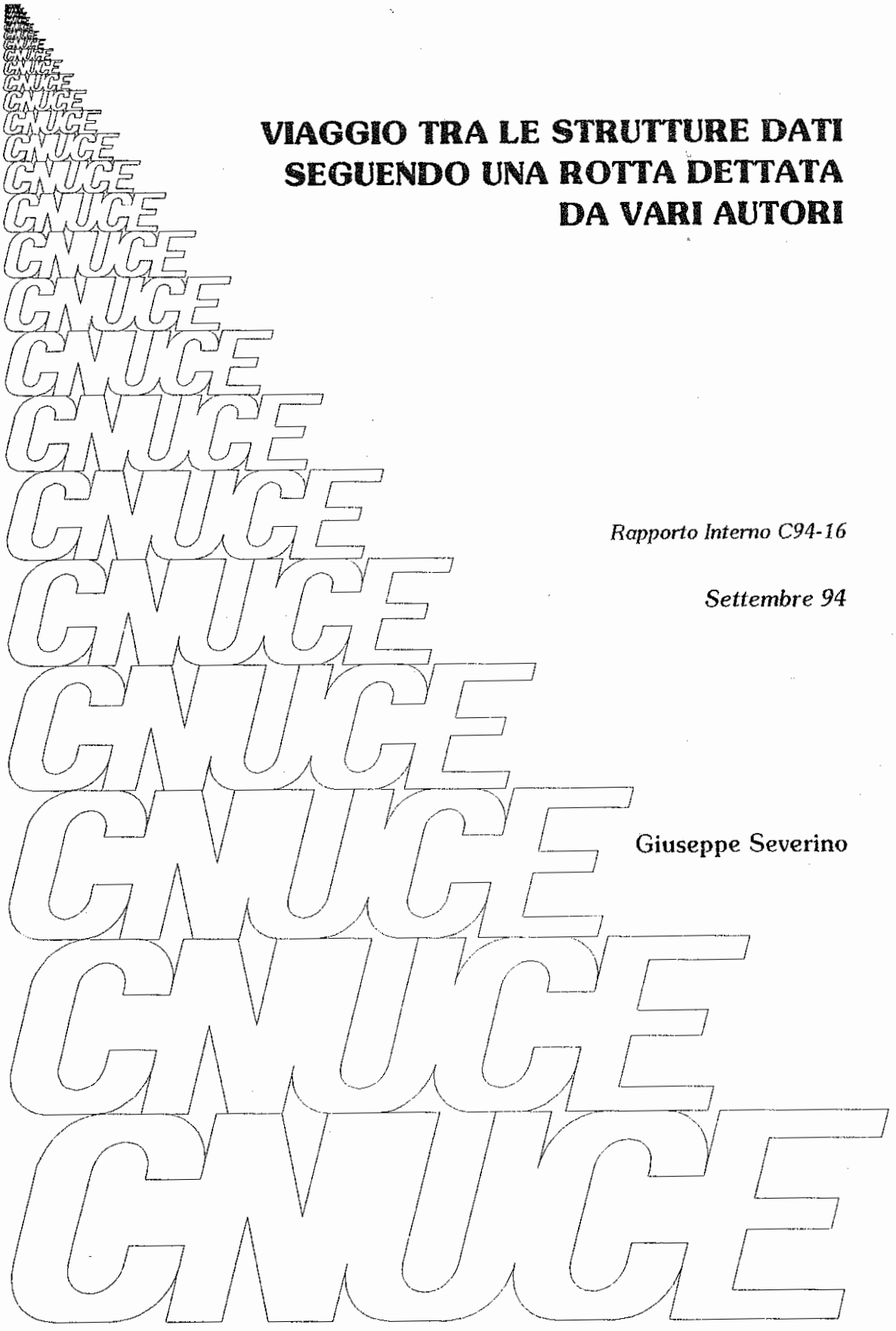
VIAGGIO TRA LE STRUTTURE DATI SEGUENDO UNA ROTTA DETTATA DA VARI AUTORI



Rapporto Interno C94-16

Settembre 94

Giuseppe Severino

Algoritmi e Dati

L' utilizzo di un qualsiasi calcolatore elettronico è subordinato alla conoscenza di un "qualcosa" capace di tradurre le nostre volontà (risoluzione dei nostri problemi) in espressioni note alla macchina o da essa riconoscibili e comunque eseguibili. Così come tra gli essere umani, la comunicazione avviene attraverso il linguaggio e quant' altro ad esso subordinato, esistono per tali apparecchiature mezzi di comunicazione detti appunto "linguaggi di programmazione" che si pongono a metà strada tra la normale concezione dell' uomo di trasmettere ordini e la capacità della macchina di eseguirli. Questi mezzi convenzionali di comunicazione consentono lo scambio di informazioni o per meglio dire il trasferimento dei dati che dovranno essere poi elaborati. Fondamentalmente essi rappresentano una macchina astratta capace cioè di simulare una macchina reale. I linguaggi di programmazione si sono con il tempo evoluti e dal macchinoso e apparentemente incomprensibile linguaggio macchina siamo arrivati a linguaggi di interazione della quarta generazione che sono anch' esse macchine astratte basate su una forma ***english-like***: si avvicinano sempre più alle espressioni della comunicazione umana. I linguaggi di programmazione sono per l' utilizzatore una astrazione dal suo consueto modo espressivo, e altrettanto vale per la macchina che riuscirà ad intercettare questi comandi codificati, interpretarli per poi eseguirli. In altre parole, l 'uomo effettuerà un' azione di astrazione dal suo normale sistema di pensiero per avvicinarsi alle esigenze della macchina; così per la macchina è previsto un meccanismo di livelli di astrazione, rispetto al suo naturale sistema di funzionamento (impulsi elettrici che determinano il funzionamento dei circuiti elettronici), per captare l' istruzione da eseguire descritta dal linguaggio di programmazione. Il computer la interpreterà, scomporrà, semplificherà, arricchirà secondo il "linguaggio" proprio del suo hardware e del sistema operativo che lo completa, per poi tradurla, definitivamente, in segnali elettrici, naturali impulsi al suo reale funzionamento. Il linguaggio di programmazione rappresenta quindi il mezzo necessario all' uomo per poter preparare i computer allo svolgimento di un determinato lavoro. Il linguaggio consente di preparare una serie di comandi sequenziali che agiranno sui dati per produrne altri (lavoro). Se ciò che viene definito nel linguaggio rappresenta quello che la macchina deve fare: -'algoritmo'- (in onore del matematico ubseco Mohammed ibn-Musa al-Khowarismi-); dove deve agire sono i 'dati' (dove compiere l' azione). L'algoritmo è allora l'insieme delle azioni che la macchina eseguirà sui dati per produrre il suo lavoro. I risultati di una elaborazione sono rappresentati dalle modifiche apportate ai dati in ingresso (input) con azioni descritte dal programma. In definitiva, il programma non è altro che l' unione tra l' insieme delle azioni da compiere (algoritmi) e i dati su cui tali azioni operano.

Tipicamente l'insieme della azioni che agiscono sui dati -l' algoritmo- è comparabile ad una ricetta di cucina. I dati di input sono rappresentati dagli ingredienti, mentre le azioni sono rappresentate dalla descrizione delle operazioni di amalgama degli ingredienti. I dati di uscita, cioè il lavoro eseguito, sono rappresentati dall' amalgama finale, magari ingentilito da estrose composizioni e impreziosito da un vassoio d 'argento da portata. *A livello di astrazione possiamo far corrispondere la descrizione delle operazioni al software, mentre gli arnesi come i tegami necessari alla preparazione o la cottura degli ingredienti, i coltelli e le posate in genere , il fornello e quant 'altro a ciò similare si associano allo hardaware (B).* La descrizione di un algoritmo non può prescindere da una descrizione dettagliata delle azioni ciascuna delle quali viene detta *azione elementare o istruzione* Le azioni elementari che compongono un algoritmo dovranno essere prive di qualsiasi ambiguità e fornire un risultato *solo, sicuro e ripetibile in un tempo limitato.* Il livello di dettaglio delle azioni elementari dipende dal tipo di esecutore a cui affidiamo il nostro algoritmo, ed essendo nel nostro caso un calcolatore elettronico ci sarà bisogno di un numero altissimo di dettagli poiché tali macchine sono semplicemente esecutori di lunghe sequenze di bit. I linguaggi di programmazione visti come macchina astratta, consentono di elevare il livello di visibilità della macchina fisica permettendo all' utilizzatore di interagire con essa attraverso i costrutti che il linguaggio di programmazione stesso gli mette a disposizione. Gli algoritmi descritti con questi *costrutti* applicabili ad un insieme di dati si chiameranno programmi. Se l' esecuzione delle istruzioni avviene una dopo l' altra diremo che eseguiamo un processo sequenziale. Gli algoritmi codificati in procedure sono generali ed evidentemente validi per un numero illimitato di dati d' ingresso. Poiché i dati da elaborare possono essere un numero arbitrario, teoricamente infinito, il tempo necessario per ottenere i risultati finali sarà direttamente proporzionale al tempo di elaborazione unitaria. Nella pratica comune, per elaborazioni complesse, è possibile richiamare da una procedura un' altra procedura e da questa un' altra ancora; tornare alla procedura chiamante, richiedere l' esecuzione di un' altra procedura, per poi avere, al termine dell' intera e complessa elaborazione, il risultato finito. In questo caso la procedura chiamante attende il termine dell' esecuzione della chiamata per poi riprendere il flusso esecutivo delle azioni elementari descritte. *Da questo punto di vista le procedure possono essere viste come costrutti dell' utilizzatore da associare a quelli del programma come se fossero parte integrante. Il meccanismo di chiamata tra procedure può prevedere la chiamata della procedura stessa. In questo caso si ha la* ***ricorsione.***

Ricorsione

Se la suddivisione di un programma in più procedure e funzioni specializzate è abbastanza intuitiva e il meccanismo del ritorno al programma chiamante chiaro, può esserlo meno l' effetto di un programma che chiama sé stesso. Intuitivamente la chiamata a sè stesso è equivalente a mettersi in mezzo a due specchi: si producono immagini riflesse nell' immagine riflessa, nell' immagine riflessa nell' ...; in maniera infinita. Proprio per evitare l' effetto infinito che si prova ponendoci tra due specchi occorre stabilire un controllo del meccanismo di chiamata attraverso l' utilizzo del costrutto condizionale che sarà il garante della chiusura dell' apparente ciclicità delle chiamate. Le chiamate interne alla stessa procedura o funzione avverrano passando dei dati di cui almeno uno sarà sempre modificato ad ogni chiamata dal programma chiamante ed è su questo dato (o dati) che il costrutto condizionale farà la sua verifica. Appena la condizione di ritorno accadrà produrrà la chiusura 'rovescia' di tutte le procedure precedentemente aperte. Per chiusura rovescia intendiamo: la prima funzione a chiudersi sarà l' ultima ad essere stata aperta, per seconda la penultima, e così via fino alla chiusura dell' ultima che sarà quella che ha innnescato il meccanismo di ricorsività.

Esempio:

```
function    fibonacci (num:integer):integer;
begin
      if n = 0
            then
                  fibonacci = 1
            else
                  fibonacci := n * fibonacci (n - 1)
end
```

Da notare come l 'utilizzo della ricorsione non garantisca miglior efficienza e tanto meno chiarezza alla struttura del programma. Il suo impiego causa una serie di chiamate primitive a strutture dati interne al sistema operativo che debbono essere tenute di conto se come detto il programma deve risultare efficiente oltre che brillante.

Tipi di dato

Tutti i linguaggi di programmazione forniscono dei tipi dato detti primitivi. Quindi ogni linguaggio forte prevede specificati e realizzati implicitamente un certo numero di dati che possono essere utilizzati dal programmatore senza bisogno di ulteriori definizioni. Nel Pascal i dati primitivi sono gli **interi** (INTEGER), i **booleani** (BOOLEAN), i **reali** (REAL), i **caratteri** (CHAR). Cerchiamo di chiarire che cosa si intende per tipo:

- un tipo specifica un dominio: insieme di possibili valori;
- un tipo prescrive una rappresentazione per i valori del suo dominio: questa rappresentazione è scelta dal compilatore;
  (Es. per gli interi valori binari con complemento a due)
- un tipo prescrive una implementazione per ogni operazione del suo insieme di operazioni: queste implementazioni sono anch'esse scelte dal compilatore, che è a sua volta vincolato dall' architettura della macchina.(Lings)

A scanso di equivoci, è opportuno distinguere tra dato e tipo di dato. Per dato si intende il valore che una variabile può assumere durante l 'elaborazione, il tipo dato è invece un modello matematico con specifiche di dominio, tipi di rappresentazione e implementazione delle operazioni possibili su quel dominio. In questo contesto possiamo proporre l'esempio utilizzato da Lings per la specifica di tipo INTEGER e la scelta del tipo più consono (appartenente alla macchina) per la sua rappresentazione definendo nel contempo una corrispondenza:

- tra gli interi ed i valori del dominio di rappresentazione;
- tra le operazioni fra gli interi e le operazioni sul dominio di rappresentazione.

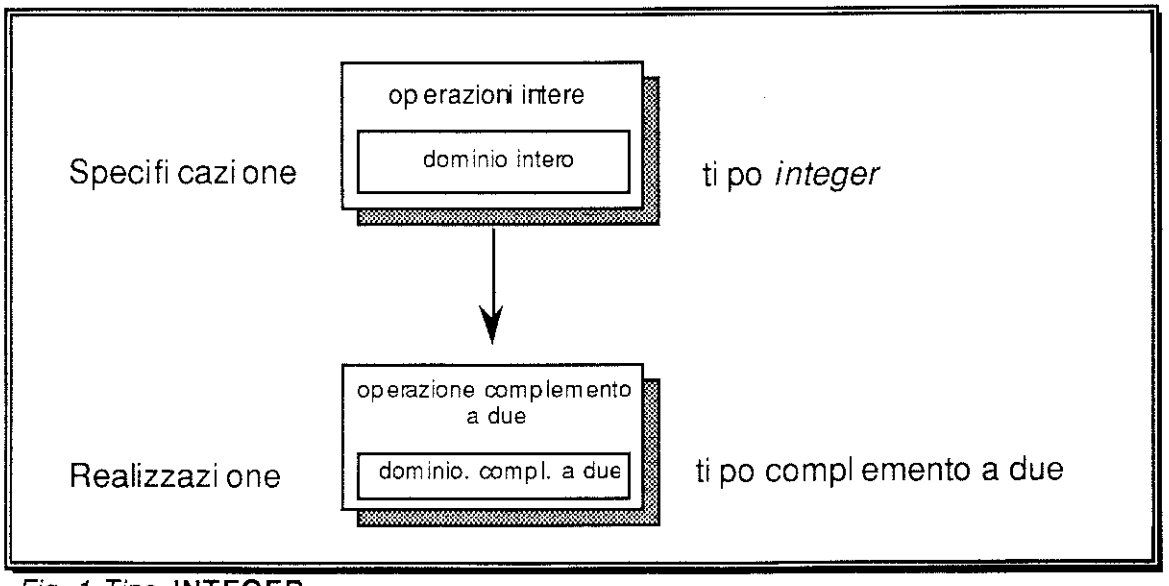


Fig. 1 Tipo **INTEGER**

Esempio:

INTEGER	COMPLEMENTO A DUE (16 bit)
Valore: -32768	1000000000000000
Valore: 0	0000000000000000
Valore: 32767	0111111111111111
Operazione: +	IADD (somma intera)
Operazione: DIV	IDIV: ritorna quoziente
Operazione: MOD	IDIV: ritorna resto

Da questo possiamo passare alla notazione implicita in tutti i programmi Pascal che usano gli interi.

Specifiche di tipo **INTEGER**

Dominio

-32678 ... 32677

Operazioni

+, -, *, DIV, MOD	:- (INTEGER,INTEGER): INTEGER
<, <=, =, >=, >	:- (INTEGER,INTEGER): BOOLEAN
:=	:- (INTEGER,INTEGER)

Semantica

Basata sugli assiomi di Peano, con le opportune modifiche a causa del dominio finito

Realizzazioni del tipo **INTEGER**

Rappresentazione

Binaria in complemento a due

Implementazione

+(X, Y)	-->	load X
		iadd Y, ecc.

Fig. 2 Tipo **INTEGER**

Osservando la figura 2 possiamo notare:

1. sotto la voce operazioni viene data la forma di ogni operazione: "+" agisce su due interi e fornisce risultato di tipo intero;
2. in un' implementazione, alla destra del simbolo " -->" sono possibili solo operazioni definite per il tipo della rappresentazione o per il tipo in questione. Qui siamo al livello del linguaggio assemblatore. Se non lo fossimo potrebbe apparire una specifica e una realizzazione per valori binari complementati a due.

La definizione dei tipi in un linguaggio di programmazione ha un aspetto positivo che consente al programmatore di utilizzarli senza doversi preoccupare di ulteriori definizioni e superare i vincoli imposti dall' architettura della macchina per escogitare una realizzazione. D' altro canto, risulta negativo se visto nell' ottica di una accettazione passiva del modo con cui il compilatore ha implementato la realizzazione che, quasi sempre, risulta essere un compromesso, a volte lontano, dal risultato ideale. I tipi dati fondamentali di un linguaggio sono di norma definiti nel preludio del linguaggio e sono considerati come parte integrante del codice del programma. Nel nostro contesto abbiamo definito il tipo INTEGER ma esso non è riconducibile al Pascal in quanto in tale linguaggio i tipi dato fondamentali non possono essere definiti dal programmatore. Conservando la notazione vediamo come potremmo definire il tipo dato fondamentale CHAR. Intanto possiamo dire che la cardinalità del suo dominio è 128. Infatti tante sono le combinazioni di sette bit che vanno da 0000000 a 1111111 compreso. Queste combinazioni rappresentano i caratteri ASCII definiti dalle norme americane. Tale dominio, per come è stato concepito, ammette l' ordinamento ed ogni elemento ha un solo precedessore escluso uno che chiameremo il minimo ed un unico successore escluso uno che chiameremo massimo. Sono applicabili operazioni di confronto (>, =, <), e, poiché ordinato, l' identificazione del precedessore e del successore. Anche INTEGER ha tale tipo di ordinamento ma che nella specifica già vista non abbiamo indicato per semplificare l 'esposizione. Prima di definire il tipo CHAR apriremo una parentesi sulla notazione. Nell' esempio del tipo INTEGER abbiamo utilizzato operatori che agiscono su due operandi (binari), inoltre siamo abituati a scrivere con notazione infissa, tuttavia non esiste ragione per cui non si possa scrivere in un altro modo.

Esempio:

SOMMA(X,Y)	invece di	**X+Y.**

Nella rappresentazione dei caratteri così come viene definita dalle norme ASCII la descrizione di ciascun carattere (insieme delle combinazioni di 7 bit) può essere visto e interpretato come intero. Pertanto nella descrizione della realizzazione utilizzeremo le stesse istruzioni macchina utilizzate per gli interi.

La prima notazione è tipica di linguaggi applicativi tipo LISP. Il punto fondamentale su cui soffermarci non è tanto la notazione infissa piuttosto notare come gli operatori del Pascal non siano altro che funzioni ad essi legate con specifiche notazioni. Quindi sotto la voce operazioni possiamo elencare non solo gli operatori specifici ma anche le funzioni e procedure (una procedura è una funzione che ritorna il risultato a se stessa). Da notare nella figura 3 come la procedura := ritorna un argomento e quindi viene sottolineata. Nel passare alla realizzazione occorre notare come nelle specifiche

americane ASCII i caratteri sono rappresentati dall' insieme delle combinazioni di 7 bit. Osserviamo quindi che il carattere più basso è legato un codice, come nella rappresentazione binaria di un intero, può essere interpretato come uno 0:

0000000

ed al valore più alto è assegnato un codice

1111111

che se interpretato come intero avrebbe il valore decimale 127.

