Tabella dei Simboli

Dispensa del corso di Linguaggi e Traduttori

A.A. 2005-2006

Giovanni Semeraro

Le tabelle dei simboli servono a due scopi: controllano la correttezza semantica (la parte dipendente dal contesto della grammatica) ed aiutano nella generazione del codice.

Tutte queste attività si ottengono mediante accesso alla tabella dei simboli per inserire o leggere alcuni attributi delle variabili del programma sorgente.

Questi attributi sono tipo, nome, dimensione, indirizzo e si determinano direttamente dalla dichiarazione o implicitamente dal contesto in cui le variabili compaiono.

Qui discuteremo 1'organizzazione della tabella dei simboli e la modalità per creare ed accedere simboli.

La tabella dei simboli è inserita in memoria centrale e viene normalmente modificata dinamicamente.

Ogni identificatore del programma sorgente richiede da parte del compilatore accessi alla tabella dei simboli. Questo spiega perchè l'accesso alla tabella dei simboli sia una delle parti più costose in tempo del processo di compilazione e si debba cercare di ottimizzare.

Quindi e' importante studiare metodi efficienti per l'accesso alla tabella dei simboli.

Strutture dati: tavole o tabelle

Una tavola o tabella è un tipo di dato astratto per rappresentare insiemi di coppie ***<chiave, attributi>.*** Ciascuna coppia rappresenta dati riferiti ad un' unica entità logica (ad es., persona, documento, etc.) identificata in modo univoco dalla ***chiave.***

Esempio:
Matricola,   Cognome, Nome, DataDiNascita, ...

Operazioni tipiche sulle tavole:

- inserimento di un elemento <Chiave, Attributi>
  inserisci: tavola x chiave x attributi → tavola

- cancellazione di un elemento (nota la chiave)
  cancella: tavola x chiave → tavola

- verifica di appartenenza di un elemento
  esiste: tavola x chiave → boolean

- ricerca di un elemento nella tavola
  ricerca: tavola x chiave → attributi

L'operazione di ricerca è la più importante (soprattutto nelle TS sei compilatori). Spesso la rappresentazione concreta viene scelta in modo da ottimizzare questa operazione.

In Pascal, ciascun elemento della tavola rappresenta una ***aggregazione*** di dati → ***tipo record.*** La chiave e ciascun attributo corrispondono ad un ***campo del record.***

Quando costruire ed interagire con la tabella dei simboli

In un compilatore a molti passi, la tabella dei simboli (TS) viene creata durante l'analisi lessicale.

Gli indici degli entries per le variabili nella TS formano parte della stringa di tokens prodotta dallo scanner.

