovato allora non bisogna fare niente. Se non viene trovato si raggiunge un punto dell'albero in cui il percorso si ferma (nel senso che manca il figlio corrispondente alla direzione scelta nella ricerca): in questo caso il nuovo elemento viene aggiunto nel posto del figlio mancante.

```
procedure Insert( x : Elem  IN, s : Set  IN-OUT)
{
    if  s = null  then
    {
        new(s)
        s->info ← x
        s->left ← null
        s->left ← null
    }
    else if x < s->info then
        Insert(x,s->left)
    else if x > s->info then
        Insert(x,s->right)
                          se x = s->info non si fa niente
}
```

L'operazione Delete è quella che presenta più problemi di implementazione. Infatti, mentre è facile cancellare una foglia dell'albero, abbiamo il problema di capire come fare a togliere un nodo interno senza compromettere la struttura. Il punto è che se eliminiamo un nodo interno n avremmo come effetto collaterale di "sganciare" dall'albero i sottoalberi di n. Se n ha un solo figlio (quindi un solo sottoalbero) il problema si risolve facilmente "tirandolo su" nel senso che si mette tale figlio al posto di n (tirandosi dietro tutto il sottoalbero). Se n ha due figli non sappiamo come fare. Per ora accantoniamo il problema nel caso di due figli e vediamo un caso facile, che ci garantisce di avere (al più) un solo figlio del nodo da eliminare.

Consideriamo l'elemento minimo m di un insieme: questo non può avere un figlio sinistro, perché altrimenti esisterebbe un elemento inferiore, contraddicendo il fatto che m sia il minimo. Quindi o m è una foglia o ha solo il figlio destro. Vediamo una funzione che cancella l'elemento minimo di un insieme e lo restituisce in output. Questa funzione resterà "privata" all'implementazione del tdd nel senso che non sarà fornita tra le operazioni del tdd, ma potrà essere utilizzata per la loro implementazione.

```
function DeleteMin(s : Set  IN-OUT) : Elem
var e : Elem, s1 : Set
{
    if s->left = null then              s punta all'elemento più piccolo
    {   e ← s->info
        s1 ← s
        s ← s->right                elimino il minimo "tirando su" il suo figlio destro
        free(s1)
        return(e)
    }
    else    return(DeleteMin(s->left))    cerco l'elemento da eliminare nel sottoalero sinistro
}
```

Adesso vediamo di affrontare il caso generale nel modo seguente: se vogliamo cancellare un nodo x prima di tutto cerchiamo la sua posizione sull'albero. Se non lo troviamo non abbiamo niente da fare. se lo troviamo possono darsi tre casi:

a) x è una foglia: allora si può cancellare senza problemi liberando il record e mettendo a null in puntatore corrispondente del padre;

b) x ha un solo figlio: allora si può cancellare "tirando su" il sottoalbero corrispondente, come fatto sopra;

c) x ha due figli: in questo caso consideriamo il sotoalbero destro tr di x, che contiene tutti valori più grandi di x, ma più piccoli di quelli presenti nel suo sottoalbero sinistro. Si cancella l'elemento minimo y di tr (usando la DeleteMin) e si mette y al posto di x. Vediamo perché funziona:

   d) l'albero risultante contiene tutti i nodi dell'albero di partenza tranne x: infatti x è stato tolto e y è stato prima tolto e poi reinserito

   e) la struttura dell'albero è consistente con l'ordinamento. Bisogna dimostrarlo. L'unica parte che è stata modificata riguarda la parte di albero tx che aveva come radice x e come sottoalberi sinistro e destro tl e tr, rispettivamente. Sia tr' l'albero ottenuto da tr togliendo y, allora tx è stato sostituito da ty, che è un albero con radice in y e come sottoalberi sinistro e destro tl e tr', rispettivamente. Sia r il padre di x (ammesso che esista, altrimenti x era la radice dell'albero, quindi ora ty rappresenta tutto l'albero). Se x era figlio sinistro [destro] di r significa che tutti gli elementi di tx erano più piccoli [grandi] di r. Questo vale a maggior ragione per ty, in quanto contiene gli stessi elementi di tx tranne x. Quindi tutto l'albero risultante è consistente se e solo se ty è consistente. Ora, tl e tr' sono sicuramente consistenti: infatti tl lo era già prima, tr' è stato ottenuto da un albero consistente eliminando un elemento in modo consistente (casi a o b). Inoltre, sappiamo che y stava in tr, quindi: y > x, a maggior ragione y > z per qualsiasi z di tl. Sappiamo anche che y era l'elemento minimo di tr, allora y > w per qualsiqsi w di tr'. Quindi anche ty è consistente.