Specifiche di tipo CHAR

Dominio
ASCII (l'insieme dei caratteri ASCII e il relativo ordinamento)

Operazioni

<. <=, =, >=, >	:-	(CHAR, CHAR): BOOLEAN
ORD	:-	(CHAR): INTEGER
:=	:-	(CHAR, CHAR)
LOW, SUCC, PRED, HIGH	:-	(CHAR): CHAR

Semantica
Basata sulla matematica degli insiemi ordinati

Fig. 3 - Specifiche per il tipo **CHAR**

Realizzazione del tipo **CHAR**

Rappresentazione
Binaria in complemento a due

Implementazione

```
succ(X)   --> load X
              iadd uno
              ...
basso(X)  --> 0, ecc.
```

Fig. 4 - Realizzazione del tipo **CHAR**

Tipi si, tipi no.

Molti linguaggi di programmazione non hanno la definizione di tipo e trattano i caratteri come gli interi e viceversa. Evidentemente da un punto di vista di utilizzo del computer tale diversificazione non è fondamentale da un punto di vista realizzativo mentre lo è da quello specifico poichè si opera su domini diversi tra loro. Pertanto la definizione dei tipi risulta essere una situazione non necessaria bensì ridondante. Il successo della definizione dei tipi è appunto la ridondanza che consente di scoprire **errori** e ricostruire lá dove si manifestano inconsistenze. Per esempio si possono individuare ed isolare usi errati di operandi e operatori. La precisione alla quale sono costretti gli utenti dei linguaggi "tipati" favorisce la realizzazione di prodotti (programmi) strutturati cioè più affidabili e meno faticosi da manutenere. Wirth giustifica tali affermazioni attraverso i seguenti argomenti:

- *la conoscenza dei valori permessi per le variabili è indispensabile per capire il funzionamento di un algoritmo; infatti in assenza di una esplicita indicazione è estremamente difficile stabilire il tipo di oggetti rappresentati da una variabile e individuare eventuali errori;*

- *l' adeguadezza e la correttezza di un programma dipendono dai valori iniziali degli argomenti; nella maggior parte dei casi sono garantite solo per valori compresi entro determinati intervalli, che ogni buona documentazione deve indicare esplicitamente;*

- *lo spazio di memoria necessario per rappresentare una variabile dipende dai valori che questa può assumere; è quindi necessario indicare esplicitamente il campo di tali valori, affinché il compilatore possa riservare lo spazio di memoria necessario;*

- *gli operatori che compaiono nelle espressioni sono definiti e operano correttamente solo se i valori dei loro argomenti appartengono a ben precisi intervalli; la definizione di tali intervalli permette al compilatore di stabilire se una combinazione di operatori e di operandi è lecita e costituisce quindi una ridondanza utile per eseguire un controllo del programma già in fase di compilazione;*

- *gli operatori sono realizzati da opportuni programmi che, in generale, dipendono dal campo dei valori permessi per gli argomenti: per esempio, nella maggior parte dei calcolatori la rappresentazione interna dei numeri interi è diversa da quella dei numeri reali e completamente diversa è nei due casi la successione delle azioni macchina necessarie per eseguire le operazioni aritmetiche.*

Strutture di dati

Abbiamo visto come in un linguaggio di programmazione la differenza esistente tra un dato e un tipo di dato sia notevole: il dato è il valore che una variabile può assumere durante l'elaborazione; il tipo di dato è un modello matematico composto da una collezione di dati sui quali sono ammesse certe operazioni. Molto spesso i dati da elaborare sono riuniti in agglomerati chiamati anche strutture dati. Le strutture dati possono essere viste come un tipo di dato caratterizzate più dall' organizzazione imposta agli elementi che la compongono che dal tipo degli stessi. Il modo di rappresentare i dati potrà influenzare la scelta degli algoritmi, la stesura e soprattutto l' efficienza del programma. Infatti il programmatore può utilizzare differenti strutture dati per raggiungere il medesimo scopo, a meno che la struttura dati stessa non sia già presente. I tipi strutturati come quelli semplici forniscono informazioni che vengono utilizzate dal compilatore per allocare memoria e rilevare errori. Quindi una struttura dati è vista come:

1) un modo sistematico di organizzare i dati;

2) un insieme di operatori che permettono di manipolare gli elementi della struttura o di aggregare gli elementi per ottenere altri agglomerati.

Fondamentalmente le strutture dati si distinguono in base alle caratteristiche presentate dalla disposizione dei dati e dal loro numero. Da questo punto di vista Bertossi le distingue in:

1) ***lineari***, in cui gli agglomerati sono forniti dai dati disposti in sequenza, tra quali si individua un primo, un secondo, ..., un ennesimo elemento:

- *a dimensione fissa*, in cui il numero massimo di elementi dell'agglomerato rimane sempre costante nel tempo;
- *a dimensione variabile*, in cui non esiste limite al numero massimo di elementi appartenente all' agglomerato.

2) ***non lineari***, in cui non non è individuata alcuna sequenza;

- *a dimensione fissa*, in cui il numero massimo di elementi dell'agglomerato rimane sempre costante nel tempo;
- *a dimensione variabile*, in cui non esiste limite al numero massimo di elementi appartenente all' agglomerato.

Occorre notare che le implementazioni delle strutture, indipendentemente dal fatto che siano lineari o non lineari, a dimensione fissa o a dimensione variabile, possono essere statiche, cioé definite durante la fase implementativa del programma, o dinamiche, cioé modificabili durante l'esecuzione del programma o come suol dirsi a run-time. Nella prima circostanza si possono utilizzare funzioni di accesso ai dati fornite dallo stesso linguaggio di programmazione e quindi certe operazioni sono implicite alla definizione della struttura (calcolo degli indirizzi, posizionamento, ecc.), nella seconda, la

definizione, la gestione e le operazioni relative alla struttura dati devono essere definite dall' utente stesso. In quest' ultimo caso il linguaggio di programmazione mette a disposizione costrutti che consentono di operare direttamente con l'indirizzo fisico della memoria di ciascun elemento. E' abbastanza intuitivo comprendere come la gestione di tali costrutti risulti macchinosa e delicata, mentre l' elaborazione complessiva più lenta.

Durante la descrizione delle strutture dati occore stabilire la notazione con la quale sono indicati i tipi di dato e le operazioni, il significato che ad essi si vuole associare, il modo in cui è memorizzata la struttura e sono eseguite le operazioni. Occorrono cioè specifiche sintattiche, semantiche e realizzative:

a) la specifica sintattica fornisce:

   1) l' elenco dei nomi dei tipi di dato utilizzati per definire la struttura, quello delle operazioni specifiche della struttura stessa e quello delle costanti;

   2) i domini di partenza e di arrivo, cioè i tipi di operandi e del risultato, per ogni nome di operatore.

b) la specifica semantica associa:

   1) un insieme ad ogni nome di tipo introdotto nella specifica sintattica;

   2) un valore ad ogni costante;

   3) una funzione ad ogni operatore espicitando le seguenti condizioni sui domini di partenza di arrivo:

   - la precondizione definisce quali restrizioni debbono essere soddisfatte prima che l 'operatore sia applicabile;
   - la postcondizione stabilisce come il risultato sia vincolato agli argomenti dell' operatore.

c) la realizzazione riporta la specifica ai tipo di dato e alle operazioni proprie del linguaggio di programmazione. Da notare che ad una stessa specifica possono corrispondere diverse realizzazioni, più o meno efficienti, a seconda dei dati e delle procedure utilizzati. Le attuazioni sono di norma nascoste agli utenti. Qualora si cambi la realizzazione l' utente, poco curioso, nemmeno se ne accorgerà.

Anche Collins definisce i tipi come una collezione di valori.

Una operazione è un processo che viene eseguito da una certa mansione. Per esempio, la differenza verrà interpretata così:

Eseguire:

5 - 3

produce la differenza tra due interi 5 e 3.

Il simbolo '-' è l' **operatore**; **5, 3** sono gli **operandi** mentre **2** è il **risultato**.

Un altro esempio:

Read (alunniquarta, alunno);

questa esecuzione **memorizza** la **componente corrente** del file **alunniquarta** nella **variabile alunno.**

Il comando Read è l' operatore mentre alunniquarta e alunno sono gli operandi. Così arriva a definire il tipo dato (**data type**) come composto da:

1 un tipo T;
2 una collezione di operatori in modo che per ciascun operatore c'è un operando o un risultato di tipo T.

Per esempio, il tipo di dato degli interi: il tipo è interi ed alcuni operatori sono:

+, -, *, **div**, **mod**, :=, = , and <

partendo dai tipi di dato predeterminati nel T-Pascal i tipi vengono classificati secondo lo schema seguente:

1 **Simple:** E' il tipo di dato il cui valore ha un senso anche se isolato, quindi non scomponibile cioè atomico. Questi tipi di dato non sono composti da più valori né riferiscono ad altri tipi. I tipi di dato semplici possono essere suddivisi in ordinali **(ordinal)**, dove il valore può essere ordinato dal primo all' ultimo, e tipo di dato reale **(real)**.

2 **Structured:** Sono tipi di dato costituiti da una collezione di valori di altri tipi di dato semplici o strutturati.

3 **Pointer:** E' il tipo dato il cui valore è un indirizzo di macchina di qualche variabile e ne definisce una gerarchia:

Tipi strutturati del Pascal

Nel seguito definiremo delle strutture dati e gli esempi che forniremo saranno implementazioni di funzioni Pascal per cui ci pare doveroso presentare i tipi di strutturati che il linguaggio di programmazione utilizzato come esempio ci mette a disposizione. Infatti i tipi strutturati, come del resto quelli semplici, forniscono una serie di informazioni che vengono utilizzate dal compilatore per allocare memoria e rilevare errori. Inoltre evidenziano come i dati descritti vengono utilizzati nel programma. Le variabili, dichiarate di tipo strutturato potranno essere manipolate sia a livello di struttura intera sia a livello di singolo campo. I tipi strutturati forniti dal Pascal sono i record, gli array, i file, i set e solo per la versione turbo del Pascal le string e gli oggetti. Per completezza e necessità implementativa presenteremo anche i tipi puntatori alla memoria heap.

Record

Il tipo di dato *record* è un insieme di uno o più elementi di tipo diverso correlati logicamente tra loro mediante un nome. Ciascun elemento è chiamato *campo* (field) e dichiarato come variabile. Attraverso il nome assegnato al record si identifca un oggetto logico del nostro modello che andiamo costruendo attraverso il nostro programma. Per Lings un record rappresenta un' entità nel sistema e che le entità hanno degli attributi (campi del record). Il Pascal consente la definizione dei record sia nella sezione *type* o in quella *var* :

```
type
    identificatore = record
            < dichiarazioni >

            end
```

Tutte le variabili che intendiamo definire quali record con le caratteristiche proprie del record appena definiti saranno:

```
var identificatore_var {, identificatore_var1} : identificatore_record ;
```

I record devono essere definiti definizione per definizione assegnando a ciascuno il proprio valore, poiché non è possibile né utilizzare funzioni esplicite di composizione né assegnare un valore all' intero record.

Esempio:

```
type aereo = record
        nome := char;
        posti :4..300;
        propulzione:(jet, elica, turboelica);
        equipaggio:3..20
        end

var executivel, dc8: aereo;

(* executivel := 8, jet, 3 *)

with executivel do
        begin
                nome := 'executivel';
                posti := 8;
                propulzione := jet;
                equipaggio := 3
        end
```

I dati di un record sono acceduti componendo il nome del record con il campo che si intende leggere o scrivere.

Es.

identificatore_record.identificatore_campo{.identificatore_campo}

temp := executivel.equipaggio

la variabile temp conterrà il valore 3.

Possiamo ora passare alla definizione di record quando questo è inteso come tipo di dato astratto prendendo esempio da Tanenbaum.
In Pascal come abbiamo visto la definizione di record è la seguente:

```
type or var
        tipoc1c2 = record
             c1 : tipocampo1;
             c2 : tipocampo2
         end;
```

Per quanto si riferisce alla defizione astratta del tipo record Tanenbaum la codifica come segue:

Specifiche per il tipo record

Dominio:

Il dominio dei tipi che compongono il record

Operazioni

```
leggicampo1     :- (tipoc1c2): tipocampo1
leggicampo2     :- (tipoc1c2): tipocampo2
scrivicampo1    :- (tipoc1c2, tipocampo1): tipoc1c2
scrivicampo2    :- (tipoc1c2, tipocampo2): tipoc1c2
```

Semantica:

tipo astratto tipoc1c2 = sequenza(2) di (tipocampo1, tipocampo2);

funzione astratta leggi1(c: tipoc1c2): tipocampo1;
postcondizione leggi1 = c1;

funzione astratta leggi2(c: tipoc1c2): tipocampo2;
postcondizione leggi2 = c2;

procedura astratta scrivi1(c: tipoc1c2; val1:tipocampo1);
postcondizione c1= val1

procedura astratta scrivi2(c: tipoc1c2; val2:tipocampo2);
postcondizione c2= val2

Realizzazione:

Utilizzando il tipo dato "record" del Pascal.

Fig.5 Realizzazione del tipo dato Astratto: -Record

Array

L' array è la struttura dati più nota e più diffusa. Infatti è presente in tutti i linguaggi di programmazione ad alto livello; in alcuni di essi come il FORTRAN è l' unica strutura dati predefinita. Un array è una collezione di elementi tutti dello stesso tipo e a ciascuno dei quali è possibile accedere direttamente. La definizione di un array si ottiene attraverso la specifica del numero degli elementi che lo comporranno e del tipo dei dati. Il tipo dei dati può essere uno qualunque dei tipi semplice (integer, boolean, char, real) oppure strutturati. In quest' ultimo caso si possono ottenere *array multidimensionali*. Intuitivamente un array si può vedere come una sequenza di celle contigue contenente ciascuna un dato del tipo dichiarato al momento della definizione di array. Per referenziare un dato basta indicare il numero della cella che contiene il dato cercato. L' indice di referenza alla celle può essere di qualsiasi tipo purché sia scalare. La dichiarazione di array deve essere contenuta nelle sezioni cost, type. var di un blocco.

```
identificatore : array [n..m] of tipodato;
```

Un elemento dell' insieme viene acceduto tramite l' identificatore dell' array seguito dal numero della cella che contiene il dato se questo è di tipo semplice altrimenti utilizzando la tecnica di indirizzamento del tipo preceduta dall' identificatore dell' array e ancora dal numero di cella che contiene il dato cercato.

Es.

```
numeri : array [1..10] of integer;

numeri[1] := 10;
```

la cella con indirizzo 1 del' array **numeri** contiene il valore 10.

Es.

```
temp:= numeri[1];
```

la variabile temp vale 10.

Es.

```
aeroplani : array[1..100] of aereo
.
.
aeroplani[5].nome := 'executivel;
aeroplani[5].posti := 8;
aeroplani[5].equipaggio := 3;
```

la parte 'propulsione' del record contenuto nella cella 5 dell' array 'aeroplano' risulterà senza significato.

Detto del funzionamento della struttura dati array, passiamo alla sua specifica quale tipo di dato astratto dal punto di vista di autori diversi:

Tanenbaum:
Premesso che utilizzerà una pseudo-funzione tipo() che consente di determinare il tipo di dato di una qualunque variabile a run time per poi assegnare lo stesso tipo dato ad un' altra variabile. Come è noto in Pascal tale possibilità è negata in quanto l' assegnazione dei tipi dato a tutte le variabili deve avvenire in fase di scrittura del programma stesso. In questo caso possiamo accettarlo poiché siamo in fase di specifica e non implementativa. La parametrizzazione del tipo di dato astratto tipovettore.

```
tipo astratto tipovettore(indmin, indmax, tipoelem) =
                    sequenza(ord(indmin) - ord(indmax) + 1) di tipoelem;

condizione tipo(indmin) = tipo(indmax);

funzione astratta leggere(tipovettore(indmin, indmax, tipoelem);
                    i: tipo(indmin)):tipoelem

precondizione indmin<= i <= indmax;
postcondizione leggere = nomevettore[ord(i) - ord(indmin)+1];

procedura astratta scrivere(nomevettore: tipovettore(indmin, indmax, tipoelem);
                    i: tipo(indmin; elem: tipoelem);   {scrive nomevettore[i] := elem}

precondizione indmin<= i <= indmax;
postcondizione nomevettore[ord(i) - ord(indmin)+1] = elem;
```

Bertossi descrive in questo modo la strutura dati vettore:

a) *Specifica sintattica*
Tipi: vettore, intero, tipoelem
Operatori:
CREAVETTORE: ()--> vettore
LEGGIVETTORE: (vettore, intero) --> tipoelem
SCRIVIVETTORE: (vettore, intero, tipoelem) --> vettore

b) *Specifica semantica*
Tipi:
intero: l'insieme dei numeri interi
vettore: l'insieme delle sequenze di n elementi di tipoelem
Operatori:
CREAVETTORE = v
Post: per ogni i, $1 \leq i \leq n$, l i-esimo elemento del vettore, v(i), è uguale ad un prefissato elemento di tipo tipoelem
LEGGIVETTORE(v, i) = e
Pre: $1 \leq i \leq n$
Post: e = v(i)
SCRIVIVETTORE(v, i, e) = v'
Pre: $1 \leq i \leq n$
Post: v'(i) = e, v'(j) = v(i) per ogni j tale che $1 \leq j \leq n$ e $j \neq i$.

c) *Realizzazione*
Quella del Pascal.

File

I programmi applicativi hanno spesso bisogno di memorizzare una grande mole di dati correlati. I file nascono per supportare le carenze che altri tipi di dato presentano. Per esempio, i record vengono definiti al momento che un programma inizia la sua elaborazione e memorizzati su supporti volatili (tipicamente RAM) infatti al termine della stessa se ne perde qualsiasi traccia. I tipi di dato file vengono invece memorizzati su memorie fisse che consentono il recupero delle informazioni anche dopo il termine dell' elaborazione, non solo dallo stesso programma ma anche da parte di altri. Le dimensioni della variabile di tipo file, non debbono essere dichiarate in anticipo, quindi di fatto i programmi dovranno considerare la possibilità di incorrere in problemi di spazio a causa dei limiti della memoria di massa. Gli elementi che compongono un file sono chamati *componenti*. I componenti potranno essere di qualunque tipo escluso il tipo **file**. Lo scopo fondamentle del tipo file è quello di trasferire i dati da o in una memoria di massa. Le memorie di massa sono generalmente unità a dischi ma per essere più precisi dovremmo parlare di periferiche esterne, cioè di dispositivi che comunicano con sistema e da esso sono riconosciuti attraverso l' interfacciamento. L' interfaccia non è altro che la composizione di hardware e software che permette il riconoscimento e l' utilizzo di dette periferiche. I dati ivi contenuti vengono acceduti sequenzialmente e le operazioni permesse sono di solito dette input ed output. Le operazioni di input sono quelle che trasferiscono i dati dal supporto periferico alla memoria centrale dove i dati possono essere letti manipolati e modificati; quelle di output sono quelle che trasferiscono il dato elaborato dalla memoria centrale alle periferiche per essere memorizzato, stampato; comunque reso disponibile. Per un maggiore dettaglio ed approfondimento riamandiamo a letture specifiche del Pascal. Fondamentalmente abbiamo una definizione del dato strutturato e le operazioni di ricerca del dato, lettura, modifica e scrittura.

Specifiche per il tipo FILE

Dominio:
tipo_file

Operazioni:

NEW:	-(): FILE
INSERISCI:	-(FILE,ELEMENTO):FILE
PRIMO, ULTIMO,SUCCESSIVO:	-(FILE):FILE
EOS:	-(FILE):BOOLEAN

Semantica:
Come quella degli insiemi finiti
Realizzazione
Quella del Pascal

Definizione:

```
var f: file of tipo;
```

Set

Un set è un agglomerato di elementii che appartengono a tipi dato ordinali escluso il tipo real ed il programmatore può manipolarlo come una unica unità dati. Gli elementi che appartengono ad un set variano da 0 (insieme vuoto) ad un numero massimo dipendente dalla macchina su cui andiamo ad elaborare -di norma nummax_elementi= 256-. Intuitivamente un insieme funziona al seguente modo: pensate ad un dopolavoro di una grossa industria dove affluiscono tutti i dipendenti fuori dall' orario di lavoro con i diversi hobbies. L'iscrizione al circolo è aperta a tutti però i dipendenti con l' interesse alla pesca saranno un sottoinsieme, così pure per gli amanti degli scacchi, del bridge, del tennis, del calcio e così via. Potranno poi esserci attività che non interessano nessuno ed allora queste attività p.e. cucito rappresenteranno un insieme vuoto. Dobbiamo prevedere inoltre che un iscritto al tennis puo' partecipare ad altre attività sociali come la podistica, la pesca, lo sci ecc. Se un iscritto non svolge alcuna attività risulterà solo nell' insieme degli iscritti. Da un punto di vista di utilizzo potremo verificare se un dato iscritto svolge una certa attività, ma non sarà mai possibile stampare l' elenco dei soci pescatori, solo stabilire se il signor Pinco è o no pescatore.