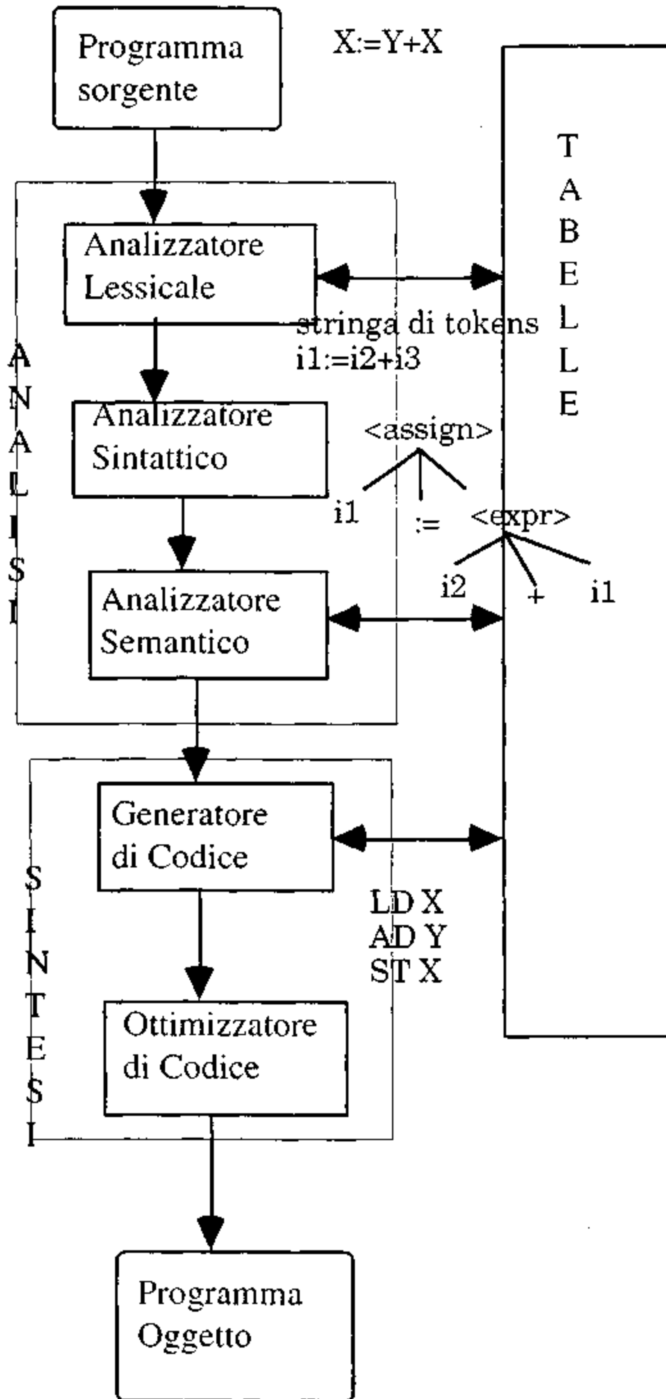
Nella figura seguente se X e Y occupano le posizioni 1 e 2 nella TS si generano i tokens i1 e i2.

Quando il parser produce l'albero sintattico, le foglie hanno dei riferimenti alla tabella dei simboli.

L'analisi sintattica comunque non produce modifiche a TS.

Solo durante l'analisi semantica e la generazione di codice si possono assegnare valori agli attributi della variabile in TS.

Si pensi alla dichiarazione esplicita di tipo.


Programma sorgente
X:=Y+X
TABELLE
ANALISI
Analizzatore Lessicale
stringa di tokens
i1:=i2+i3
Analizzatore Sintattico
<assign>
i1
:=
<expr>
i2
+
i1
Analizzatore Semantico
SINTESI
Generatore di Codice
LD X
AD Y
ST X
Ottimizzatore di Codice
Programma Oggetto

Contenuti della TS

Una TS e' costituita da una serie di righe ognuna delle quali contiene una lista di valori di attributi associati con una particolare variabile.

Gli attributi dipendono dal linguaggio di programmazione che si compila (se non ha i tipi non comparirà tale attributo).

Variable Name	Address	Type	Dimen sion	Line Declared	Lines Referenced	Pointer
COMPANY	0	2	1	2	9,14,25	7
X3	4	1	0	3	12,14	0
FORM1	8	3	2	4	36,37,38	6
B	48	1	0	5	10,11,13,23	1
ANS	52	1	0	5	11,23,25	4
M	56	6	0	6	17,21	2
FIRST	64	1	0	7	28,29,30,38	3

Si possono considerare i seguenti attributi:

1. Nome della variabile;
2. Indirizzo nel codice oggetto;
3. Tipo;
4. Numero dei parametri di una procedura (o dimensione della variabile);
5. Linea sorgente in cui la variabile e' dichiarata;
6. Linee sorgenti in cui la variabile e' referenziata;
7. Puntatori per listarli in ordine alfabetico.

Ora commentiamo tali attributi.

1. Un problema puo' essere la dimensione della stringa che compone il nome. E' inserito dallo scanner.

2. L' indirizzo indica la locazione relativa delle variabili a run-time. Questo indirizzo e' inserito nella TS quando una variabile e' dichiarata o incontrata per la prima volta. E' richiamato dalla tabella quando la variabile e' referenziata nel codice per generare il codice oggetto corrispondente.