Vediamo lo pseudo-codice:

```
procedure Delete(x : Elem  IN,  s : Set  IN-OUT)
var s1 : Set
{    if s ≠ null then                    se s è vuoto non abbiamo niente da fare
         if x < s->info then
              Delete(x, s->left)         scendiamo ricorsivamente nell'albero sinistro
         else  if x > s->info then
              Delete(x, s->right)        scendiamo ricorsivamente nell'albero destro
         else if (s->left = null) and (s->right = null) then          trovato x: è una foglia
         {    free(s)
              s ← null
         }
         else if s->left = null          trovato x: non ha sottoalbero sinistro
         {    s1 ← s
              s ← s-right
              free(s1)
         }
         else if s->right = null              trovato x: non ha sottoalbero destro
         {    s1 ← s
              s ← s->left
              free(s1)
         }
         else                            trovato x: ha entrambi i sottoalberi
              s->info ← DeleteMin(s->right)
}
```

Un albero si dice **bilanciato** se tutti i nodi non pieni (cioè a cui mancano figli) stanno sugli ultimi due livelli. In questo caso le foglie possono differire al più di un livello e l'albero è il più largo possibile.

Su alberi bilanciati le operazioni Insert, Delete e Member funzionano in modo molto efficiente: il numero di passi necessario a compiere un'operazione è logaritmico nel numero di nodi dell'albero.

Se l'albero viene costruito prendendo elementi in maniera casuale, in media l'albero verrà bilanciato o quasi, ma se siamo particolarmente sfortunati (ad esempio se la sequenza in input per costruire l'albero è perfettamente ordinata) l'albero potrebbe però venire completamente sbilanciato (al limite, equivalente ad una lista). Gli alberi sbilanciati influiscono negativamente sull'efficienza delle operazioni.

Definiamo intuitivamente un "passo" fatto da un'operazione (Insert, Delete, Member) come l'insieme delle istruzioni necessarie per passare da un nodo al figlio. Se l'albero è bilanciato, il numero di passi per fare un'operazione sarà logaritmico nel numero di nodi dell'albero. Se invece l'albero è sbilanciato, le operazioni richiedono un numero di passi proporzionale al numero di nodi nell'albero, ossia molto più elevato. Per esempio, se in albero ci sono circa 100.000 nodi per fare un'operazione saranno necessari in media circa 16 o 17 passi in un albero bilanciato e circa 50.000 in uno non bilanciato.

Ci sono implementazioni ottimizzate (alberi bilanciati di ricerca, anche detti *alberi AVL*), che non vediamo, che garantiscono che l'albero rimanga bilanciato dopo ogni operazione di Insert e Delete. Con queste implementazioni siamo sempre sicuri di avere un'efficienza ottimale. Per gli interessati: vedere in biblioteca Horowitz e Sahni: Fundamentals of Data Structures, Sez.9.2.

13.3.2 Tabelle hash

Ci sono due versioni che vertono su un'idea fondamentale:

- l'insieme viene rappresentato da una grande tabella di elementi;
- dato un nuovo elemento x abbiamo una funzione *h(x)* in grado di calcolare la posizione (codice hash) in cui si trova, o va inserito, x nella tabella.

La funzione h non è iniettiva, nel senso che diversi elementi possono generare lo stesso codice hash.

Sorge quindi il problema di gestire i conflitti, cioè il fatto che potremmo trovarci a dover mettere due o più elementi nello stesso posto. Le due implementazioni differiscono fondamentalmente per la strategia usata nella gestione dei conflitti.