Espressione	Descrizione/Risultato
set1 * set2	Intersezione tra insiemi: il risultato contiene gli elementi comuni ai due insiemi
set1 + set2	Unione di insiemi: il risultato contiene tutti gli elementi di entrambi gli insiemi
set1 - set2	Differenza fra insiemi: il risultato contiene gli elementi di set1 che non appartengono a set2
set1 = set2	Uguaglianza fra insiemi: true se gli insiemi sono identici
set1 <> set2	Disuguaglianza tra insiemi: false se gli insiemi sono identici
set1 <= set2	true se set1 è sottoinsieme di set2
set1 => set2	true se set2 è sottoinsieme di set1
set1 **in** set2	true se set1 è sottoinsieme di set2 (come per <=); set1 può però essere un singolo elemento di tipo uguale al tipo base di set2

Tabella delle operazioni sui set

La forma generale della dichiarazione del tipo set è la seguente:

```
type
    nome_tipo = set of tipo_base
```

Es.

```
type
    cane = set of (lupo, terrier, mastino, levriero, pechinese);
    ottali = set of 0 .. 7;
    minuscole = set of 'a'..'z';
```

Le operazioni sui set vengono interpretate dal compilatore come tali se sono applicate a variabili di tipo set. E' infatti possibile assegnare ad altre variabili di tipo set dello stesso tipo base, attraverso l' operatore di assegnamento di set (:=) il valore di una variabile di tipo set o di un' espressione di tipo set o anche un set di elementi. Inoltre, è possibile assegnare a variabili booleane espressioni relazionali su set, che il programmatore può usare al posto di espressioni booleane.

L' implentazione della specifica di insieme sarà la seguente:

Specifica per il tipo **<N_Insieme>**

Dominio:

Insieme dei tipi: *tipoelem* , boolean, insieme

Operazioni:

+, *, -	:- (N_Insieme, N_Insieme): N_Insieme
<,<=, =, >=, >	:- (N_Insieme, N_Insieme): Boolean
IN	:- (tipoelem, N_Insieme): Boolean
[ ]	:- (tipoelem): N_Insieme

Semantica:

Tipi:

insieme:tutti gli insiemi con elementi di tipo *tipoelem*

boolean: [vero, falso]

Operatori:

Identica a quella definita per la teoria degli insiemi a cardinalità finita.

Realizzazione:

tipica del Pascal.

String

Questo tipo di dato strutturato è una evoluzione dei tipi definiti nel linguaggio Pascal, infatti è propria nella versione Turbo del linguaggio poiché nella versione iniziale di Wirth non era definita. La necessità di operare su un tipo dato strutturato come la string si è manifestata con l'elaborazione interattiva dei testi con i personal computer. Infatti la string è molto più flessibile del semplice tipo dato char e dello strutturato *array of char*. Fondamentalmente la string consente di lavorare con sequenze di caratteri a lunghezza variabile. Il tipo stringa viene definito come un array. La lunghezza rappresenta il numero massimo di caratteri che la stringa potrà contenere. Il riferimento alla variabile definita come tipo string avviene attraverso il suo nome o il suo nome con indice. Nel primo caso sarà disponibile l' intero contenuto, nel secondo l' elemento indicato dall' indice perché in questo caso il tipo di dato string viene trattato come array. Le stringhe consentono anche altre operazioni sui caratteri o composizioni di essi presenti nei dati di tipo strutturati string. Per un approfondimento rimandiamo ad un testo di Turbo Pascal poiché la string non verrà trattata in questo contesto.

Oggetti

Il turbo pascal nelle sue ultime versioni ha ammeso un tipo dati detto oggetti. Intuitivamente l' oggetto non è altro che una estensione del concetto di record. In effetti l' oggetto è una incapsulazione delle definizioni di un record e delle azioni che il record ammette per manipolare i dati in esso definiti. Alla staticità del **record** si associa la dinamicità delle operazioni, rappresentate da funzioni e procedure che elaborano il record (**metodi**). Questo tipo di dato consente al progettista di dare maggiore modularità al pogramma. I benefici che ne derivano si manifestano in fase di manutenzione e aggiornamento del programma. Un semplice esempio di un oggetto, lo prendiamo da Collins. Supponiamo di aver sviluppato un tipo dato detto "studente". Il tipo è composto da tre campi: il nome dello studente, l' indirizzo e il GPA. Le operazioni sono:

- leggere il nome dal file Studenti;
- scrivere il nome dello studente;
- riconoscer e se abbiamo letto studente dal file studente;
- vedere se il GPA è minore di alcuni valori predefiniti.

In Turbo Pascal scriveremo:

```
type
  AddressType = record
        Street        : string  [30];
        CityStateZip  : string  [20]
  end  { AddressType }
  StudentType = object
        Name          : string  [30];
        HomeAddress   : AddressType;
        GPA           : real;
        procedure ReadIn;
        procedure WriteOut;
        function Last: boolean;
        function GPALessThan (CutOff: real) : boolean;
  end;        { StudentType }
```

Il tipo dato studenti è stato dichiarato object con tre campi e quattro metodi. L' oggetto è similare al record come raggruppamento, ma mentre il record ha solo componenti passive -i campi- l' oggetto ha anche componenti attive: i metodi che operano esclusivamente sui campi dell' oggetto. Il completamento della dichiarazione prevede l' esplecitazione dei metodi. Per distinguere i metodi dagli altri sottoprogrammi è

richiesto un qualificatore. La dichiarazione di ciascun metodo si scosta da quelle delle altre funzioni e procedure per il fatto che il nome della funzione o precedura è composta dal nome dell' oggetto a cui si riferisce, un punto ed il nome della funzione quale sua estensione.
La dichiarazione dei singoli metodi per StudentType sono:

```
procedure StudentType.ReadIn;
begin
  ReadIn  ( StudentFile, Name, HomeAddress.Street,
            HomeAddress.CityStateZip, GPA )
end; { ReadIn }

procedure StudentType.WriteOut;
begin
  WriteIn    (Name);
  WriteIn    ( HomeAddress.Street );
  WriteIn    (HomeAddress.CityStateZip );
end;   { WriteOut }

function StudentType.Last: boolean;
begin
  Last  := Eof ( StudentFile )
end;   { Last }

function StudentType.GPALassThan ( CutOff: real ) : boolean;
begin
  GPALessThan := (GPA < CutOff)
end;      {GPALessThan}
```

Quando un oggetto è creato, la variabile risultante è detta **istanza** dell' oggetto es:

```
var
Student : StudentType;
```

la variabile Student è un' istanza di StudentType. Per riferire a qualsiasi campo o metodo dell' istanza basta far seguire il nome dell' istanza dall' identificatore del campo o del metodo interponendo tra loro un **punto**.
Quando si parla di object-oriented, un **message** è l' attivazione di un metodo per una determinata istanza. In questo caso il significato di message è l' isolamento dell' oggetto dal resto del programma. In altre parole messaggio è rendere **vivo** un oggetto **animandolo** dell' attività prevista dal metodo che con il messaggio andiamo ad attivare. E` doveroso notare come questa attività interna di un oggetto renda ancora più chiara l' idea della modularità: l' oggetto viene visto nel suo complesso e le sue attività interne lo rendono attivo indipendentemente dal mondo che lo circonda.

Units (cenni).

Una modularità completa non può prescindere da un isolamento fisico e concettuale. Il Turbo Pascal fornisce questa separazione fisica tra le parti di un programma consentendo compilazioni separate dei moduli e vegono chiamate **unit.** Fondamentalmente una unit è una compilazione separata di dichiarazioni Per un maggior approfondimento rimandiamo alla letteratura.

Puntatori (memoria heap)

Il puntatore è una struttura dati importante poiché consente di riferire alla RAM del computer in maniera diretta durante l' elaborazioen di un programma. Con questo meccanismo è possibile aggiungere memoria dati o dualmente rilasciarla.(espansione o rilascio) a seconda delle necessità contingenti, o come suol dirsi a run-time, sganciando il programma dai vincoli delle richieste iniziali. La RAM citata non è l' intero della memoria della macchina, ma quella parte che rimane libera (**heap**) dopo che sono stati allocati: il sistema operativo, i programmi residenti in memoria, il programma stesso, i dati ecc. I linguaggi di programmazione forniscono un indirizzo con cui il programmatore può riferire alla partizione assegnata. La memoria allocata con questo meccanismo rimane a disposizione fino a quando lo stesso programmatore non decide il suo rilascio.Il puntatore non è altro che un indirizzo di una locazione di memoria tale da contenere il tipo dato a cui il puntatore si riferisce (es. integer). Quindi il puntatore consente la definizione di blocchi di memoria uguali tra loro, e realizzare attraverso una loro connessione degli insiemi modulari formati dai blocchi definiti man mano che se ne presenta la necessità, chiamati anche nodi, che rendono la struttura modulare nella sua composizione e flessibile nel suo sviluppo. Il variare del valore del puntatore ci consentirà di inserire, estrarre riordinare i cammini in funzione della contrazione o dell' accrescimento della lista, in funzione comunque dell' andamento elaborativo del programma. Tutto questo è molto importante perché il programmatore amplia di fatto l' area di memoria definita dal compilatore, e la alloca ora a questa, ora a quella struttura secondo una gestione razionale ed oculata . Ovviamente l' impiego dei tipi puntatori non deve essere sostitutivo ma integrativo ai precedenti tipi dati strutturati. Genericamente la struttura puntutatore a memoria heap è definita dalla seguente istruzione:

```
type
    nome tipo = ^tipo-base;
```

I puntatori a parti di memoria heap trovano impiego nella realizzazione di strutture dati complesse come liste ed alberi. Le strutture a lista si compongono di record con un successore, ogni record è composto da due o più campi di norma uno dei quali contiene il puntatore all' elemento successivo così da creare una catena di elementi dello stesso tipo . Si possono realizzare applicazioni che prevedono anche la memorizzazione dell' indirizzo dell' elemento precedente (lista bidirezionale). Una struttura ad albero è simile a quella a lista ma ogni componente può avere più di un successore. Gli operatori che un programmatore ha a disposizione sono:

new	per creare nella heap un elemento del tipo richiesto (il suo valore sarà indefinito) e ritornare il relativo puntatore assegnandolo ad un identificatore;
dispose(*el*)	rimozione dal heap dell'elemento identificato da *el* .

Specifiche per heap

Dominio:
tutti i domini

Operazioni:

newheap	:- (): heap
NEW	:- (heap, type (T)): ^T
DISPOSE	:- (heap, type (T), ^T)
^	:- (heap, type (T), ^T): T
^ :=	:- (heap, type (T), ^T, T)

Semantica:

Tipi:	Descrizione dei tipi dato semplici o strutturati in gioco
Operatori:	
NEW	"creare" nel heap un elemento del tipo richiesto (il suo valore sarà indefinito) e ritornare il relativo identificatore;
DISPOSE(el)	rimozione dal heap dell 'elemento identificato da el el^ritorna il contenuto dell 'elemento puntato da el el^:= Tsi assegna all' elemento puntato da el il valore "T"
Realizzazione:	
	Utilizzando le primitive Pascal.

Es. 1:

Uso dei puntatori:
procedura new:

```
var     pun1, pun2 : ^tipoelem; {puntatori a tipoelem}
  x: tipoelem;
{ assumeremo che tipoelem = char }
```

Es. 2:

```
new(pun1); {pun1 indirizzo della heap configurata a tipoelem}
    pun1 := "a";
    pun2 := pun1;
    wirteln( pun1^, pun2^);  { alla stampante avremo "a" ed "a"}
```

Es. 3:

```
    x := "prova";
    pun2^ := x;
writeln( p^, q^); { alla stampante avremo "a" e "prova"}
```

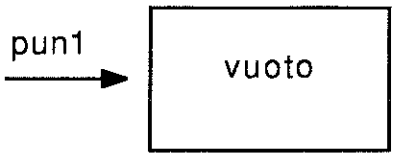
Es. 4:

```
    new(pun1);
    pun1^ := "riprova";
    writeln(pun1^, pun2^); {alla stampante avremo "riprova" e "prova"}
```

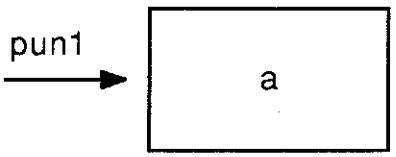
Non tratteremo volutamente i side effect prodotti da un uso improprio dei puntatori quali puntare alla solita heap (aliasing), utilizzare new in successione sullo stesso parametro, utilizzare dispose parametro e poi cercare di riutilizzarlo (dangling pointer), ecc. , poiché dipendono dalle implementazioni dei vari Pascal. Si lascia pertanto al buon senso dell' utilizzatore o della utilizzatrice.

Per concludere segue un esempio grafico sull' utilizzo della memoria heap:

new(pun1)

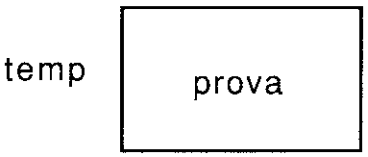


pun1^ := "a"

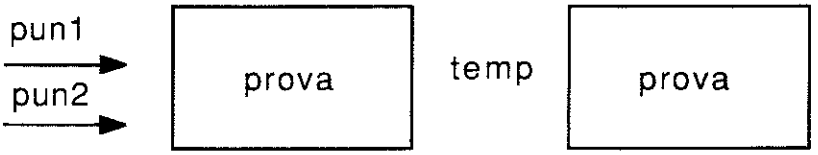


pun2 := pun1

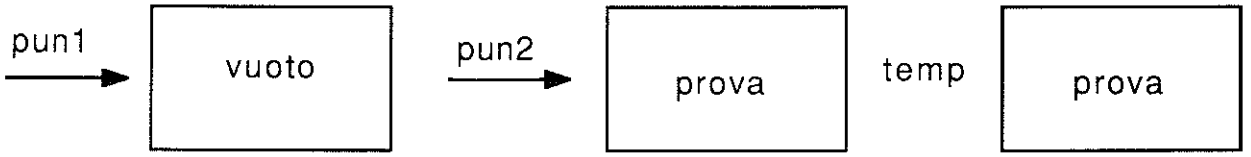
temp := prova



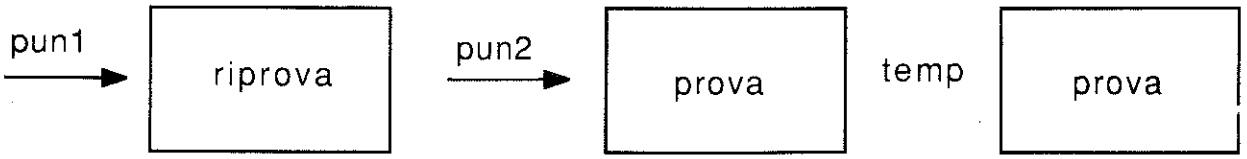
pun2 := temp



new(pun1)



pun1 := "riprova"



Liste

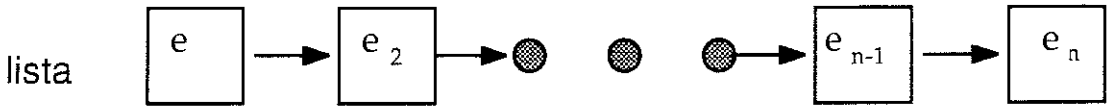
Le specifiche di Lista definiscono un insieme di elementi di un certo tipo totalmente ordinati. Con totalmente ordinati specifichiamo che esiste un solo elemento che non ha predecessore, ed un solo elemento che non ha successore. Essi si chiamano rispettivamente ***primo*** ed ***ultimo***. Nella fase realizzativa di una lista occorre tener conto di:

a) della grandezza dell' insieme lista che non è nota;
b) di una funzione di ordinamento intrinseca all' applicazione.

Es. grafico
Data la funzione di ordinamento:

$$e_k = O\ (e_j)$$

La lista risultante degli elementi e_j ordinati con la funzione ordinamento O sarà:



Poiché questa struttura dati è accessibile solo il primo elemento, ogni volta che intende raggiungere, inserire o cancellare qualsiasi elemento, la lista deve essere scorsa sequenzialmente fino a raggiungere l' elemento cercato. Nella letteratura, alcuni autori prevedono l' inserimento di un nuovo elemento solo in successione ad un altro; altri consentono anche un inserimento in precedessione. In definitiva il primo elemento consente "l' accesso" alla lista e la funzione di ordinamento che correla gli elementi permette la navigazione al suo interno. Passiamo ora alla descrizione del dato strutturato "lista" secondo Bertossi. Nel far ciò l' autore assume che ciascun elemento contenuto nella lista è caratterizzato da una posizione in L indicata con pos(i) e da un valore a_j. A prima vista 'i' si associa con la posizione nella lista ma questo non è sempre vero perché dipende dal modo in cui si è implementata il tipo di dato lista. Infine per comodità si assume l' esistenza di una posizione pos(n+1) che segue quella dell' ultimo elemento della lista. Dalla specifica semantica emerge che per accedere ad un elemento occorre conoscere la posizione. Ciò è possibile solo conoscendo a priori la posizione dell' elemento precedente (o successivo) ed applicando quindi l' operatore SUCCLISTA (o PREDLISTA). L 'unico operatore che dà per risultato una posizione senza che gliele venga fornita una in ingresso è l' operatore PRIMOLISTA. Per accedere ad un elemento occorre pertanto scandire la lista elemento per elemento a partire dal primo. L'inserzione o la cancellazione di elementi in una lista provocano un suo allungamento od un suo accorciamento e di fatto queste operazioni modificano la posizione di ciascun elemento che seguono quello cancellato o quello inserito.

a) *Specifica Sintattica*

Tipi: lista, posizione, booleano, tipoelem

Operatori:

CREALISTA:	() --> lista
LISTAVUOTA:	(lista) --> booleano
LEGGILISTA:	(posizione, lista) --> tipoelem
SCRIVILISTA:	(tipoelem, posizione, lista) --> lista
PRIMOLISTA:	(lista) --> posizione
FINELISTA:	(posizione, lista) --> booleano
SUCCLISTA:	(posizione, lista) --> posizione
PREDLISTA:	(posizione, lista) --> posizione
INSLISTA:	(tipoelem, posizione, lista) --> lista
CANCLISTA:	(posizione, lista) --> lista

b) *Specifica semantica*

Tipi: lista: insieme delle sequenze L= $a_1, a_2, ..., a_n$ di elementi di tipo tipoelem, dove l'i-esimo elemento ha valore a_i e posizione pos(i).

booleano: insieme dei valori di verità [vero, falso]

Operatori:

CREALISTA = L'

Post: L' = Ø, la sequenza vuota

LISTAVUOTA(L) = b

Post: b = vero, se L = Ø; b = falso, altrimenti

LEGGILISTA(p,L) = a

Pre: L= $a_1, a_2, ..., a_n$ e p = pos(i) per un i, $1 \le i \le n$

Post: a = a_i

SCRIVILISTA(a, p, L) = L'

Pre: L= $a_1, a_2, ..., a_n$ e p = pos(i) per un i, $1 \le i \le n$

Post: L' = $a_1, ..., a_{i-1}, a, a_i, a_{i+1}, ..., a_n$

PRIMOLISTA(L) = p

Post: p =pos(1)

FINELISTA(p, L) = b

Pre: L= $a_1, a_2, ..., a_n$ e p = pos(i) per un i, $1 \le i \le n$

Post: b = vero, se p = p(n+1); b = falso, altrimenti

SUCCLISTA(p, L) = q

Pre: L= $a_1, a_2, ..., a_n$ e p = pos(i) per un i, $1 \le i \le n$

Post: q =pos(i+1)

PREDLISTA(p, L) = q

Pre: L= $a_1, a_2, ..., a_n$ e p = pos(i) per un i, $1 \le i \le n$

Post: q =pos(i-1)

INSLISTA(a, p, L) = L'

Pre: L= $a_1, a_2, ..., a_n$ e p = pos(i) per un i, $1 \le i \le n$

Post: L' = $a_1, ..., a_{i-1}, a, a_{i+1}, ..., a_n$, se $1 \le i \le n$

L' = $a_1, ..., a_n, a$, se i = n+1

(e quindi L' = a, se i = 1 e L = Ø)

CANCLISTA(p, L) = L'

Pre: L= $a_1, a_2, ..., a_n$ e p = pos(i) per un i, $1 \le i \le n$

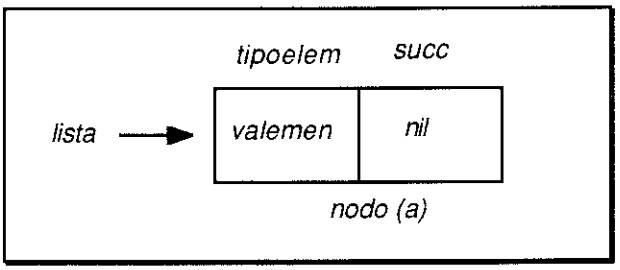
Post: L' = $a_1, ..., a_{i-1}, a_{i+1}, ..., a_n$

Nella presentazione del tipo dato strutturato *lista* abbiamo presentato un grafico di visualizzazione del tipo in studio senza soffermarci sulla metodologia implementativa. Il modo di implementare il tipo dato strutturato lista consente di valutare la complessità di una procedura che utilizza tale struttura di tipo di dato ed i suoi

operatori. Vedremo ora tre possibili realizzazioni della lista basate sull 'uso di puntatori, cursori e doppi puntatori.