Per un linguaggio senza allocazione dinamica di memoria (es FORTRAN) gli indirizzi sono attribuiti in modo sequenziale da 1 a M dove M e' la massima dimensione di memoria allocata per il programma.

Per linguaggi a blocchi, l'indirizzo e' rappresentato da una coppia <BL (block level), ON (occurrence number)>. A run-time ON rappresenta l'offset rispetto al blocco.

3. Il tipo puo' essere implicito (es. FORTRAN), esplicito (es. PASCAL) o non esistere (es. LISP, PROLOG). Il tipo della variabile e' fondamentale per il controllo semantico. Ad esempio, S*3 e' scorretto se S e' dichiarato come una variabile di tipo stringa.

Il tipo serve anche per sapere quante memoria deve essere allocata per la variabile. Il tipo e' inserito normalmente con una codifica (numero intero).

4. La dimensione e' importante per il controllo semantico. Se si tratta di un array ci dice le sue dimensioni e serve anche per calcolare l'indirizzo di un particolare elemento dell'array. In una procedura indichiamo il numero dei parametri per procedere al controllo durante le chiamate.
Negli esempi dati qui gli scalari sono considerati di dimensione 0, vettori 1, matrici 2 ecc.

7. Il Pointer serve per costruire, ad esempio, una lista per ordine alfabetico della tabella dei simboli od una cross-reference (con i riferimenti di dove e' dichiarata e referenziata nel codice sorgente).

Operazioni sulla tabella dei simboli

Le operazioni piu' comuni sono **l'inserimento** e **la ricerca.**

Se il linguaggio ha dichiarazioni esplicite di tutte le variabili allora l'inserimento avviene al momento della dichiarazione.

Se la tabella dei simboli e' ordinata, ad esempio, alfabeticamente mediante il nome della variabile si deve fare anche una ricerca nella tabella e quindi diventa un procedimento costoso.

Se invece la tabella e' disordinata l'inserimento e' rapido, ma diventa poi piu' inefficiente la ricerca.

Le operazioni di ricerca vengono svolte per ogni riferimento a variabili in istruzioni non di dichiarazione.

L'accesso serve per il controllo semantico e la generazione di codice, nonche' per la rilevazione di errori se le variabili non si trovano all'interno della tabella.

Quando un linguaggio ha la possibilita' di dichiarare variabili implicitamente allora le operazioni di inserimento e ricerca sono legate strettamente.

Ogni riferimento ad una variabile genera una ricerca ed eventualmente un inserimento se non e' gia' stata inserita in TS.

Linguaggi a blocchi

Per linguaggi a blocchi sono necessarie due operazioni addizionali che chiamiamo **set** e **reset.**

Set si invoca quando si entra in un blocco, reset quando si esce.

Questo e' necessario perche' in un linguaggio a blocchi, una variabile con lo stesso nome puo' essere dichiarata in punti diversi del programma ed assumere attributi diversi.

In un linguaggio innestato e' importante assicurare che per ogni istanza di un nome di variabile sia associato un unico entry nella TS.

Quindi ad ogni inizio di un blocco l'operazione di set attribuisce una nuova sotto tabella per gli attributi delle variabili dichiarate nel nuovo blocco.

Supponiamo che le sottotavole siano create attribuendole numeri interi crescenti.

Se la ricerca comincia nella sottotavola N e poi procede fino alla sottotavola 1, la variabile che si trova e' l'ultima che si e' inserita (secondo le regole di scope di linguaggi a blocchi) ed e' eliminata ogni ambiguita'.

All'uscita del blocco, l'operazione di reset rimuove l'ultima sottotabella allocata. Le variabili di quel blocco, infatti, non possono piu' essere referenziate.

Organizzazione delle tabelle dei simboli per linguaggi non a blocchi

Intendiamo linguaggi in cui ogni singola unita' di compilazione e' un modulo senza alcun sottomodulo.