13.3.2.1 Hashing aperto

L'idea è la seguente: abbiamo una tabella fatta di un certo numero di "scatole" (dette *bucket*), ciascuna delle quali può contenere un numero più o meno grande (eventualmente illimitato) di elementi. La funzione hash ci consente di trovare la scatola giusta dove cercare/mettere l'elemento che ci interessa. All'interno di ogni scatola gli elementi sono tenuti come in un set, quindi ordinati oppure alla rinfusa, secondo l'implementazione. Perciò, in questo caso la gestione dei conflitti si fa semplicemente prevedendo che in ogni posto ci possa stare più di un elemento (ossia, un insieme di elementi).

In pratica abbiamo frazionato l'insieme in tanti sottoinsiemi, ciascuno dei quali si mette in una scatola. La funzione hash ci fa trovare (velocemente) la scatola e poi dobbiamo cercare in un sottoinsieme (molto) più piccolo di quello totale.

Esempio: Pensiamo di voler costruire uno schedario per un insieme di persone (es: gli studenti dell'Università di Genova). Potremmo costruire un mobile con molti cassetti, ad esempio uno per ogni lettera dell'alfabeto, e mettere dentro ogni cassetto le schede di tutti gli studenti il cui cognome comincia per la lettera corrispondente. Quindi la funzione hash sarebbe semplicemente: data una stringa s, h(s) è il primo carattere di s. Avremmo ben presto sorprese spiacevoli: i cassetti delle lettere H, J, K, X, Y, Z sono ancora vuoti mentre il cassetto della C è già pieno e ci rimangono ancora un sacco di schede da archiviare nella C. Soluzioni?

- Potremmo smontare tutto e costruire un mobile con cassetti più grandi, ma non sembra molto furbo: dopo un altro po' saremmo da capo.
- Potremmo usare cassetti di profondità infinita: con le schede cartacee non possiamo ma in informatica si può, ad esempio utilizzando liste dinamiche (tuttavia il concetto di "infinito" è solo virtuale: le memorie e i dischi sono pur sempre finiti). Resta il problema dei cassetti vuoti o quasi che sono "sprecati" e di quelli molto pieni che ci fanno perdere un sacco di tempo per cercare i documenti o per mantenerli ordinati.
- Ridistribuire le scede in modo diverso, cioè cambiare funzione hash, in modo che il numero di schede sia approssimativamente lo stesso in ogni cassetto. Negli schedari fisici (utilizzati dagli umani) si tende a prevedere quanto saranno affollati i cassetti: ad esempio ci saranno diversi cassetti per la C (esempio: da CA a CEC, da CED a CIO, da CIP a COB, da COC a CUZ) e magari uno solo per W, X, Y e Z. Non è detto che la scelta fatta non costringa a ricambiare tutto dopo un po'. Comunque non in tutti i casi è così facile risolvere il problema.

La strada generale scelta per le tabelle hash è cercare una funzione hash che generi un valore il più possibile "casuale". Vediamo un modo semplice. Nell'esempio abbiamo 26 cassetti. Potremmo assegnare un numero d'ordine a ciascuna lettera dell'alfabeto (ad esempio il loro codice ASCII) e calcolare il codice hash per uno studente x nel modo seguente: sommiamo il numero d'ordine di tutti i caratteri nel nome di x, dividiamo il risultato per 26, prendiamo il resto e aggiungiamo 1. Il risultato è sempre un numero da 1 a 26, che ci permette di scegliere un cassetto. In questo modo la funzione è univoca (dà sempre lo stesso risultato ogni volta che la calcoliamo) ed è abbastanza probabile che sparpagli le schede dei vari studenti per tutti i cassetti piuttosto che concentrarli in pochi.