Realizzazione con puntatori.

Per riconoscere l' ultimo elemento di una lista, il Pascal definisce un ***nil pointer*** che rappresenta un puntatore a niente: altro non è che la fine della lista. Lo stesso nil pointer è utilizzato nella creazione della lista (lista vuota o lista nil) e l' operazione di assegnamento sarà *lista := nil* . Non appena un elemento verrà aggiunto alla lista la variabile list assumerà il valore del puntatore al primo elemento. Detto *p* il puntatore al *nodo(p)* allora: *lista := p* . Mentre in attesa dell' inserimento del secondo elemento *succ(p) := nil.* Occorre tenere presente che nella realizzazione con puntatori alla memoria heap, il programmatore dovrà prevedere altri due tipi di operatori: creanodo e liberanodo. Il primo riserva nella memoria heap lo spazio necessario a contenere un elemento di tipoelem fornendo l' indirizzo dello stesso (puntatore), il secondo rilascia la memoria impegnata e definita secondo tipelem, e distrugge il puntatore a quella parte di memoria.



Es:

a)Creazione nuova lista:

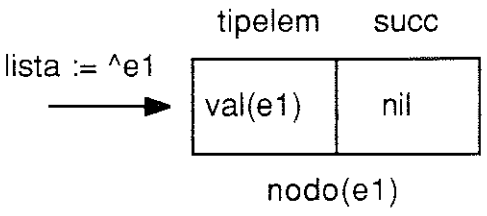
comando:

nuova(lista) lista := nil lista ⟶ ;

b) Inserimento primo elemento { valido anche per inserimento testa lista }

```
punt := getnodo  {p.e. ^e1}
nodo(e1)_tipoelem  :=  val(e1)
node(e1)_succ := lista
lista := punt
```

Postcondizione:



c) Inserimento dopo ennesimo elemento:

{Dopo aver trovato nodo inserimento seguendo funzione ordinamento}

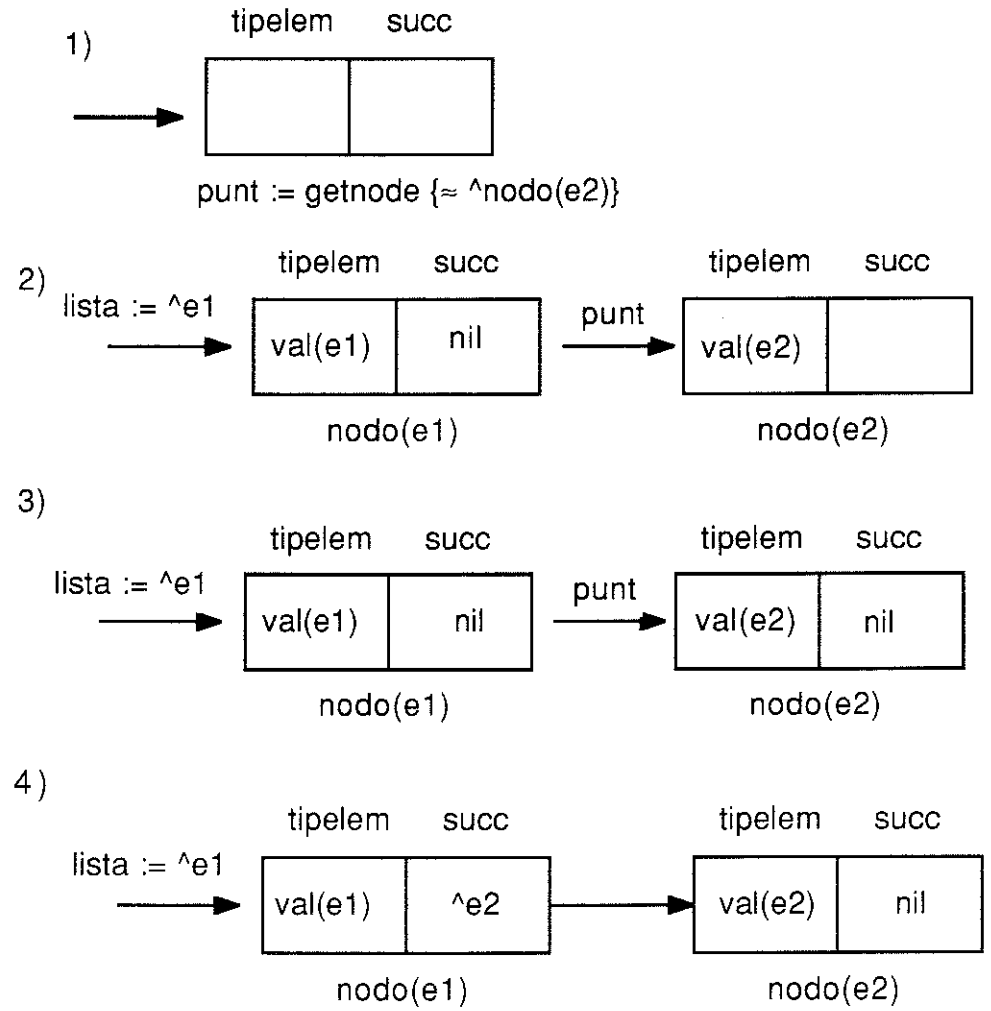
```
1 )     punt := getnodo; {p.e. ^e2}
2 )     node(e2)_succ := nodo(corrente)_succ;
3 )     nodo(e2)_tipoelem := val(e2);
4 )     nodo(corrente)_succ := punt;

        { Inserimento effettuato si continua eleborazione}
```

Visione grafica dell' operazione inserimento.

Precondizione:
{vedi esempio precedente di postcondizione}

Operazioni:



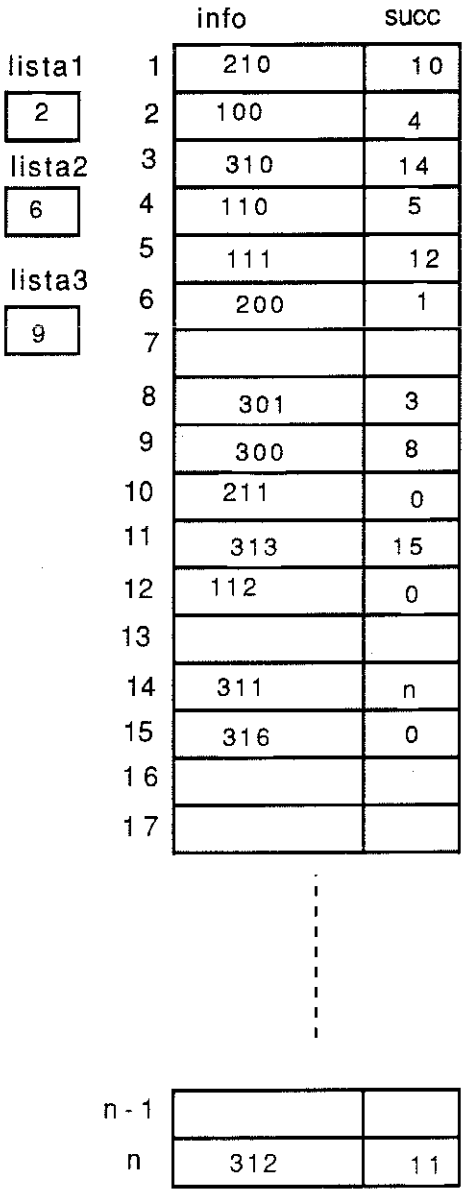
Per facilitare la ricerca in una lista e le operazioni relative all' inserimento di nuovi elementi o alla loro cancellazione spesso si realizzano liste con doppi puntatori, uno che punta all' elemento successivo ed uno che punta al precedente. Questa configurazione assegnata a ciascun elemento della lista consente di navigare nella lista in entrambe le direzioni. Da questo punto di vista predecessore e successore perdono il loro significato ordinale poiché la lista diventa simmetrica. Una realizzazione di questo tipo pur rimanendo flessibile dal punto di vista di ampliamento e restrizione a seconda

del numero degli elementi che sono coinvolti nell' elaborazione presenta anche il vantaggio di una complessità O(1).

Abbiamo visto la realizzazione e la gestione di una lista con i puntari e ci siamo resi conto della attenzione che occorre prestare per una buona ed efficace gestione. Se il Pascal così come altri linguaggi di programmazione consente l' utilizzo di un tipo dato così potente, altri linguaggi presentano questo tipo di dato quindi occorre trovare un metodo che consenta la creazione di tipi di dato strutturato come le liste. In questo caso Bertossi introduce il concetto di ***cursore*** cioè di simulatori di puntatori ad elementi di array (nodi). Lings come Tanenbaum parlano invece di realizzazione statica della lista. Al di là della sintassi con cui si affronta il problema la semantica è comune a tutti gli autori che affrontano il problema riservando allo scopo un' area di memoria predefinita (matrice) in fase di scrittura del programma composta per semplicità da due colonne e da enne righe, tante quante dovranno essere le informazioni che si intendono memorizzare. Le colonne saranno una di tipo integer, l' altra di tipo "tipoelem". Tutto questo si rende necessario poiché la lista è una sequenza di nodi ordinati tra loro che l' ordinamento dell' array non è suffficiente a garantire. Dobbiamo allora prevedere un sistema di aggancio tra un elemento della lista ed il successore attraverso dei puntatori creati ad hoc. Tornando all' array la colonna di tipo element sarà dedicata a memorizzare le informazioni (tipoelem), mentre l' altra conterrà il valore dell' indice dell' array che contiene l'elemento della lista che succede a quella corrente. La definizione in Pascal sarà:

```
const nodi = 100;
type puntnodo = 0..nodi;
     tiponodo =     record
                        info : integer;
                        succ : puntnodo
                    end;
var nodo: array[1..nodi] of tiponodo;
```

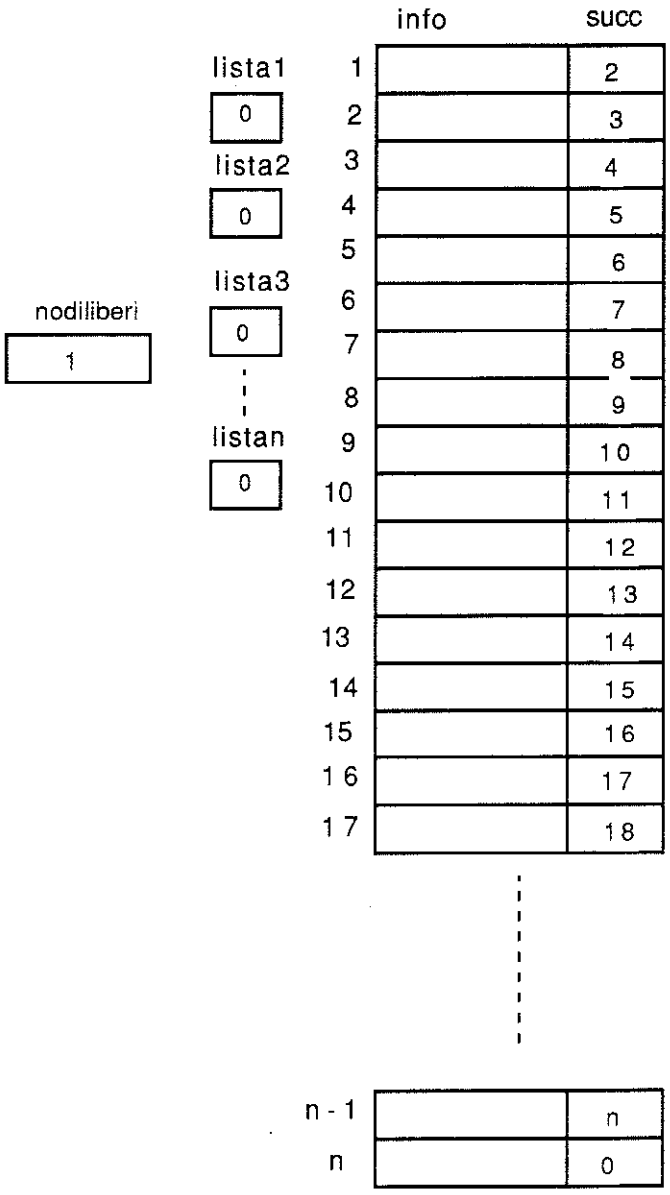
Con questa rappresentazione l'espressione *nodo[p]* è usata per referenziare l' elemento p dell' array, mentre *nodo[p].info* ci dà l' informazione contenuta nel nodo p, così come *nodo[p].succ* ci fornisce il puntatore all' elemento successivo a *p*. I puntatori potranno assumere un valore che varia tra 1 e nodi, mentre il valore zero è il puntatore *nil*. Inoltre si rende necessaria la definizione di tante variabili di tipo puntnodo che rappresentano i puntatori esterni alle liste. La definizione della variabile *nodiliberi* si rende necessaria per mantenere la lista dei nodi che non sono stati assegnati alle liste e che rappresentano gli elementi dell' array disponibili per un assegnamento a run time. La stessa lista sarà aggiornata quando su una lista sarà eseguito il comando dispose(p) ed il nodo sarà riattribuito alla lista libera pronto per essere nuovamente allocato. Al momento della definizione tutti gli elementi componenti il vettore saranno assegnati alla lista nodiliberi attraverso i seguenti comandi:



Visione di liste implementate staticamente.

```
procedure INIZIALIZZA;
var i: integer;
begin
        nodiliberi:=1;
        for i := nodi - 1 to nodi -1
                do nodo[i].succ := i +1;
        nodo[nodi].succ := 0;
end;
```

Graficamente avremo:



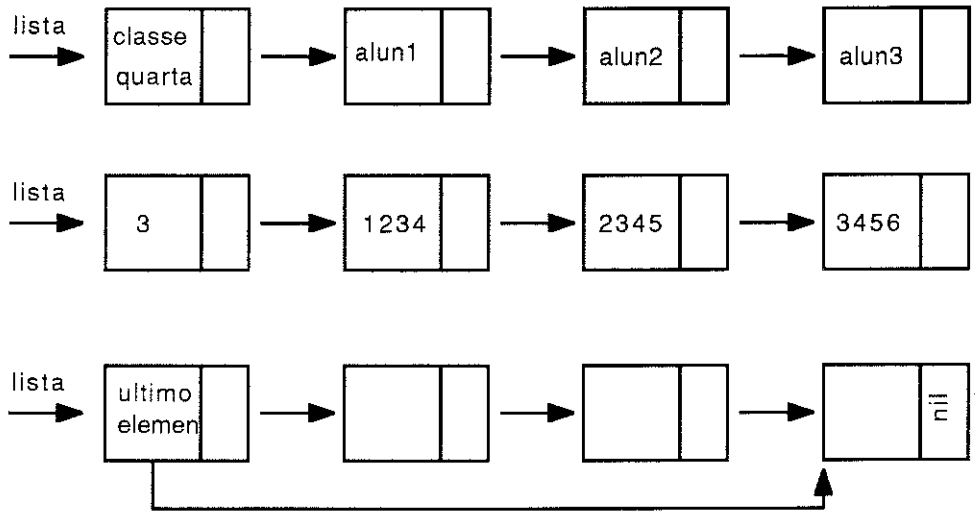
Alla richiesta di una lista la funzione prendinodo avrà due compiti, il primo di assegnare un nodo alla lista chiamante e il secondo di aggiornare la lista dei nodi liberi. Qualora tutti i nodi del vettore siano assegnati alle varie liste la crescita delle liste potrà avvenire solo quando ci saranno dei rilasci di nodi liberi e per la lista chiamante si avrà una segnalazione di spazio non disponibile (overflow).

```
function prendinodo: punnodo;
begin
     if nodiliberi = 0
          then errore("overflow")
          else    begin
                     prendinodo := nodiliberi;
                     nodiliberi := nodo[nodiliberi].succ
          end
end {function prendinodo};
```

L' azione duale rispetto a quella or ora vista sarà svolta dalla procedura 'liberanodo' che trasferirà un nodo già impegnato da una lista generica a quella di nodiliberi:

```
procedure liberanodo(p : punnodo);
begin
      nodo[p].succ := nodiliberi;
      nodiliberi := p
end {procedure liberanodo};
```

E` evidente che il comportamento della nodiliberi è quello di una coda di tipo LIFO (last in first out) cioè quella di una pila: un insieme di elementi il cui meccanismo elaborativo prevede una rimozione od un' aggiunta di elementi solo dalla cima; e come tale verrà gestita. La pila è un tipo di dato strutturato che analizzeremo tra breve.



Esempi di impiego di list header

Un' ultima annotazione sulle liste si rende necessaria per riferire sulla possibilità offerta dai nodi di testa (***header nodes o list header***). Questa implentazione consente al progettista di inserire in testa alla lista un nodo informativo sulla consistenza della lista, sul suo contenuto, ecc. Questo significa che il nodo di testa non è un elemento della lista ma un elemento complementare che di fatto arricchisce la descrizione della lista. Ovviamente le procedure e le funzioni che gestino le liste con la header debbono tener conto di questo item.