Tutte le variabili dichiarate nel modulo possono essere referenziate in ogni parte del modulo.

Tabelle dei simboli non ordinate

Il modo piu' semplice di organizzare una tabella dei simboli e' quello di aggiungere gli entries alla tabella nell'ordine in cui le variabili sono dichiarate.

In questo modo per un inserimento non e' richiesto nessun confronto, mentre una ricerca richiede, nel caso peggiore, il confronto con gli N elementi all'interno della tabella.

Poiche' cio' e' inefficiente, questa organizzazione dovrebbe essere adottata solo se la dimensione della TS e' piccola.

Altrimenti si deve ricorrere ad altre organizzazioni.

Tavole: rappresentazione in Pascal

In memoria centrale, realizzata attraverso ***array*.**

Esempio:

```
const   N=100;
type    key=packed array[1..10] of char;
        elem=record
          chiave: key;
          address: integer;
          typeel: integer;
          dimension: integer;
          lineD: integer;
          lineR: array [1.. 100] of integer;
          pointers: 0..N
          end;
   index= 1..N;
   Tavola= array [index] of elem;

var T: Tavola;
```

E' necessario porre un limite massimo alla dimensione della tavola (al massimo N elementi).

Lo spazio di memoria occupato e' fisso (indipendente dal numero di elementi della tavola).

La ricerca e' sequenziale ed esaustiva (caso peggiore, N confronti).

```
procedure ricerca(var T:Tavola; k: key;
                        var el: elem; var trovato: boolean);
{Ricerca elemento con chiave k nella tavola T realizzata
mediante array}
var i: 1..N;
begin
    i:=1;
    trovato :=false;
    while (i<=N) and (not trovato) do
        if T[i].chiave=k
        then    begin
                trovato:=true;
                el:=T[i]
                end
        else    i:=i+l          {end while }
end;    {ricerca}
```

Esercizio:
Completare il tipo di dato astratto Tavola che descrive la tabella dei simboli precedente, con ***rappresentazione sequenziale*** mediante **array**. Definire le procedure inserisci, cancella, esiste ed utilizzarle in parti del compilatore.

Tabelle dei simboli ordinate

La posizione dell'elemento nella TS e' determinato dal nome della variabile (ordinamento lessicale).

In questo caso un inserimento e' sempre accompagnato ad una procedura di ricerca.

L'inserimento puo' inoltre richiedere uno spostamento di altri elementi gia' inseriti nella tabella (e' la maggiore fonte di inefficienza).

Ottimizzazione della ricerca: si ottiene se gli elementi sono memorizzati in modo ordinato nella tavola.

Deve esistere un ordinamento sul campo Chiave. Questo induce un ordinamento sugli elementi della Tavola come segue:

$el_1 = \langle k_1, Attr_1 \rangle$
$el_2 = \langle k_2, Attr_2 \rangle$

$el_1 < el_2$ ***se e solo se*** $k_1 < k_2$

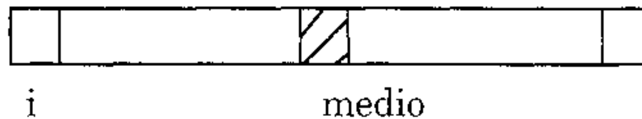
La ricerca puo' arrestarsi appena si incontra un elemento con chiave maggiore di quella cercata (caso medio N/2 confronti, caso peggiore N confronti).

Si complica l'operazione di inserimento di un nuovo elemento nella tavola → occorre mantenerla ordinata.

Miglioramento ulteriore: ***ricerca binaria.***
Si accede all'elemento mediano <Chiave,Attr> della tavola.
Se Chiave=k, fine della ricerca.
Altrimenti,

> se k<Chiave, ripeti la ricerca nella prima meta' della tavola;
> se k>Chiave, ripeti la ricerca nella seconda meta' della tavola.