Nel caso gli elementi dell'insieme siano interi, un'idea molto usata per calcolare la funzione hash consiste nel costruire h(n) prendendo le cifre centrali del numero decimale n^2. Questo va bene se il numero di bucket nella tabella è una potenza di 10. Più in generale, si approssima questo procedimento nel modo seguente: se le chiavi hanno valori in un intervallo [1..K] e la tabella contiene B bucket (numerati da 0 a B-1), si sceglie C numero primo tale che BC^2 sia circa uguale a K^2 e si definisce

$$h(x) = \lfloor n^2/C \rfloor \bmod B$$

dove mod è l'operazione resto della divisione intera.

Implementazione: possiamo rappresentare la tabella con un array e appendere una lista dinamica a ogni cella dell'array. Dato un valore x da cercare/inserire/cancellare si trova tramite h(x) la cella dell'array da cui parte la lista appropriata a contenerlo e poi si fa ricerca seuenziale su quella (già vista). La gestione di ogni

lista può essere fatta mantenendo l'ordine oppure no. Se ci si aspetta che in ogni bucket vadano a finire "pochi" elementi può convenire tenere la lista disordinata.

Tutte queste valutazioni le facciamo qui intuitivamente, in realtà ci sarebbero un sacco di conti di calcolo delle probabilità per capire la bontà di una funzione hash, quant'è l'aspettativa di affollamento dei bucket, ecc.…

Avremmo quindi:

```
Type   Node   =      record
                        key : Elem
                        next : Puntatore_a_Node
                    end
       Open_Hash_Table  =   array [0..B-1] of Node
```

L'idea è: mettiamo nella lista appesa alla cella i tutti gli elementi x tali che vale h(x)=i.

Lo schema generale di operazione primitiva è:

- sia x l'elemento in input, calcola h(x)
- sia T la tabella, vai nella lista T(h(x)) ed esegui l'operazione sull'insieme che questa rappresenta

13.3.2.2 Hashing chiuso

La strategia scelta è la seguente:

- invece di permettere l'inserimento di diversi elementi in ogni cella della tabella, permettiamo l'inserimento di un solo elemento per cella;
- se, volendo inserire un nuovo elemento x, troviamo la cella h(x) già occupata, allora mettiamo x in un altro posto.

Dove lo mettiamo? Bisogna usare una tecnica di *rehashing* che ci permetta di scegliere una nuova locazione per x. E se anche questa è piena? Allor se ne cerca un'altra e poi un'altra ancora e così via, finché o ne troviamo una vuota, oppure ci rendiamo conto che la tabella è completamente piena. Quindi bisogna che il rehashing possa essere applicato iterativamente e ci consenta al limite di fare tutto il giro della tabella alla ricerca di un posto vuoto. Una soluzione molto semplice per una tabella di dimensione B è la seguente:

$$h_i(x) = (h(x) + i) \bmod B$$

Facciamo il primo tentativo con h(x), se fallisce tentiamo con $h_1(x)$, poi $h_2(x)$ e così via. Per come sono definite le funzioni hi, significa scandire la tabella in avanti (e ciclicamente) a partire da h(x) finché si trova un posto vuoto oppure si ritorna al punto di partenza (in quest'ultimo caso significa che la tabella è piena).

Per funzionare, questo meccanismo ha bisogno che la dimensione della tabella sia fissata a priori.

Implementazione: la tabella è semplicemente un array:

```
Type   Closed_Hash_Table    =   array [1..B] of Elem
```

Ovviamente, a parità di applicazione, se usiamo una tabella chiusa B assumerà un valore molto più grosso di quello utilizzato nel caso della tabella aperta (di solito almeno nell'ordine del quadrato di quest'ultimo).

Bisogna inoltre che Elem contenga due valori che non sono mai usati come elementi validi e che chiamiamo NULL_ELEM e DELETED_ELEM.

Un'insieme vuoto (generato da Empty_Set) è una tabella in cui tutte le celle siano a NULL_ELEM.

L'operazione Member funziona come segue: dato il valore x da cercare, si accede all'array con il codice hash h(x) e poi eventualmente con i codici di rehashing $h_i(x)$ finché:

- o si trova x;
- o si trova NULL_ELEM: in questo caso x non è nell'insieme.