Pile

Nel parlare delle liste abbiamo avuto modo di accennare al dato strutturato pila vediamo di ampliare il discorso incominciando a sottolineare che in letteratura la pila viene identificata con Stack o Pushdown List. Visivamente la pila non è altro che una sequenza di elementi ai quali si può accedere uno alla volta e sempre a quello in testa. Anche l' inserimento degli leementi che allungano la pila avviene sempre dalla testa. E' intuibile che l' ultimo elemento inserito è quello che si propone quando intendiamo estrarre un elemento dallo stack. Il meccanismo di ultimo entrato primo uscente nella letteratura viene definito LIFO (last in first out). Per chi ha fatto il militare o conosce le armi da fuoco, la pila è paragonabile al caricatore di un' arma da fuoco appunto nel quale i proiettili vengono inseriti uno dopo l' altro dalla testa del caricatore ed il primo ad essere esploso sarà l' ultimo che è stato inserito. Altro esempio per tutti: chi utilizza pastiglie in tubetto ingerirà come prima pastiglia l' ultima inserita al momento del riempimento. Il caricatore di un' arma da fuoco sarà preso ad esempio perchè introdurrà con chiarezza le operazioni che si possono effettuare sulla pila di elementi. L' inserimento di un elemento nella pila è equivalente a quello di un proietile nel caricatore. Per compiere quest' ultima operazione occorre appoggiare il proiettile alla fessura di caricamento e spingerlo giù (push), mentre per l' estrazione si sfila il proiettile che si trova in testa al caricatore(pop), mentre una molla spingerà i proiettili verso l'alto proponendo il proiettile immediatamente sottostante per la successiva richiesta. In informatica vi sono parecchie applicazioni che utilizzano pile o per meglio dire code a gestione LIFO. La chiamata ad un sottoprogramma o ad una funzione necessita la memorizzazione dello stato del programma chiamante al momento della call perchè una volta eseguito il sottoprogramma o la funzione la macchiana dovrà riprendere il processo chiamante e continuare l' elaborazione. Se il programma chiamante o la funzione a sua volta chiama altri programmi o altre funzioni l' utilizzo della pila risulta palese. Le operazioni relative alla pila oltre alle classiche di Creapila e Vuoto, sono la PUSH, la POP, e la TOP. Come esempio riportiamo le definizioni di questo importante dato strutturato.

Specifica di Tanenbaum:

```
type stackitem = ...;              {type of item on stack}
abstrac type stack = sequence of stackitem;

abstract function empty ( s: stack): boolean;
postcondition empty = ( len(s) = 0 );

abstract function pop (s : stack): stackitem;
precondition not empty (s);
postcondition pop = last (s');
                    s = sub(s', 1, len (s') - 1);

abstract procedure push (s: stackitem; elt: stackitem);
postcondition s = s' + (elt)
```

Specifica BERTOSSI:

a) Specifica sintattica

Tipi: pila, booleano, tipoelem
Operatori:

CREAPILA:	() --> pila
PILAVUOTA:	(pila) --> booleano
LEGGIPILA:	(pila) --> tipoelem
INPILA:	(tipoelem, pila) --> pila

b) Specifica Semantica

Tipi:

pila:	insieme delle sequenze P= $a_1, a_2, ..., a_n$ di elementi di tipolelem
booleano:	insieme dei valori di verità {vero/falso}

Operatori:

CREAPILA = P'
post — P' = Ø, la sequenza vuota

PILAVUOTA(P) = b
post — b = vero, se P = Ø; b = falso, altrimenti

LEGGIPILA(P) = a
Pre — P = $a_1, a_2, ..., a_n$, e n ≥ 1
Post — a = a_1

FUORIPILA(P) = P'
Pre — P = $a_1, a_2, ..., a_n$, e n ≥ 1
Post — P' = $a_2, ..., a_n$, se n > 1, L = Ø se n= 1

INPILA(a, P) = P'
Pre — P = $a_1, a_2, ..., a_n$, e n ≥ 0
Post — P' = a, $a_1, a_2, ..., a_n$

c) Realizzazioni

poiché la pila è un caso particolare di lista, la realizzazione della lista funziona anche per la pila. Perciò gli operatori della pila possono essere simulati con quelli della lista.

CREAPILA = CREALISTA
PILAVUOTA(P) = LISTAVUOTA(P)
LEGGIPILA(P) = LEGGILISTA(PRIMOLISTA(P),P)
FUORIPILA(P) = CANCLISTA(PRIMOLISTA(P),P)
INPILA(a, P) = INSLISTA(PRIMOLISTA(P),P)

La pila come altri dati strutturati possono essere implementati utilizzando costrutti predefiniti del Pascal. Il migliore o peggiore rendimento di una implementazione dipenderà da cosa dovremo elaborare nel suo complesso. Le considerazioni su implementazioni statiche o dinamiche rimangono validi anche in questo caso. Vediamo intanto come Bertossi realizza il tipo dati strutturato Pila con un array (statico). Gli elementi da memorizzare nella Pila saranno inseriti a partire dalla primo elemento dell' array in poi. Poichè l' estrazione avverrà sull' ultimo elemento inserito le possibilità sono due o scandire tutto l' array fino a trovare l' ultimo elemento valido oppure aggioranre costantemente un puntatore alla testa della Pila in modo che possa fornirci l' indirizzo esatto dell' ultimo elemento inserito evidandoci qualunque ricerca.

Specifica di Lings:

Definizione di Tipo dato PILA

Dominio
- Insieme ELEMENTO
- Insieme BOOLEAN

Operazioni
- NEWSTACK: = () : STACK
- VUOTO : = (STACK) : BOOLEAN
- PUSH : = (STACK, ELEMENTO) : STACK
- POP : = (STACK) : STACK
- TOP : = (STACK) : ELEMENTO

Semantica

Per s: stack; e: elemento

vuoto(newstack)	= true
vuoto(push(s,j))	= false
pop(newstack)	= newstack
pop(push(s, j))	= s
top(newstack)	= indefinito
top(push(s, j)	= e

Realizzazione in Pascal:

```
type
        pila = record
            testa: 0..maxlung;
            elementi: array[1..maxlung] of tipoelem;
        end;

procedure CREAPILA(var P: pila);
    begin
        P.testa := 0
    end;

function PILAVUOTA( var P: pila): boolean;
    begin
        PILAVUOTA := (P.testa= 0)
    end;

function LEGGIPILA(var P: pila): tipoelem;
    begin
        if not PILAVUOTA(P) then LEGGIPILA := P.elementi[P.testa]
    end;

procedure FUORIPILA(var P: pila);
    begin
        if not PILAVUOTA(P) then P.testa :=P.testa - 1
    end;

procedure INPILA(a: tipoelem; var P: pila);
    begin
        if P.testa = maxlung then
        { messaggio di errore  di pila piena}
    else begin
        P.testa ;= P.testa + 1;
        P.elementi[P.testa] := a
    end
end;
```

Vediamo anche l' implementazione dinamica di uno stack suggerito da Lings.

```
(* Implementazione del tipo STACK *)

Procedure Newstack (var s: Stack);
(* Inizializza s come stack vuoto *)

    begin
        s := nil
    end;

Function Vuoto ( s: Stack): Boolean;
(* Ritorna true se lo stack è vuoto, altrimenti false *)

    begin
        Vuoto := (s = nil)
    end;

Procedure Push (var s: Stack; e: Elemento);
(* Sistema l' elemento e nella posizione top dello stack s *)
    var
        tem: Stack;                     (* per il nuovo componente top *)
    begin
        New(temp);
        with temp^ do
            begin
                data:= e;               (* inserisce l' elemento e *)
                pred := s               (* con predecessore top di s *)
            end;
        s := temp                       (* rende temp top di s *)
    end;

Procedure Pop (var s: Stack);
(* toglie da s l'elemento top. Eccezione: Underflow: s è vuoto *)
    var
        temp: Stack;
    begin
        if s <> nil
            then                        (* Esiste componente top: rimuovilo *)
                begin
                    temp := s;          (* temp fa riferimento al top di s *)
                    s := s^.pred;           (* toglie temp da s *)
                    Dispose (temp)      (* ritornalo ad Heap *)
                end
            else                        (* lo stack è vuoto *)
                Report (underflow)          (* eccezione *)
    end;

Function Top (s: Stack): Elemento;
(* Ritorna l' elemento top di s. Eccezione: Indefinito; s è vuoto *)
    begin
        if s <> nil
            then                        (* l'elemento esiste *)
                Top := s^.data          (* ne ritorna il valore *)
            else                        (* s è vuoto *)
                Report (indefinito)         (* eccezione *)
    end;
(* Fine implementazione di STACK dinamica*)
```

Code (Queue)

Una coda è una sequenza di elementi dello stesso tipo che si può ampliare aggiungendo elementi ma dualmente si può ridurre sottraendone altri. Il meccanismo di ampliamento e riduzione della grandezza di una coda deve seguire le seguenti regole:

a) l'inserimento del nuovo elementi è ammesso solo dopo l' ultimo elemento della coda

b) l' estrazione di un elemento avviene solo dalla testa della struttura dati coda.

Come si può ben vedere gli elementi estratti sono stati i primi ad essere inseriti; l' immissione dei nuovi rispettano la presenza dei vecchi. Questo meccanismo si traduce in First In First Out. Cioè il primo ad entrare è anche il primo ad uscire. La specifica sintattica della Coda espressa sarà:

a) Specifica sintattica
Tipi: coda. booleano, tipoelemen
Operatori:
CREACODA: () --> coda
CODAVUOTA: (coda) --> booleano
LEGGICODA: (coda) --> tipoelemen
FUORICODA: (coda) --> coda
INCODA: (tipoelemen, coda) --> coda

b) Specifica semantica
Tipi: coda insieme delle sequenze Q = a_1, a_2, ..., a_n, di elementi di tipo tipoelemen
booleano: insieme dei valori di verità { vero, falso}
Operatori
CREACODA< CODAVUOTA< LEGGICODA e FUORICODA hanno la stessa specifica dei rispettivi operatori della pila
INCODA(a, Q) = Q'
Pre Q = a_1, a_2, ..., a_n e n ≥ 0
Post Q' = a_1, a_2, ..., a_n, a

c) Realizzazioni
Ogni realizzazione per la lista va bene anche per la coda. Occorre stare attenti al fatto che se non conosce la posizione dell' ultimo elemento l' inserimento prevede la scansione dell' intera coda e quindi la complessità della procedura INCODA risulterà di O(n).

Realizzazione Statica

Defiamo un array di enne elementi di tipoelem grazie al costrutto Pascal:

```
ELEMENTI : Array [1.. enne] of tipoelem;
```

All' inizio dell' elaborazione il puntatore di inizio coda e di fine coda avranno lo stesso valore 0. Da questo momento in poi assumeremo che la coda è vuota quando:

```
Inizio - Fine = 0;
```

Questa assunzione è dovuta al fatto che il puntatore di Fine sarà aggiornato ogni volta che avviene un inserimento, mentre quello di Inizio ogni volta che si ha una

eliminazione o fuoriuscita dalla coda. Poiché queste operazioni portano ad un incremento costante degli indici di riferimento dell' array può accadere di raggiungere l' ennesimo ed ultimo elemento del nostro vettore. A questo punto l' elaborazione termina con un overflow di memori a disponibile. I rimedi a questo tipo di situazione sono due:

a) implementiamo una rutine di copia che trasla tutti gli elemneti della coda verso gli indirizzi più bassi del vettore. Questo tipo di operazione è molto costoso.

b) non apprena si ha overflow (ind > enne) verifichiamo che il primo ELEMENTI [1} = vuoto e assegnamo tale indirizzo a **fine**. Lo stesso accadrà quando **inizio** raggiungerà il valore enne + 1. In questo modo realizzeremo un vettore circolare che tornerà overflow quando si avranno al più enne inserimenti consecutivi senza nessuna eliminazione.

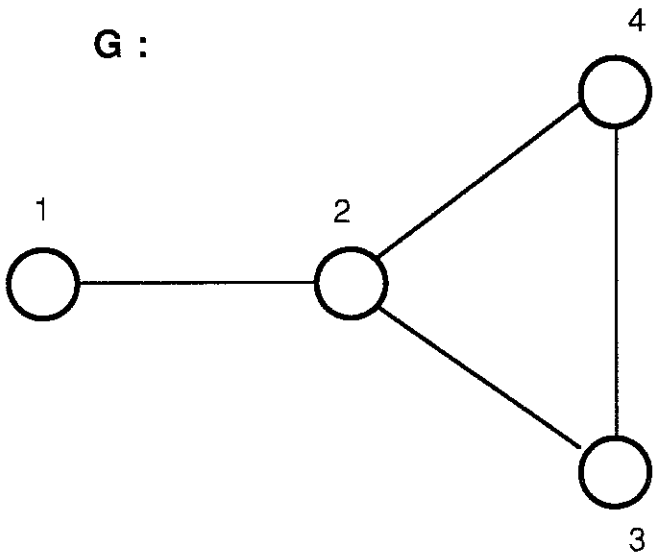
Abbiamo termiato la precedente esposizione affermando che l' overflow di memoria si manifesterà quando avremo inserito enne elementi senza averne eliminato alcuno. Occorre fare attenzione poiché cadiamo in un caso di ambiguità poichè inizio= fine =>coda piena e coda vuota. In questo caso occorrerà inserire un controllo al momento di un inserimento affinché segnali l' overflow un inserimento prima che esso accada. In questa maniera limitiamo l' utilizzo della memoria a enne - 1 elementi.

Grafi

Un grafo è un insieme di nodi, A, ed un insieme di archi, b. Un arco è una linea che collega due nodi qualsiasi:

G= { 1, 2, 3, 4, (1, 2), (2, 3), (2, 4), (3, 4)}

che può essere visto come:

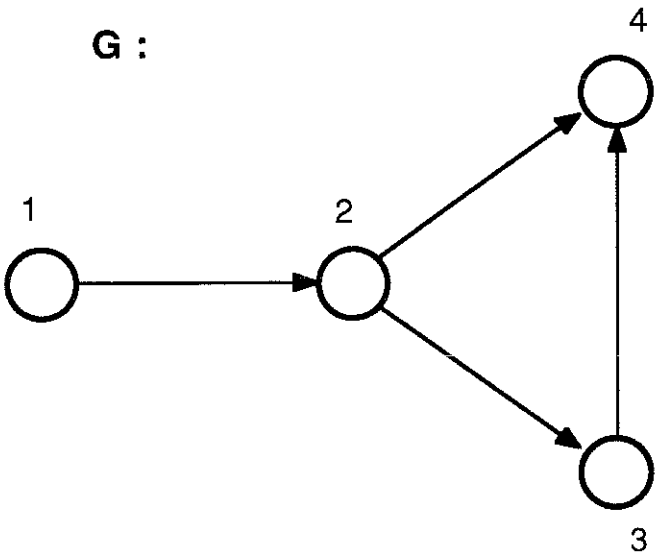


Se due nodi sono connessi da un arco, diremo che sono adiacenti. Se partiamo da un nodo n_1 e ci muoviamo attraverso i nodi adiacenti fino araggiungere n_2, gli archi corrispondenti formano un cammino nel grafo che parte da n_1 ed arriva in n_2. Per esempio se partiamo da dal nodo 1 di G ed arriviamo al nodo 4 passando dal nodo 2, possiamo vedere che gli archi (1, 2) e (2,4) formano un cammino dal nodo 1 al nodo 4. I nodi attraversati da un cammino devono essere distinti, sempre che lo stesso nodo non designi sia l' inizio che la fine di un cammino. Un cammino che ha lo stesso nodo come punto d' inizio e di fine è chiamato ciclo. Un esempio di ciclo è:

(2, 3), (3, 4), (2, 4) [Nota: (2, 4) ≈ (4, 2)]

che descrive il cammino che parte e finisce nel nodo 2.

Definiamo grafo connesso un insieme di n_n nodi e di n_a archi tali che data una coppia di nodi distinti del grafo esiste un cammino che li collega. Un grafo è orientato se gli archi hanno un verso di percorrenza (sono orientati): i nodi sono ordinati. I grafi orientati sono spesso rappresentati con delle frecce che indicano la direzione dei nodi. Se G fosse ordinato risulterebbe:



Il tipo dato grafo è rappresentabile attraverso una matrice con i nodi sulle colonne e gli archi sulle righe:

	n_1	n_2	n_3	n_4
a_1	1	1	0	0
a_2	0	1	1	0
a_3	0	1	0	1
a_4	0	0	1	1

Tabella o matrice di adiacenza

Alberi

La topologia definisce un grafo *albero libero* Å purché:

a) sia connesso;

b) non contenga cicli.

Un albero libero Å con *n* nodi presenta le seguenti proprietà:

1) Å contiene n -1 archi;

2) esiste solo un percorso semplice tra ogni coppia di nodi;

3) con la rimozione di un arco qualunque di Å si ottiene una struttura sconnessa di due alberi liberi;

4) la definizione di albero libero può essere riformulata sostituendo la condizione 1) a b) o ad a) indifferentemente.

In altri termini un albero libero si può definirlo come una coppia T = (N, A), dove N è un insieme finito di nodi ed A è un insieme di coppie non ordinate di nodi detti archi, tali che:

a) il numero di archi è uguale al numero di nodi meno uno;

b) T è connesso, ovvero per ogni coppia di nodi u e v in N esiste una sequenza di nodi distinti $u_0, u_1, ..., u_j$ tali che $u = u_0$, $v = u_j$, e la coppia $[u_i, u_{i+1}]$ è un arco di A, per i = 0, 1, ..., j-1.

Se ad un albero libero T si designa un nodo r arbitrario quale radice e ne organizziamo i nodi a livelli avremo un albero radicato.

Qualora si dia un ordinamento ad una struttura ad albero radicato si vengono a definire:

- la radice; *Un albero con radice è un grafo connesso e orientato nel quale ogni nodo ha un predecessore eccetto un solo nodo chiamato radice che non ha precedessori ma solo successori.*
- il grado del nodo; *è il numero di successori di quel nodo.*
- foglie; *Un nodo di un abero con zero successori è chiamato foglia di Å.*
- rami; *Un nodo di un abero con almeno un successore è chiamato ramo di Å*
- il grado dell' albero; *Il grado di un albero è dato dal massimo grado di tutti i nodi di Å*
- predecessore; *Se limitiamo ad uno il grado di un albero allora avremo che ogni nodo n_n ha un predecessore eccetto uno che chiameremo radice;*
- successore; *Se limitiamo ad uno il grado di un albero allora avremo che ogni nodo n_n ha un successore eccetto uno che chiameremo foglia;*
- livello di un nodo; *Se L è il numero degli archi che compongono il cammino che unisce la radice di un albero Å ad un*

nodo n_n dello stesso albero, il livello di n_n in Å è definito come L+1. Al nodo radice è assegnato il livello L = 0 .

- altezza; *L' altezza di un albero è data dal massimo livello di tutti i nodi che lo compongono.*
- foresta; *Una foresta è un insieme di alberi con radici separati tra loro.*
- padre; *Se n_1 è il prerdecessore di n_2, allora n_1 è chiamato padre di n_2.*
- figlio; *Se n_2 è il successore di n_1 allora n_2 è chiamato figlio di n_1.*
- fratello; *Se n_1 e n_2 hanno lo stesso padre allora n_1 e n_2 sono chiamati fratelli.*
- antenato; *Se n_1 si trova sull' arco che unisce la radice a n_2, n_1 è chiamato antenato di n_2.*
- discendente; *Se n_1 si trova sull' arco che unisce la radice a n_2, n_2 è detto discendente di n_1.*
- sottoalbero; *Un albero (con radici) Å può essere nullo o un insieme di nodi con un nodo distimguibile , r, chiamato radice. I rimanenti nodi di Å sono suddivisi in s sottoinsiemi separati , chiamati sottoalberi di Å , ciascuno dei quali è un albero (con radici). Il valore s è il grado di r. Questa definizione è ricorsiva e definisce un albero n termini di sottoalberi a loro volta considerati alberi. Per questa ragione si fa frequentemente riferimento alla struttura ad albero come ad una struttura ricorsiva. La definizione sottolinea i nodi di un albero come ad una struttura ricorsiva.*
- albero ordinato. *Un albero ordinato O è un albero radicato stabilendo un ordinamento tra i nodi che stanno alllo stesso livello, distinguendo così il primo figlio di un nodo , il suo secondo figlio, e così via.*
- albero binario *Un albero binario è un particolare albero ordinato in cui ogni nodo ha al più due figli e si fa distinzione tra* figlio sinistro *e* figlio destro.

Riassumendo gli alberi liberi rappresentano l' insieme matematico delle coppie Å = (N, A) dove N rappresenta l' insieme finito dei nodi ed A l' insieme delle coppie non ordinate di nodi chiamati archi. Gli alberi radicati $Å_r$ sono il sottoinsieme ordinato gerarchicamente (padri, figli) e per livelli dell' insieme Å. Gli alberi ordinati $Å_o$ sono il sottoinsieme degli $Å_r$ che ha subito un successivo ordinamento dei nodi che stanno al medesimo livello (primo, secondo, ..., ennesimo figlio). Gli alberi binari $Å_b$ sono particolari alberi ordinati con al più due figli distinti in figlio destro e figlio sinistro. Le operazioni definite sugli alberi sono, oltre la creazione dell' albero, l' inserimento o la cancellazione dei nodi, siano essi foglie o intermedi. In quest' ultimo caso parleremo

di rimozione o inserimento di sottoalberi. Un altro insieme di operazioni che non modificano la struttura del tipo dato è rappresentato dalla possibilità di spostarsi da un nodo ad un suo parente e attraverso visite guidate per recuperare i singoli dati. Diversamente da come si comporta in natura la rappresentazione degli alberi da parte degli informatici è rovesciata in quanto la radice è posta in alto (top) mentre le foglie sono poste in basso (bottom). La direzione dalla radice alle foglie è giù (down), mentre dalle foglie alla radice è detta su (up). Così come viaggiare dalla radice alle foglie è detta discesa il movimento inverso è detta arrampicata. Noi ci soffermeremo sugli alberi ordinati dando maggior enfasi a quelli binari.Nella tabella che segue è riportata la descrizionedegli alberi secondo Bertossi.

Struttura dati "albero ordinato"

a) Specifica sintattica

Tipi : albero, booleano, nodo.

Operatori:

Crea Albero:	() -> albero
AlberoVuoto:	(albero) -> booleano
InsRadice:	(nodo, albero) -> albero
Radice:	(albero) -> nodo
Padre:	(nodo, albero) -> nodo
Foglia:	(nodo, albero) -> booleano
PrimoFiglio:	(nodo, albero) -> nodo
UltimoFratello:	(nodo, albero) -> booleano
SuccFratello:	(nodo, albero) -> nodo
InsPrimoSottoAlbero:	(nodo, albero, albero) -> albero
InsSottoAlbero:	(nodo, albero, albero) -> albero
CancSottoAlbero:	(nodo, albero) -> albero

b) Specifica Semantica

Tipi: albero: Insieme degli alberi ordinati $Å_O = (N, A)$, come definiti precedentemente, in cui ad ogni nodo u in N è associato l' intero livello(u).

booleano: insieme dei valori di verità (vero, falso).

Operatori:

CreaAlbero = $Å_O'$

Post: $Å_O' = Ø$, l'albero vuoto

AlberoVuoto($Å_O$) = b

Post: b = vero, se $Å_O = Ø$; b = falso latrimenti

InsRadice(u, $Å_O$) = $Å_O'$

Pre: $Å_O = Ø$

Post: $Å_O' = (N, A)$ con $N = \{u\}$, $A = Ø$, e livello(u) = 0

Radice($Å_O$) = u

Pre: $Å_O \neq Ø$

Post: u è la radice di $Å_O$, cioè livello(u) = 0

Padre(u,$Å_O$) = v

Pre: $Å_O \neq Ø$, u è in N, e livello(u) > 0

Post: v è il padre di u in $Å_O$, cioè [v,u] è in A e livello(u) = livello(v)+1

Foglia(u, $Å_O$) = b

Pre: $Å_O \neq Ø$, u elemento di N

Post: b = vero, se non esiste alcun v in N tale che [u,v] è in A e livello(v) = livello(u)+1; b = falso altrimenti

PrimoFiglio(u,T) = v
Pre: $Å_O \neq Ø$, u è in N, Foglia(u,$Å_O$) = falso
Post: [u,v] è in A, livello(v)=livello(u)+1, e v è il primo figlio di u secondo una realzione di precedenza stabilita tra i figli di u

UltimoFratello(u, $Å_O$) = b
Pre: $Å_O \neq Ø$, u è in N
Post: b = vero, se non vi sono altri fratelli di u che lo seguono nella relazione di precedenza; b = falso altrimenti

SuccFratello(u, $Å_O$) = b
Pre: $Å_O \neq Ø$, u è in N, UltimoFratello(u, $Å_O$) = falso
Post: v è il fratello di u che segue immediatamente u nella relazione di precedenza

InsPrimoSottoAlbero(u, $Å_O$, $Å_O'$) = $Å_O''$
Pre: $Å_O \neq Ø$. $Å_O' \neq Ø$, u è in N
Post: $Å_O''$ è ottenuto da $Å_O$ aggiungendovi l' albero $Å_O'$, la cui radice r' diventa il nuovo primo figlio del nodo u

InSottoAlbero(u, $Å_O$, $Å_O'$) = $Å_O''$
Pre: $Å_O \neq Ø$. $Å_O' \neq Ø$, u è in N, u non è la radice di $Å_O$
Post: $Å_O''$ è ottenuto da $Å_O$ aggiungendovi l' albero $Å_O'$, la cui radice r' diventa il nuovo fratello che segue immediatamente u nella relazione di precedenza

CancSottoAlbero(u, $Å_O$) = $Å_O'$
Pre: $Å_O \neq Ø$, u è in N
Post: $Å_O'$ è ottenuto da $Å_O$ togliendovi il sottoalbero di radice u, cioè u stesso e tutti i discendenti di u

Accesso ai dati

L' accesso ai dati in una struttura ad albero si ottiene con una operazione detta di "traversata" o di "navigazione". Questa operazione consente di visitare una sola volta tutti i nodi dell' albero seguenda una determinata rotta di viaggio. Evidentemente la "visita" può essere eseguita in tanti modi detti "ordini"; i principali sono: ordine anticipato (previsita), differito (postvisita) e simmetrico (invisita).
Sia $Å_O$ un albero non vuoto di radice r. Se r non è foglia ma ha k > 0 figli indichiamo con $Å_{O1}$, , $Å_{Ok}$ i K sottoalberi di $Å_O$ aventi come radice i k figli di r. I predetti ordini vengono definiti ricorsivamente come segue:

1) La previsita di $Å_O$ consiste nell' esaminare r poi nell' effettuare, nell' ordine, la previsita di $Å_{O1}$, , $Å_{Ok}$;
2) La postvisita di $Å_O$ consiste nell' effettuare, nell' ordine, la postvisita di $Å_{O1}$, , $Å_{Ok}$ e poi nell' esaminare r;
3) La invisita di $Å_O$ consiste nell' effettuare, nell' ordine, la invisita di $Å_{O1}$, , $Å_{Oi}$, nell' esaminare r, e poi nell' effettuare, nell' ordine. la invisita di $Å_{Oi+1}$, , $Å_{Ok}$, per un prefissato i ≥ 1.

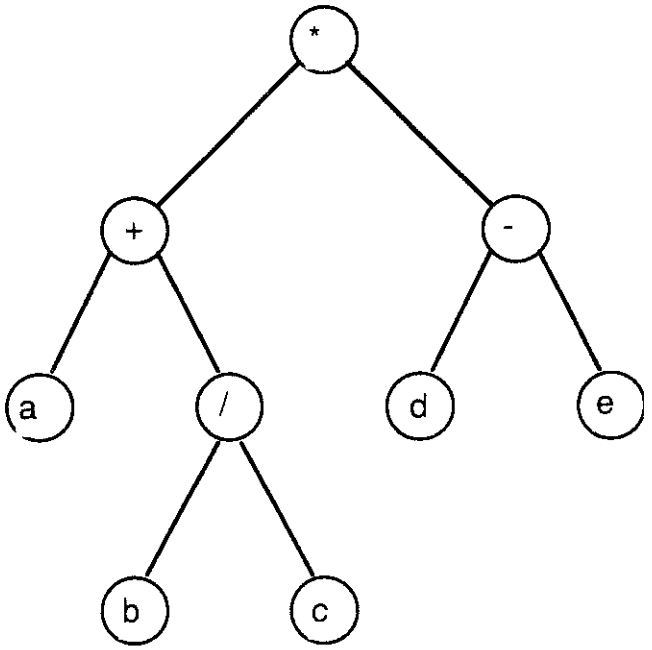


Fig.Albero ordinato

Prendendo come esempio l' albero rappresentato nella figura precedente, i tre ordini di visita daranno i seguenti risultati:

Previsita :

*** + a / b c - d e**

Postvisita :

a b c / + d e - *

Invisita :

a + b / c * d - e

Realizzazioni

L' utilizzo degli alberi in informatica è molto diffuso poichè questo tipo di struttura dati consente archiviazioni e ricerche dati funzionali e sufficientemente rapide. I tipi di implementazione sono pertanto diversi e comunque orientate verso algoritmi specifici. La scelta della rappresentazione migliore, come sempre, dipende dalle operazioni necessarie a risolvere il problema. La facilità nell'eseguire le operazioni dipende sia dall'elaboratore che dal linguaggio di programmazione disponibile, come pure dalla rappresentazione scelta. Più spesso una rappresentazione a doppia concatenazione sembra migliore particolarmente per gli alberi binari, ma sono state suggerite rappresentazioni con quattro *link* (per esempio padre, fratello di destra, fratello di sinistra primo figlio). un albero a molti link rende facile la navigazione in ogni direzione, e, in generale, l' inserzione, e la cancellazione sono facilitati. Comunque sia, indipendentemente dalla memoria in più richiesta per i link , ci può

essere una considerevole quantità di elaborazione in più per trattare link piuttosto improduttivi. Vedremo alcune realizzazioni suggerite.

Realizzazione con vettore dei padri:

Supponiamo che i nodi dell' albero $Å_{01}$ siano numerati da 1 a n. La più semplice realizzazione fa uso di un vettore contenente, per ogni nodo i, $1 \le i \le n$, il cursore padre: $Å_{01}[i] = j$, se j è il padre di i; $Å_{01}[i] = 0$ se i è la radice. Con questa realizzazione è facile visitare i nodi procedendo dalle foglie alla radice, cioè dal basso verso l' alto. Viceversa è costoso passare da un nodo ai suoi figli, individuare il livello del nodo, e inserire e cancellare sottoalberi. Infine non è chiara la relazione di precedenza tra fratelli, a meno che se ne assuma una implicitamente, per esempio imponendo che un figlio abbia sempre un numero maggiore del padre e che i nodi fratelli siano numerati in modo crescente da sinistra verso destra. Assumendo la suddetta relazione di precedenza tra fratelli, qui sotto è riportata come esempio soltanto la procedura SuccFratello:

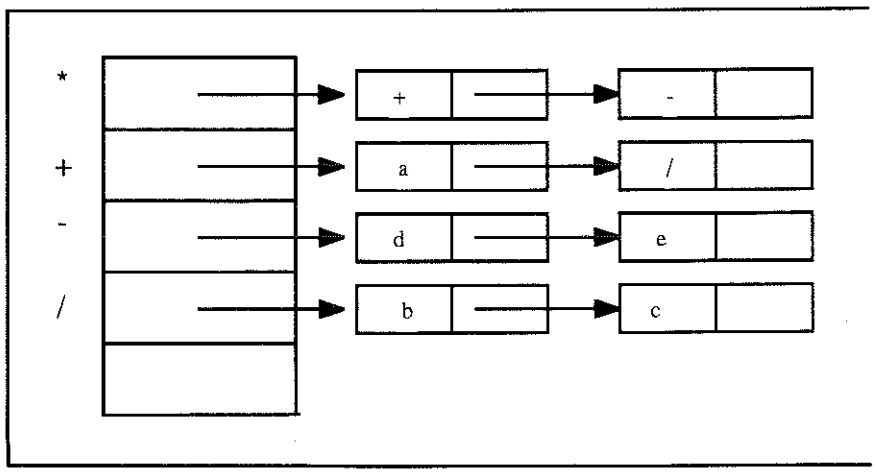
```
type
    nodo = integer;
    albero = array [ 1 .. maxlung] of nodo;

funcion SuccFratello (u: nodo; var Å0: albero) :nodo;
    var i,j: nodo; trovato: boolean;
    begin
        j := Å0[u]; i := u + 1; trovato := false;
        while (not(trovato)) and (i ≤ maxling) do
            if Å0[i] = j then
                    begin SuccFratello := i; trovato := true end;
                    else i:= i + 1
    end;
```

La complessità della procedura SuccFratello è O(n).

Realizzazione con la lista dei figli

Una realizzazione molto usata consiste nel mantenere, per ciascun nodo dell' albero, una lista di tutti i suoi figli. Poiché il numero di figli può variare nel tempo, di solito la lista è realizzata con puntatori o cursori.



Realizzazione con liste dei figli dell' albero figura precedente.

Trascurando la realizzazione del tipo di dato lista, si hanno le seguenti dichiarazioni:

```
type
    nodo = integer;
    albero = record
        testa: array[1..maxling] of lista;
        radice: nodo
    end;
```

gli operatori degli alberi possono essere realizzati in termini degli operatori delle liste. ad esempio, qui sotto è riportata le realizzazione dell' operatore PrimoFiglio:

```
function PrimoFiglio(u: nodo; var Å0: albero): nodo;
    var L: lista;
    begin
        L := Å0.testa[u];
        PrimoFiglio := LeggiLista(PrimoLista(L); L)
    end;
```

Si imponga adesso una realizzazione specifica per le liste, in cui "lista" e "posizione" sono interi, usati come cursori in un array di record, e $Å_0$.testa[u] punta direttamente alla cella contenente il primo elemento della lista:

```
var celle: array[1..maxlung] of record
    elemento: integer;
    sucecssivo: integer
end;
```

Gli operatori PrimoFiglio e Padre sono realizzati come segue:

```
function PrimoFiglio(u: nodo; var Å0: albero): nodo;
    var L: lista;
    begin
        L := Å0.testa[u];
        PrimoFiglio := celle[L].elemento
    end;

function Padre(u: nodo; var Å0: albero): nodo;
    var p: nodo; i: posizione; trovato: boolean;
    begin
        p:= 1; trovato := false;
        while (not(trovato)) and (p≤ maxlung) do begin
            i := Å0.testa[p];
            while (not(trovato)) and (i ≠ 0) do
```

```
                if celle[i].elemento = u
                    then begin Padre := p; trovato := true end
                else i:= celle[i].successivo;
            p := p + 1;
        end
    end;
```

La complessità di quest' ultima funzione è O(n).
L' ultimo esempio proposto da Bertossi e cioè la realizzazione con liste dei figli non si presta bene ad eseguire operazioni come la InSottoAlbero. Infatti ciascun albero ha un proprio vettore testa, quindi un' inserzione richiede la copiatura di due alberi in un terzo albero. Per ovviare a questo inconveniente si consente a tutti gli alberi di condividere uno stesso vettore in modo che le inserzioni siano consentite con semplici variazioni di valori all' interno del vettore comune. La variazione del puntatore figlio ad una testa di un albero permette il rapido aggancio. Per eseguire efficientemente l' operazione Padre si può imporre alla configurazione della cella un campo padre in cui manterremo un puntatore al padre. Una relizzazione di questo tipo tranne che per l' operazione CancSottoAlbero, consente ad ogni operatore un tempo di esecuzione di O(1). Per migliorare anche l' operazione CancSottoAlbero occorre prevedere anche un campo relativo al fratello precedente. Comunque il recupero delle celle dell' albero cancellato richiede un tempo lineare pari al numero dei nodi del sottoalbero poichè tale recupero avverrà comunque tramite una visita in ordine differito.

Altre considerazioni

Per migliorare l' efficenza di qualunque visita di un albero può essere conveniente utilizzare delle strutture dati di complemento quali le pile. Durante la discesa è possibile effettuare delle push che memorizzano i vari nodi che si vanno ad incontrare, in modo che una volta raggiunte le foglie sia possibile tornare indietro effettuando delle pop dalla pila che negli algoritmi ricorsiivi viene implicitamente utilizzato. Quindi nella procedura di visita, ogni volta che viene fatto un richiamo a se stessa, il valore corrente del nodo viene salvato nello stack per mezzo della push. Quando la procedura esce il valore precedente del nodo viene recuperato dallo stack tramite una pop. I movimenti verso l' alto vengono quindi nascosti dai meccanismi stessi della recursione. In questo caso si ha un sovraccarico di spazio necessario per lo stack. L' utilizzo di procedure ricorsive per processare un albero non deve sorprendere data la natura ricorsiva della definizione di albero. Se le procedure di processo non sono ricorsive allora occorrono stack espliciti. D' altra parte esistono dei metodi chè non fanno uso di stack o campi aggiuntivi per poter memorizzare un puntatore al padre e risalire un

albero. Consideriamo un albero binario con N nodi. Tale albero avrà 2N puntatori, dei quali solo N-1 avrà un valore diverso da **nil**. Nell' albero esistono quindi n+1 puntatori con valore nil che rappresentano un sottoalbero vuoto e che potrebbero essere rappresentati con un solo bit. Così facendo, possiamo utilizzare gli altri puntatori **nil** per identificare dei nodi superiori nell' albero. L' albero che ne risulta è chiamato albero *infilato*. I puntatori originariamente necessari per rappresentare l' albero binario sono chiamati *collegamenti* mentre i puntatori che erano **nil** e che ora sono utilizzati per l' infilatura sono chiamati *fili*. Per poter sviluppare degli efficienti algoritmi di navigazione dobbiamo considerare alcune convenzioni riguardanti l' infilatura:

a) Il filo sinistro identifica il nodo predecessore ordinato;
b) il filo destro identifica il nodo successore ordinato.

Questa convenzione consente di identificare sempre i nodi antecedenti. L' elemento che contiene il primo nodo ordinato non avrà predecessori e quindi il suo campo sinistro varrà nil. Similmente, il collegamento destro dell' elemento contenete l' ultimo nodo ordinato varrà nil.
Definiamo allora una nuova rappresentazione di un componente dell' albero:

```
type Etic = ( Colleg, Filo);
   ldComp = ^Comp;
   Comp = record
            Radice: Nodo;
            Etics, Eticd: Etic;
            Sinistra, Destra: ldComp
          end;
```

I campi Etic richiedono un solo bit di memoria ciascuno. Etics varrà Colleg se Sinistra identifica un sottoalbero (è un collegamento), altrimenti varrà Filo. Eticd è collegata a Destra in modo equivalente. Consideriamo dapprima come eseguire un attraversamento ordinato di un albero infilato, e quindi costruirne uno. Una navigazione ordinata attraversa sempre il sottoalbero sinistro prima di visitare la radice. Questo signifca che l' algoritmo seguirà i collegamenti sinistri. Nel nostro esempio l'albero ha un sottoalbero sinistro nullo, quindi la sua radice (A) è il primo nodo visitato. Indichiamo il componente che contiene il nodo attualmente visitato con **Corrente**. Siccome abbiamo appena visitato il nodo radice, attraverseremo il sottoalbero destro. Quindi se esiste dobbiamo seguire il collegamento destro per poi analizzare se esistono sottoalberi sinistri con l' intenzione di trovare il prossimo nodo in ordine. Se S è un ldComp possiamo implementare il nostro algoritmo tramite:

```
S:= Corrente^.Destro;
while S^.Etics = Colleg do
    S := S^.Sinistro;
Corrente := S
```

Se applichiamo questo algoritmo all' albero della figura precedente con Corrente che identifica inizialmente il componente con il nodo A, possiamo vedere che il secondo nodo in ordine sarà B. Ripetendo l' algoritmo arriveremo a C.
Esempio di Costruzione di un albero infilato:

```
Function Composizione (s: Albin; n: Nodo; d: Albin) : Albin;
(*Compone e ritorna un nuovo albero infilato partendo da s, n e d *)
    var
      Temp: Albin; (* per il nuovo albero in composizione *)
    begin
      New(temp);
      with temp^ do
        begin
          Radice := n;
          Sinistra := s;
          Destra := d
        end;
      if not Nullo(s)
        then
          begin
              (* il puntatore sinistro è un collegamento *)
              Temp^.Etics := Colleg;
              S := Ultimo(s); (* Identifica il successore *)
              S^.Destra := Temp;   (* e costruisce un filo *)
              S:Eticd := Filo
          end
        else                          (* Convenzione: lo pone a Filo *)
          Temp^Etics := Filo;
      if not Nullo(d)
        then
          begin
              (* il puntatore destro è un collegamento *)
              Temp^.Eticd := Colleg;
              D := Primo(D); (* identifica il predecessore *)
              D^.Sinistra := Temp;      (* e costruisce un Filo *)
              D.Eticd := Filo
          end
        else                          (* Convenzione: lo pone a Filo *)
          Temp^.Eticd := Filo;
      Compose := Temp
    end;
```

Queste istruzioni formano infatti le basi del nostro algoritmo. Per poterlo completare dobbiamo prevedere il caso di una visita ad una radice senza sottoalbero destro: in questo caso non ci sono più sottoalberi da attraversare, quindi dobbiamo utilizzare Filo per risalire l' albero. Abbiamo definito i fili destri che identificano i successori in

ordine, quindi se il puntatore Destro è un Filo, per identificare il successore dovremo semplicemente eseguire:

S:= Corrente^.Destro

Un attraversamento completo assume ora questa forma:

1) Identifica il Primo componente e chiamalo Corrente;

2) utilizza ripetutamente la funzione Successore per identificare i successori;

3) finisci quando non esistono più successori.