L'operazione Delete cerca in modo analogo finché:

- trova NULL_ELEM o fa tutto il giro della tabella senza trovare x: in questo caso x non è in tabella;
- trova x: in questo caso scrive DELETED_ELEM al suo posto.

Per l'operazione Insert fa prima una ricerca analoga alla Member per controllare che x non sia già in tabella. Possono accadere tra cose:

- trova x: in questo caso non fa niente;
- non trova x ma si accorge che la tabella è piena (gioè anche $h_B(x)$ fallisce): allora segnala errore (spazio insufficiente).
- non trova x perché si ferma su un valore NULL_ELEM: potrebbe mettere direttamente x in quel posto, ma in questo modo le caselle dov'è scritto DELETED_ELEM non sarebbero mai riutilizzate (a volte conviene fare così se si sa già che le cancellazioni saranno poche). Se si vogliono riutilizzare le caselle conviene ripartire con la ricerca da capo finché si trova un valore NULL_ELEM oppure un valore DELETED_ELEM.

  Nota: non si può evitare la ricerca preventiva tipo Member e scrivere x alla prima casella DELETED_ELEM che si trova: infatti x potrebbe essere già in tabella, quindi si farebbero duplicati.

Perché ci siamo complicati la vita con il valore DELETED_ELEM? Perché se per cancellare un elemento scrivessimo EMPTY_ELEM al suo posto potremmo compromettere la struttura della tabella. Vediamo con un esempio. Inseriamo dapprima un valore x. Supponiamo che:

- il posto h(x) sia occupato da un valore y
- il posto $h_1(x)$ sia libero: ci scriviamo x

Adesso supponiamo di cancellare y, il posto h(x) si libera. Se scrivessimo EMPTY_ELEM al suo posto e più tardi cercassimo x con la Member, questa concluderebe che x non è in tabella, perche h(x) è vuoto. Quindi non si può fare così. Se scriviamo DELETED_ELEM, la Member considera la casella occupata e quindi continua la ricerca (che avrà successo). Se dopo volessimo inserire un nuovo elemento z tale che h(z)=h(x), potremmo tranquillamente ricoprire il valore DELETED_ELEM che si trova in h(x) (infatti la Insert fa così) senza causare guai nelle ricerche successive. In altre parole, le caselle non occupate hanno due scopi:

1. sono posti a disposizione per inserire elementi: questo vale sia per quelle dove c'è un valore EMPTY_ELEM che per quelle dove c'è un valore DELETED_ELEM;
2. sono segnali di fine ricerca per la Member e per la Delete: questo vale solo per quelle dove c'è un valore EMPTY_ELEM; su quelle con DELETED_ELEM la ricerca va avanti.

13.4 Discussione

Le diverse implementazioni del tipo di dato Set (dizionari compresi) vanno confrontate tenendo conto di tre criteri:

- Potere espressivo (cosa riusciamo a farci)
- Occupazione spaziale
- Efficienza computazionale
- Semplicità

Il terzo concetto sarà più chiaro in seguito (cfr Terza puntata) ma vediamo di trattarlo intuitivamente. Consideriamo quanto lavoro fa ogni algoritmo che implementa un'operazione primitiva. Quantifichiamo il

lavoro fatto in “passi”, dove un passo è un’operazione elementare (fare un’assegnazione, confrontare due valori, ec…). Il numero di passi fatto da un algoritmo può essere indipendente dall’input (allora si dice costante) oppure dipendere dalla dimensione dell’input (numero di elementi in un set o nell’universo).

Bit-vector.

Il potere espressivo è basso, si può utilizzare solo se l’universo è piccolo.

L’occupazione spaziale è alta relativamente alla dimensione dell’universo, in quanto ogni insieme richiede lo stesso spazio dell’intero universo. In assoluto, poiché l’universo è piccolo, un insieme occupa uno spazio piccolo.

L’efficienza è buona perché le operazioni più importanti si fanno in un numero costante di passi (Insert, Delete, Member). Tuttavia alcune richiedono un tempo proporzionale alla dimensiondell’universo (operazioni Booleane, Empty_Set, Min, Max, ecc…). Se l’universo è sufficientmente piccolo è possibile un’implementazione ottimizzata (ad esempio in C, indirizzando i signoli bit) in cui tutte le operazioni si fanno in tempo costante.