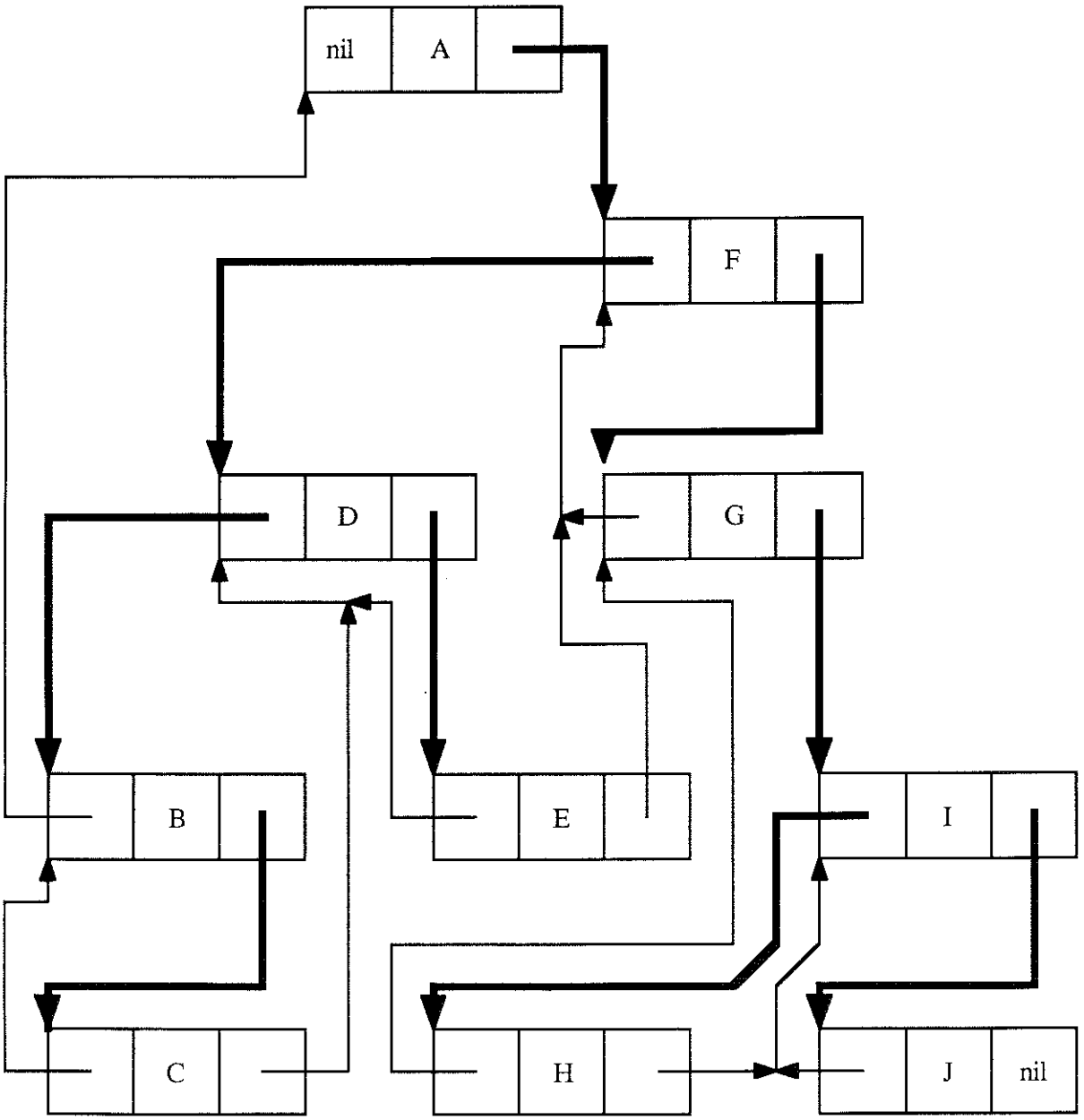


Grafico di un albero infilato

Abbiamo visto come sia possibile navigare all' interno di un albero sfruttando gli spazi di memoria che non vengono utilizzati, modificandone di fatto la struttura ad albero in

una struttura " a grafo", senza per questo snaturare il concetto di albero utilizzando la tecnica del "filo". Daremo ora un esempio di attraveramento di un albero senza utilizzare una tecnica come la "filo", che si porta sempre appresso dei puntatori di ritorno, con una che sfrutta quelli esistenti attraverso un meccanismo di inversione. Una metodologia utilizzata negli algoritmi per l' attraversamento degli alberi dinamici standard implica l' inversione dei collegamenti negli alberi. Praticamente, durante il movimento di navigazione verso il basso, cioè dalla radice alla foglia, ogni collegamento viene invertito fornendo una pista dalla foglia alla radice. Ripercorrendo questa pista i collegamenti dovranno essere ripristinati nella loro direzione originale. Anche in questo caso utilizzeremo delle etichette di un bit per ogni componente, per indicare se la pista da seguire risalendo l' albero è il collegamento destro o sinistro. La procedura di gestione dovrà fondamentalmente prevedere tre condizioni:

1) preludio: muoversi ad un sottoalbero non nullo;
2) interludio: visitare la radice e quindi muovere ad un sottoalbero non nullo;
3) postludio: risalire una pista nell' albero.

Esempio di un algoritmo che consente l' attraversamento di un albero utilizzando l' inversione di collegamento:

```
Procedure Ordina(A: Albin);
(* Stampa ordinatamente tutti i nodi di A usando uno spazio di memoria costante.
   Si assume che tutte le etichette siano inizializzate ad 1 (su verso sinistra) *))
   var
       Pred,          (* Predecessore corrente nella pista *)
       Corr,          (* nodo corrente sulla pista *)
       Succ: Albin;   (*probabile componente successore nella pista *)
   begin
       if not Nullo(A)
          then
                begin
                    Pred := nil;
                    Corr := A;
                    Sviluppo := Preludio;
                    repeat
                              case Sviluppo of
                                  Preludio: begin
                                  (* Prova il sotoalbero sinistro *)
                                  Succ := Corr^.Sinistra;
                                  if Succ<> nil     (* esiste ? *)
                                     then
                                  (* si: segui il collegamento ed invertilo *)
                                            begin
                                            (* inverte il collegamento *)
                                                  Corr^.Sinistra := Pred;
                                                  Pred := Corr;
                                            (* vai a sinistra *)
                                                  Corr := Succ
                                            (* lo sviluppo è già preludio *)
                                            end
```

```
                    else
                    (* no: visita la prossima radice *)
                    Sviluppo := Interludio
                end;
                Interludio: begin
                    Visita( Corr^.Radice);
                    (* Prova il sottoalbero destro *)
                    Succ := Corr^.Destra;
                    if Succ <> nil     (* esiste ? *)
                        then
                    (* si: segui il collegamento ed invertilo *)
                            begin
                        (* segnala l' inversione destra *)
                                    Corr^.Destra := Pred;
                                    Pred := Corr;
                        (* vai a destra *)
                                    Corr := Succ;
                        (* attraversa questo sottoalbero *)
                                    Sviluppo := Preludio
                            end
                        else
                    (* no: bisogna risalire la pista *)
                            Sviluppo := Postludio
                end;
                Preludio: begin
                if Pred^.etic = S (* risalire a sinistra ? *)
                    then ( * risali a sinistra *)
                        begin  (* ripristinalo *)
                            Succ = Pred^.Sinistra;
                        (* ripristina il collegamento *)
                            Pred^.Sinistra := Corr;
                            Corr := Pred;
                            (* e risali la pista *)
                            Pred := Succ;
                (* sinistra attraversata: visita la radice *)
                            Sviluppo := Interludio
                        end
                    else
                        begin
                            Succ := Pred^.Destra;
                            (* ripristina etichetta *)
                            Pred^.Etic := S;
                            (* ed il collegamento *)
                            Pred^.Destra := Corr;
                            Corr := Pred;
                            (* e risali la pista *)
                            Pred := Succ
                (* Sviluppo Postludio: è stato attraversato
                    l' intero sottoalbero quindi risali ancora *)
                        end
                    end
                end (* case *)
            until (Pred = nil) and (Sviluppo = Postludio)
        end
    end;
```

Altre implementazioni di alberi sono descritte in Knuth e Standish (vedi bibliografia). Analizzeremo ora un tipo particolare di alberi: i binari. Come abbiamo già anticipato, un albero binario è un particolare albero ordinato in cui ogni nodo ha al più due figli. Il riferimento ai figli è posizionale rispetto al nodo: sinistro o destro. L' importanza degli alberi binari deriva dalla semplicità della struttura che dalle innumerevoli applicazioni in cui possono essere impiegati. Il livello di un nodo in un albero binario è definito come segue:

la radice ha livello zero (0);
i nodi hanno un livello pari ad uno più di quello del padre.

Gli alberi binari subiscono delle distinzioni dovute alla loro struttura. Molti autori distinguono gli alberi binari in tre categorie:

1)	binari:	I nodi possono avere nessuno (foglia), uno (sinistro o destro) o due figli.
2)	strettamente binari:	I nodi possono avere nessuno (foglie) o due figli.
3)	completamente binari	Le foglie si trovano tutte al solito livello l e sono 2^l

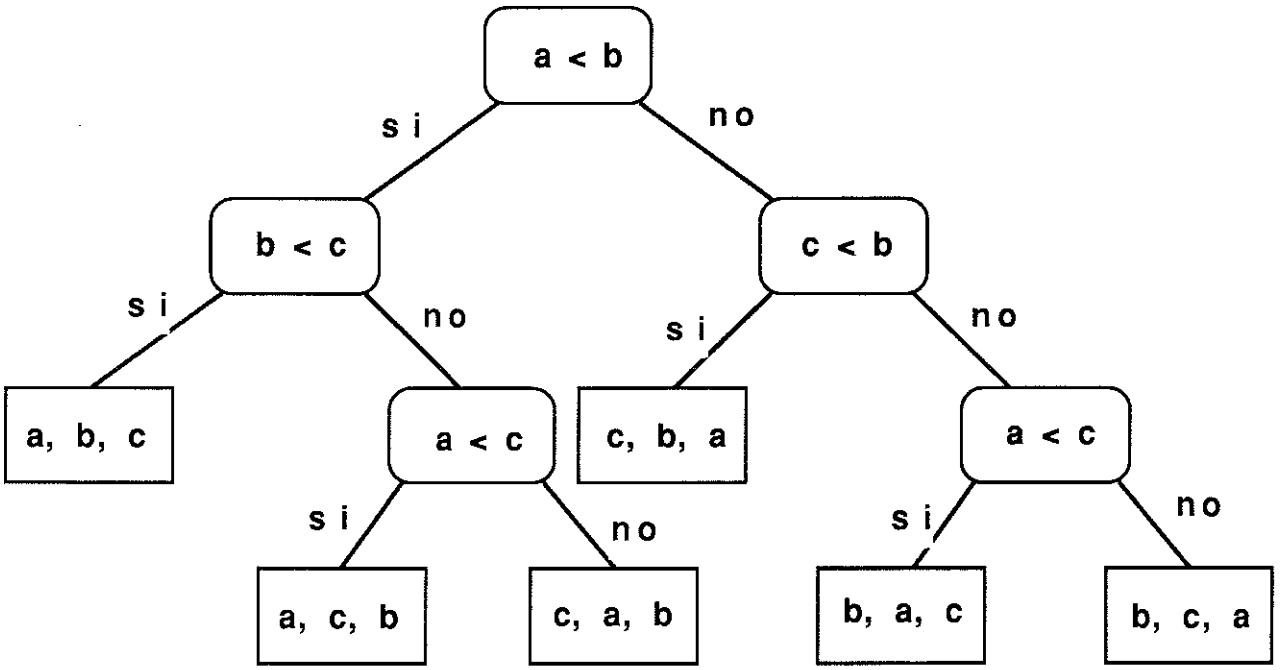
Un albero binario completo di profondità p ha un numero di nodi totale n_t uguale alla somma di nodi che si incontrano a ciascun livello:

$$n_t = 2^0 + 2^1 + 2^2 + \ldots + 2^p = \sum 2^j = 2^p - 1$$

Quale operazione inversa possiamo stabile che la profondità p dell' albero è :

$$p = \log_2 (nt + 1) - 1$$

La figura che segue mostra come la struttura ad albero consenta di risolvere un problema di decisione come quello che prevede l'ordinamento di tre numeri.
Poichè un qualsiasi algoritmo di ordinamento di n numeri deve fare una sequenza di scelte che permetta di individuare la soluzione corretta tra le n! permutazioni possibili dei dati di ingresso, l' albero di decisione per un qualsiasi algoritmo di ordinamento ha almeno n! foglie. Poichè ad ogni livello dell' albero il numero di nodi al più raddoppia, la lunghezza del percorso radice-foglia più lungo non può essere inferiore a $\log_2$ n!. Essendo n! = n(n-1) ... 3·2·1 ≥ n(n-1) ... $(n/2)^{n/2}$, si ha $\log_2$ n! ≥ (n/2) $\log_2$(n/2). Pertanto un limite inferiore alla complessità del problema dell' ordinamento è Ω (n log n).La struttura dati ad albero ha un vasto campo applicativo tanto da citare un notissimo algoritmo che utilizza la struttura in oggetto per la compressione dei file di testo : l' ALGORITMO di HUFFMAN.

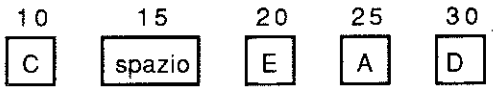


Prima di esplicitare l' algoritmo di Huffman facciamo un esempio di compressione di un file utilizzando le proprietà degli alberi. Supponiamo di operare in un insieme di computer collegati in rete che debboni trasmettersi una serie di messaggi composti da stringhe di lettere. Ovviamente i carichi delle rete sono proporzionali alla lunghezza dei messaggi; obiettivo: compressione dei testi. Chiaramente i caratteri più frequemente usati dovranno utilizzare i codici più corti dell' insieme; a quelli meno usati assegneremo quelli più lunghi.
Consideriamo un file di testi che utilizzi i sottoelencati caratteri con la relativa frequenza indicata di lato

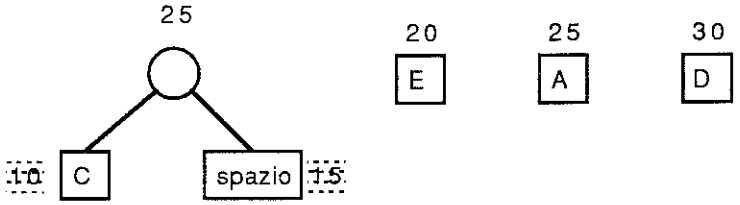
A	25
C	10
D	30
E	20
Spazio	15

Se utilizziamo una codifica standard richiediamo tre bit per carattere, producendo una lunghezza totale per il testo pari a 3n bit. Se utilizziamo l' algoritmo che porta il nome di Huffman costruiremo un albero di codifica a numero di bit variabile e le lunghezze dipenderanno dalla frequenza di utilizzo del simbolo in questione (ottimizzazione di trasmissione). Apporremo così i caratteri meno frequenti come foglie dei livelli più alti dell' albero che andiamo definendo, mentre quelli più frequenti saranno le foglie più vicino alla radice. Ciascun carattere viene memorizzato in un albero con la sola radice e assegnata la sua frequenza. Otteniamo così una foresta di alberi simili. Ad ogni

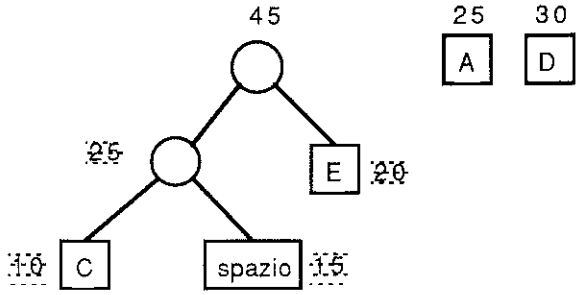
iterazione i due alberi con frequenza minore compongono un albero binario con un valore di frequenza pari alla somma delle frequenze dei simboli divenuti fratelli e rimesso nella foresta. Si itera nuovamente cercando nella foresta i due alberi con frequenza minore; si compongono in un nuovo albero; si sommano le frequenze; si riassegna l' albero ottenuto alla foresta e si procede finché la foresta non è diventata un solo albero. Prendendo spunto dall' esempio avremo questo andamento grafico:
Foresta pesata di partenza :



Iterazione nº 1 Compone un albero da C e spazio, con frequenza 25. La foresta ora sarà:

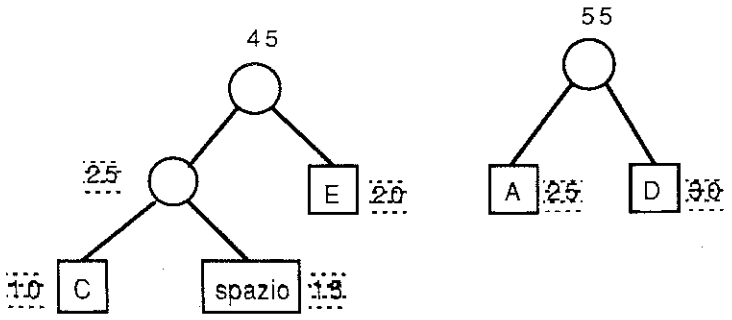


Iterazione nº 2 Compone E con il nuovo albero;

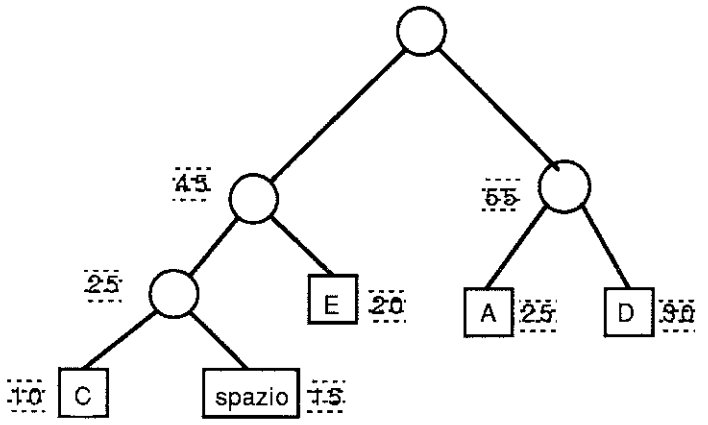


L' algoritmo genera un insieme ottimale, ma che non può essere unico in quanto poteva comporre anche E con A.

Iterazione nº 3 Compone A con D;



Iterazionen° 4 Compone i rimanenti alberi:



Assegna i codici ai caratteri facendo riferimento alle piste che dalla radice vanno alle foglie codificando con 0 se la discesa è sinistra 1 se la discesa è destra. Il risultato sarà:

A	25	10
C	10	000
D	30	11
E	20	01
Spazio	15	001

Se consideriamo il testo

A DEAD DECADE

avremo il seguente codice: 10001110110110011101000101101
lungo 29 bit.

Utilizzando un numero uguale di bit per ciascun carattere ne sarebbero stati necessari 39. In generale possiamo dire che il testo descritto richiederà Σ_i (freq$_i$ x lungh$_i$) per i che appartine a { A, C, D, E, Spazio}

= (0,25x2+0.1x3+0,3x2+0,2x2+0,15x3)n bit = 2,25n bit

che può essere confrontato con il 3n di partenza. Per un numero alto di n possiamo ottenere grossi vantaggi.