L’implementazione è molto semplice.

Array ordinati.

Il potere espressivo ha il limite di fissare a priori la dimensione massima K di un insieme.

L’occupazione è alta perché qualunque insieme occupa uno spazio pari a quello di dimensione massima K. Oviamente potere espressivo e occupazione sono in conflitto: più risparmiamo sull’occupazione meno cose riusciamo a rappresentare.

L’efficienza non è molto buona: Insert, Delete, Member e operazioni Booleane richiedono un numero di passi proporzionale alla dimensione dell’insieme in input.

L’implementazione è semplice.

Array non ordinati.

Potere espressivo e occupzione come sopra.

Insert si fa in un numero costante di passi, mentre Delete e Member si fano in un numero di passi proporzionale alla dimensione dell’insieme in input e quando l’elemento cerato non è presente richiedono di scandire tutto l’array. Le operazioni Booleane sono inefficienti: richiedono un numero di passi quadratico nella dimensione degli insiemi in input.

L’implementazione è semplice (noiose le operazioni Booleane).

Liste ordinate e non.

Non ci sono limiti sul potere espressivo.

Un record della lista occupa più spazio rispetto ad una cella degli array sopra, ma allochiamo solo tanti record quanti sono gli elementi in lista (non si spreca spazio) quindi l’ocupzione è molto buona.

La complessità è del tutto analoga a quella dell’implementazione con array, nei due casi ordinato e non.

L’implementazione è leggermente più laboriosa che con gli array a causa della gestione dei puntatori, tuttvia anora semplice.

Alberi di ricerca.

Nessun limite sul potere espressivo.

Occupazione leggermente più alta che con le liste (due puntatori per cella aziché uno).

Molto efficiente se l’albero è bilanciato: tutte le operazioni si fanno in un numero di passi logaritmico nella dimensione dell’insieme. Se la’lbero si sbilancia le operazioni possono richiedere un numero di passi proporzionale alla dimensione dell’insieme (cioè dell’albero).

Implementazione leggermente più laboriosa che con le liste a causa della Delete. Più difficile da capire che da fare.

Tabelle hash aperte.

Nessun limite sul potere espressivo (purché si usino le liste per implementare l'insieme all'interno di ogni bucket).

Occupazione appena più alta che con le liste a causa dell'array di bucket, in genere trascurabile rispetto alla dimensione dell'insieme rappresentato.

L'inserimento (su liste non ordinate) si fa sempre in un numero costante di passi.

L'efficienza di Member e Delete dipende da due cose:

- il bilanciamento dei bucket cioè la lunghezza relativa delle varie liste;
- il rapporto tra il numero di bucket B e il numero di elementi dell'insieme rappresentato.

Nel caso ottimale di bucket perfettamente bilanciati, se l'insieme ha k elementi, Member e Delete richiedono un numero di passi proporzionale a k/B: più bucket mettiamo, più spazio occupiamo, più efficienti diventano le primitive.

Implementazione fondamentalmente analoga a quella con liste (con qualche dettaglio in più).

Tabelle hash chiuse.

Limiti di potere espressivo, analogamente all'impelemtazione con array.

Occupazione ed efficienza vanno viste insieme. Affinché funzioni bene una tabella hash chiusa deve restare piuttosto vuota (quindi l'occupazione è alta): in questo caso in media tutte le operazioni sono motlo efficienti: si fanno in un numero costante di passi, cioè indipendente da quant'è la dimensione dalla tabella e da quanti elementi ci sono dentro. Quando la tabella comincia a riempirsi però son dolori: le operazioni primitive diventano inefficienti: funzionano in un tempo proporzionale alla dimensione della tabella. Ci sarebbero un mare di conti probabilistici per stabilire l'efficienza nel caso medio in funzione del grado di riempimento della tabella.

Implementazione semplice.

Fine della 2ª puntata