Possiamo quindi riassumere la nostra soluzione per la spedizione su rete di un file F:

1 contare la frequenza dei singoli caratteri usati nel file;
2 formare un albero di codifica;
3 utilizzare l' albero per determinare i codici di ciascun carattere;
4 trasmettere l' albero di codifica;
5 trasmettere il file utilizzando i codici precedentemente determinati.

Algoritmo di Huffman

L' algoritmo di Huffman è pensato per la costruzione di un albero di codifica ottimale. Il codice che si ottiene è di norma chiamato codice di Huffman. Riportiamo una presentazione ad alto livello, poiché nella letteratura è possibile trovare ampia documentazione della trasformazione dell' algoritmo in programma.
Consideriamo F come una foresta di alberi di foglie ciascuno dei quali è composto inizialmente solo dalla radice, che è naturalmente una foglia. Associamo ad ogni albero a di F una frequenza relativa, af.

Consideriamo gli alberi a_1, a_2 ed A;
While esiste in F più di un albero **do**
begin
seleziona a_1 in F tale che $a_1.f$ sia il minimo in F
togli a_1 da F
seleziona a_2 in F tale che $a_2.f$ sia il minimo in F
togli a_1 da F
costruisci A con
Sinistro(A) = a_1
Destro(A) = a_2
$A.f = a_1.f + a_2.f$
inserisci A f in F
end;

L' albero finale che rimarrà in F sarà l' albero di ottimizzazione che si cercava.

	APPARTIENE	INSERISCI	CANCELLA	MIN	CANCMIN
vettore bool	O(1)	O(1)	O(1)	O(N)	O(N)
liste no ordinate	O(n)	O(1)	O(n)	O(n)	O(n)
liste ordinate	O(n)	O(n)	O(n)	O(1)	O(1)
vettore ord	O(logn)*	O(n)	O(n)	O(1)	O(n)
hash (medio)	O(1)	O(1)	O(1)	O(max{m,n}	O(max{m,n}
hash (pessim)	O(max{m,n}	O(max{m,n}	O(max{m,n}	O(max{m,n}	O(max{m,n}
heap	O(n)	O(logn)**	O(n)	O(1)	O(logn)
tree AVL 2-3-4	O(logn)	O(logn)	O(logn)	O(logn)	O(logn)

* Sapendo a priori che l'elemento da inserire non è già presente, altrimenti la complessità è O(n).
* * Se è utilizzabile l 'interpolazione allora la complessità è O(log logn)

Alberi bilanciati di ricerca.

Le varie realizzazioni di struttura informative viste finora privilegiano l' efficienza di certi operatori rispetto ad altri. A tale proposito Bertossi riepiloga nella tabella precedente la complessità degli operatori APPARTIENE, INSERISCI, CANCELLA, MIN e CANCMIN, sia nel caso medio, che in quello pessimo, per tutte le realizzazioni considerate:

Gli alberi AVL

Gli alberi AVL presentano una struttura tale che:

1) B è "1 bilanciato", cioè per ogni nodo il livello massimo o profondità che si incontra scendendo nel suo sottoalbero sinistro differisce si al più uno dal livello massimo che si incontra scendendo nel suo sottoalbero destro;

2) per ogni u, tutti gli elementi contenuti nel sottoalbero sinistro sono minori dell' elemento contenuto in u;

3) per ogni nodo u , tutti gli elementi contenuti nel sottoalbero destro sono maggiori dell' elemento contenuto in u.

I vincoli che definiscono il sottoinsieme degli alberi 1-bilanciati determinano:

a) una crescita logaritmica del numero delle foglie nel numero dei nodi:

Poiché detto n_k il numero minmo di nodi tra tutti gli alberi 1-bilanciati di livello massimo, è verificabile come:

$n_0 = 1$; $n_1 = 2$, ed $n_k = n_{k-1} + n_{k-2} + 1$, per $k \geq 2$.

Questa relazione di ricorrenza è molto simile alla relazione di Fibonacci :

$F_0 = 0$, $F_1 = 1$, ed $F_k = F_{k-1} + F_{k-2}$, per $k \geq 2$

quindi:

$n_k = F_{k+3} - 1$ ed essendo $F_{k+3} \geq (1/\sqrt{5}) [(1 + \sqrt{5})/2]^{k+3} - 1$

avremo

$n_k \geq (1/\sqrt{5}) [(1 + \sqrt{5})/2]^{k+3} - 2$

in conclusione:

$\log(n_k + 2) > \log(1/\sqrt{5}) + (k+3) \log[(1 + \sqrt{5})/2]$

così da affermare che

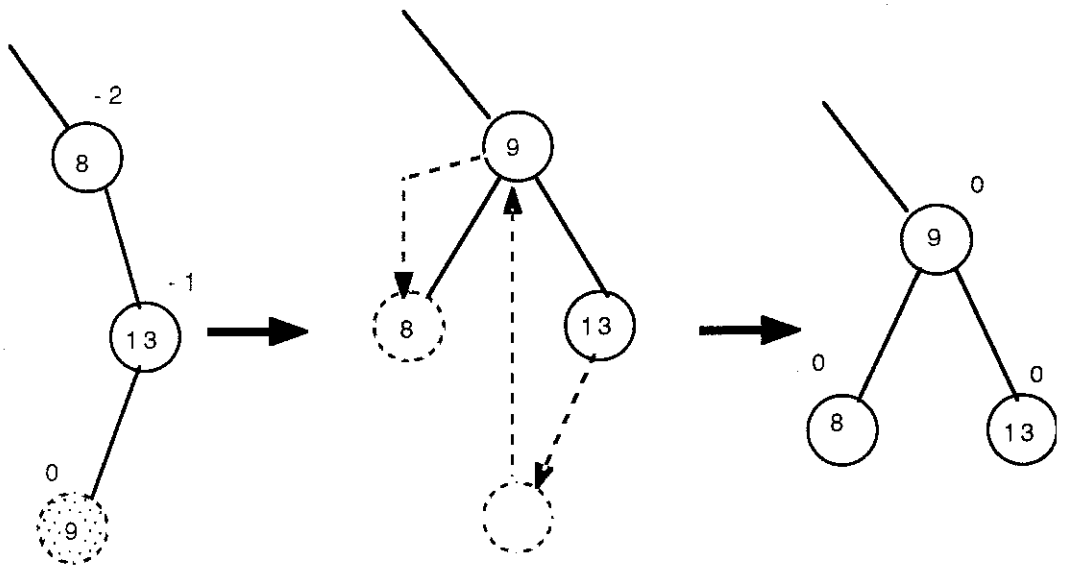
$k = O(\log n)$.

Le proprietà successive hanno il compito di semplificare la ricerca di un elemento. In definitiva ci riportano alla ricerca binaria.

Il bilanciamento viene controllato pesando ciascun nodo dell' albero. Se il peso non appartiene all' insieme B={+1, 0, -1} l' albero risulta sbilanciato. . Il "fattore di bilanciamento" risulta uguale alla differenza tra la profondità del sottoalbero sinistro di u dal sottoalbero destro di u. Nel caso in cui il fattore di bilanciamento sia un

numero che non appartiene all' insieme B, sono previsti, come vedremo, degli interventi riequilibratori.
La complessità della procedura APPARTIENE è O(log n), poichè tale procedura si basa su una discesa lungo il percorso radice foglia. Infatti:

```
function APPARTIENE (x: tipoelem; var Å: insieme): boolean;
   begin
      if Å = nil then
         APPARTIENE = false
      else if x = Å^.elemento then
         APPARTIENE = true
      else if x < Å^.elemento then
         APPARTIENE := APPARTIENE (x, Å^.sinistro)
      else
         APPARTIENE := APPARTIENE (x, Å^.destro)
end;
```



Inserzione di 9 con rotazione.

Per l' inserzione, come per la cancellazione occorre verificare che il fattore di bilanciamento appartenga all' insieme B. Qualora tale fattore non appartenga a B si riconduce ad un valore accettato attraverso un correttivo detto "rotazione". La rotazione prevede lo spostamento a foglia del nodo che presenta lo sbilanciamento e quello a nodo della foglia. Perciò se l' inserzione di un elemento x provoca sbilanciamento si presentano quattro casi:

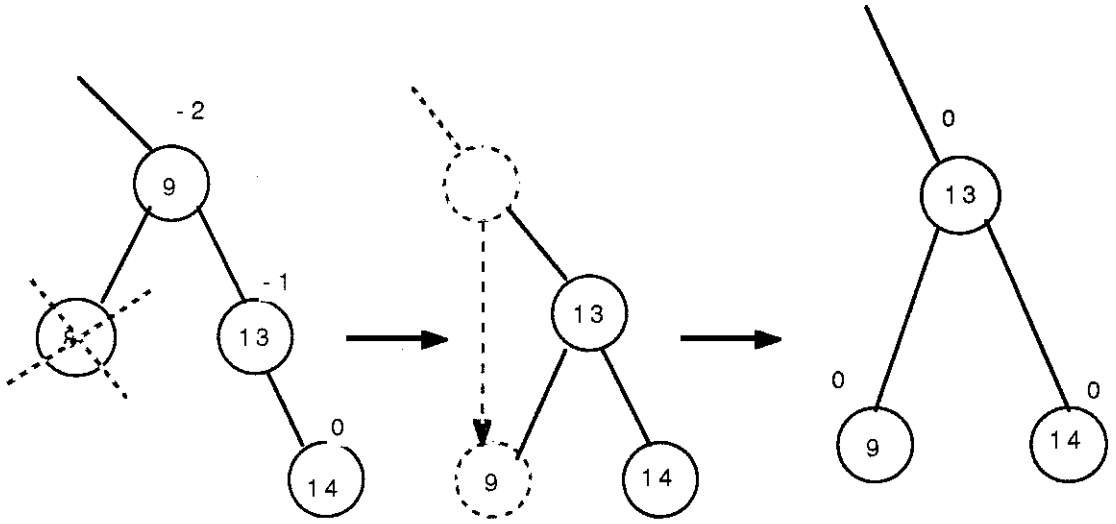
(SS) Lo sbilanciamento è dovuto all 'inserimento di x nel sottoalbero sinistro del figlio sinistro del nodo critico;

(SD) Lo sbilanciamento è dovuto all' inserimento di x nel sottoalbero destro del figlio sinistro del nodo critico;

(DD) Lo sbilanciamento è dovuto all' inserimento di x nel sottoalbero destro del figlio destro del nodo critico (analogo ad SS);

(DS) Lo sbilanciamento è dovuto all 'inserimento di x nel sottoalbero sinistro del figlio destro del nodo critico (analogo ad SD).

Per bilanciare sono previste delle rotazioni che prevedono la discesa del nodo critico a foglia sostituendolo (nodo critico) con l' elemento intermedio.



Cancellazione di 8 con rotazione

Alberi 2-3-4

Gli elementi di un insieme I possono essere contenuti nei nodi di un albero ordinato Å tale che:

1) ogni nodo non foglia (interno) ha K figli, con $2 \le k \le 4$, e contiene k-1 elementi di I, ordinati in senso crescente.;

2) le foglie non contengono nessun elemento di I;

3) tutte le foglie sono sullo stesso livello;

4) se un nodo u ha figli, allora l'elemento h-esimo di u è maggiore di tutti gli elementi contenuti nel sottoalbero la cui radice è l' h-esimo figlio di u, ed è minore di tutti gli elementi contenuti nel sottoalbero la cui radice è l'(h+1)-esimo figlio di u, per $1 \le h \le k-1$.

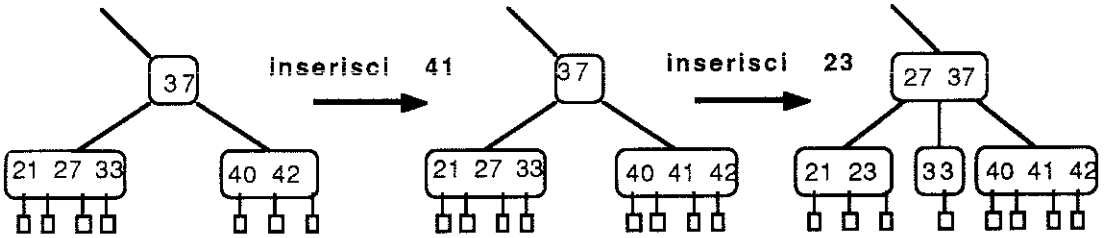
E' immediato verificare che le proprietà 1) e 3), in questo tipo di alberi con n elementi il livello L delle foglie verifica la seguente relazione:

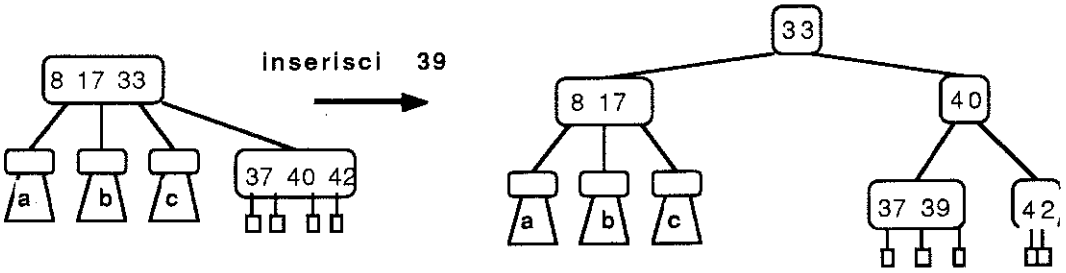
$$[\log_4(n+1)] \le L \le [\log_2(n+1)]$$

Le proprietà 1) e 4) consentono una rapida ricerca di un elemento: è sufficiente confrontare l' elemento da ricercare con gli elementi contenuti nella radice e, se l' elemento della radice è minore dell' (h+1)-esimo, riapplicare il procedimento all'

(h+1)-esimo sottoalbero. In questo modo, sia la funzione APPARTIENE che MIN possono essere realizzate in modo da richiedere $O(\log n)$ tempo per essere eseguite.

Per inserire un elemento non presente, occorre individuare il nodo in cui dovrebbe essere contenuto (APPARTIENE con complessità $O(\log n)$. Se tale nodo che si trova sempre a livello L-1, contiene 1 o 2 elementi, si effettua l' inserzione, aggiungendo una nuova foglia.





Se invece il nodo contiene 3 elementi, lo si sdoppia in due nodi fratelli contenenti rispettivamente 2 e 1 elementi e si inserisce il quarto elemento nel padre, rispettando le proprietà 1) e 4). Nel caso pessimo lo sdoppiamento si riperquote anche nel padre , e poi nel nonno, e così via fino a sdoppiare anche la radice. Poiché ogni sdoppiamento richiede tempo costante e si hanno al più L sdoppiamenti, l' operazione INSERISCI così realizzata ha complessità $O(\log n)$.

Per eliminare un elemento presente nell' albero, si individua il nodo che lo contiene. Si possono presentare tre casi di cui uno può essere distinto in tre ulteriori sottocasi.

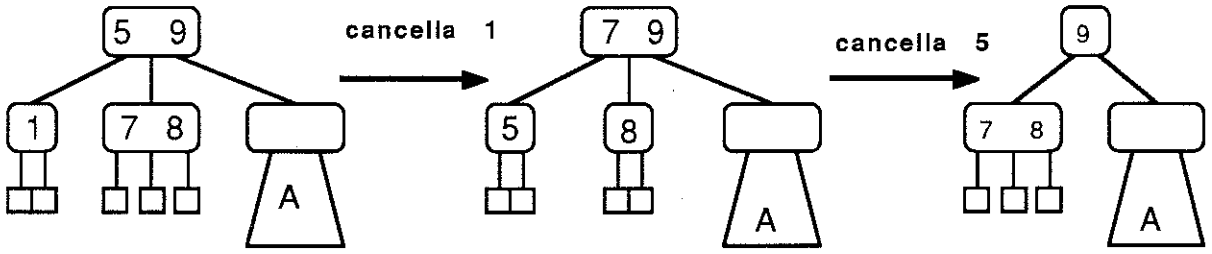
a) u è a livello L-1, cioè i suoi figlisono foglie, e contiene $h \geq 2$ elementi: si cancellano l'elemento ed un figlio di u (la foglia corrisponedente);

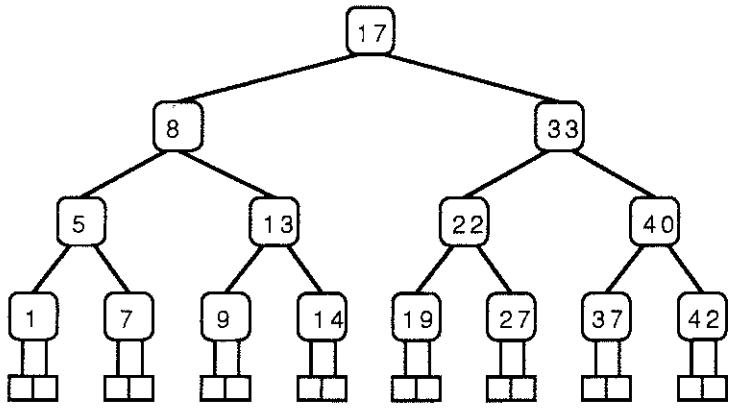
b) u è a livello L-1, ma contiene un solo elemento (quello da cancellare):

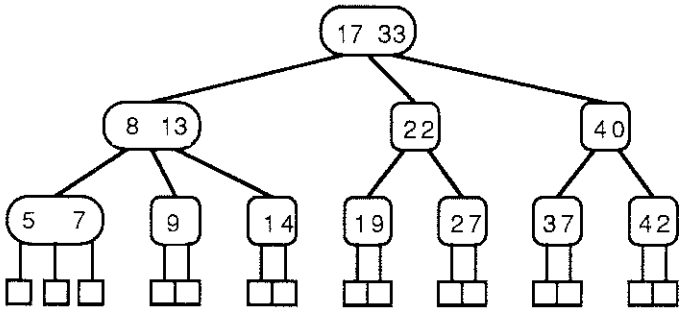
   b1) se il fratello precedente (o successivo) di u ha almeno 2 elementi si ridistribuiscono gli elementi tra i due fratelli;

   b2) se né il fratello precedente né il successivo hanno almeno due elementi ma il padre ce n' ha almeno 2, si elimina uno dei due figli e l' elemento intermedio è inserito nell' altro figlio;

b3) se sia il fratello precedente, che il successivo, che il padre, hanno un solo elemento, occorre propagare la ridistribuzione degli elementi agli antenati;







c) u è a livello minore di L-1, cioè i suoi figli non sono foglie: ci si riduce al caso a) o al caso b) sostituendo l'elemento cancellato x con il più piccolo elemento y tale che y>x, che si trova in un nodo v di livello L-1 (per l'esattezza se x è h-esimo elemento di u, y si trova il nodo "più a sinistra" di livello L-1 che si raggiunge scendendo per il figlio (h+1)-esimo di u).

E' facile convincersi che la complessità di CANCELLA E CANCELLAMIN, realizzate come descritto, è $O(\log n)$.

Bibliografia

M. AUGENSTAIN - A. TENENBAUM Data structures Using Pascal Prentice Hall International

BRIAN J. LINGS La strutture dei Dati. Un approccio sistemico in Pascal Franco Angeli

A.A. BERTOSSI Strutture Algoritmi Complessità ECIG

W.J. COLLINS Data Structures: an object-oriented approach ADDISON- WESLEY Publishing Company

D.W. CARROLL Programmazione in Turbo Pascal Mc Graw Hill

F. LUCCIO Strutture Linguaggi Sintassi Una Introduzione Boringhieri.

T. A. STANDISH Data Structure Tecniques. Addison-Wesley, Reading, MA.

N. WIRTH Algoritmi+Strutture dati = Programmi Tecniche Nuove

N. WIRTH Principi di Programmazione Strutturata ISEDI

S. WOOD Guida al Turbo Pascal 4.0 Borland- Osborne