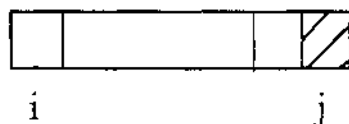
Ricerca binaria su tavola ordinata realizzata mediante array:



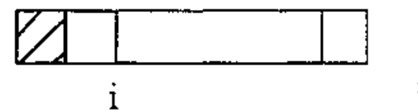
medio = (i + j) div 2

if T[medio].chiave=k → trovato

if T[medio].chiave > k



if T[medio].chiave < k



II procedimento si arresta quando:

i= j

```
procedure ricerca_bin (var T:Tavola; k: key;
            i,j: index; var el: elem; var trovato: boolean);

{Ricerca binaria di un elemento con chiave k nella
 tavola ordinata T realizzata mediante array}

var medio: 1..N;
begin
      trovato:=false;
      if (j>i) and (not trovato) then
            begin
            medio:= (i + j) div 2;
            if T[medio].chiave=k
            then trovato:=true
            else
                  if T[medio].chiave>k
                  then
                  ricerca_bin(T,k,i,pred(medio),el,trovato)
                  else
                  ricerca_bin(T,k,succ(medio),j,el, trovato)
            end
      else if T[i].chiave=k then trovato:=true
end;        {ricerca binaria}
```

Ad ogni passo si eliminano dalla ricerca meta' degli elementi della tavola "corrente" (indici da i a j).

Nel caso peggiore si eseguono $\log_2 N$ confronti.

Tavole: rappresentazione collegata e ad albero

II tempo per inserire un elemento in una TS ordinata si puo' ridurre utilizzando una struttura collegata od ad albero.

Nel caso di ***rappresentazione collegata*** di una tavola, si utilizza una lista semplice:

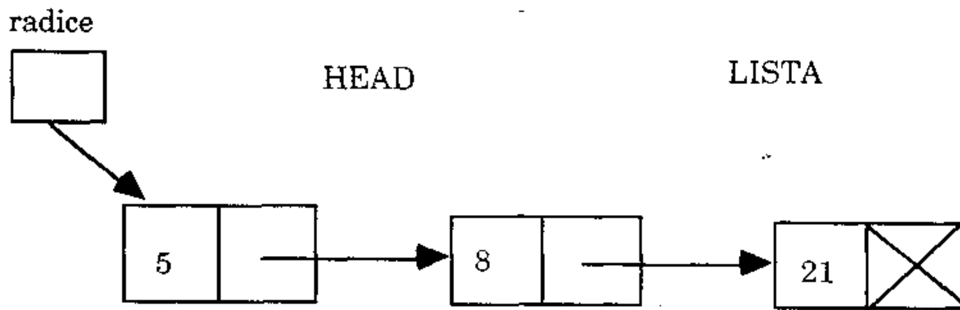
```
const   N=100;
type    key=packed array[1..10] of char;
         ptr_tavola=^elem;
         elem=record
                 chiave: key;
                 address: integer;
                 typeel: integer;
                 dimension: integer;
                 lineD: integer;
                 lineR: array[1..100] of integer;
                 next: ptr_tavola
         end;
     index= 1..N;
     Tavola= array [index] of elem;

var T: Tavola;
```

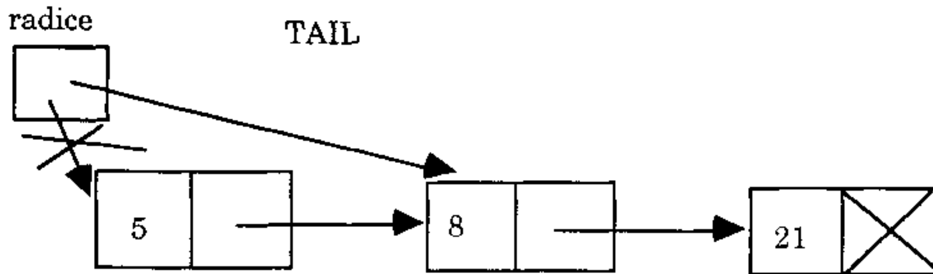
La rappresentazione collegata:

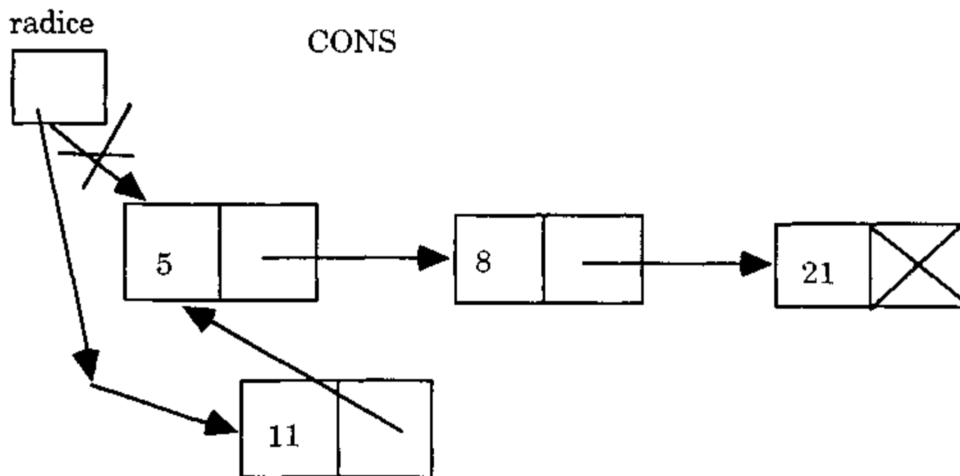
Idea fondamentale: memorizzare gli elementi associando a ciascuno una particolare informazione ***(riferimento)*** che permetta di individuare la locazione in cui e' inserito l'elemento successivo.

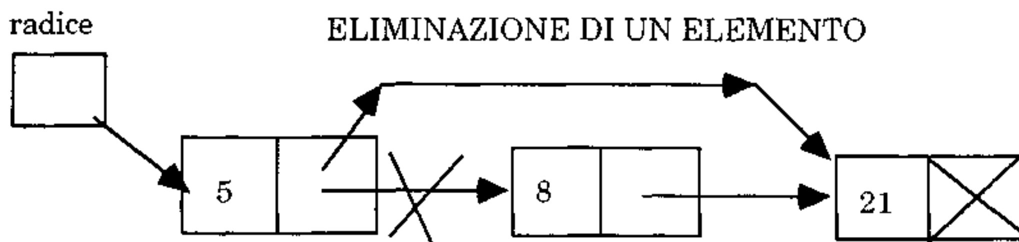
La sequenzialita' degli elementi della lista non e' rappresentata mediante l'adiacenza delle locazioni di memoria in cui sono memorizzati (nel caso dell'esempio precedente c'era il campo pointers che era un indice.

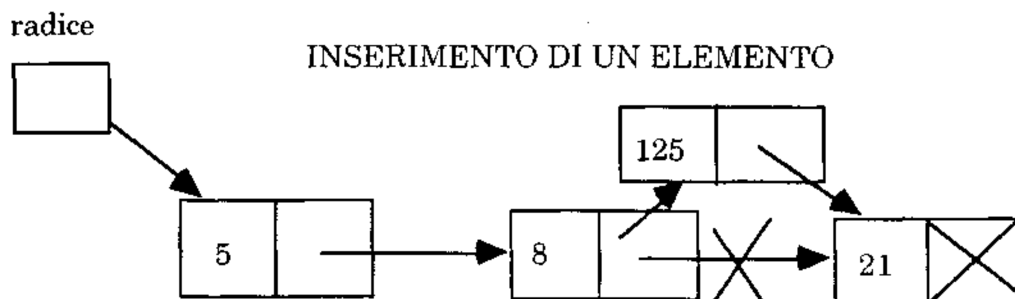
Notazione grafica per rappresentazione collegata:
elementi della lista come nodi e riferimenti come archi.
Lista: [5, 8, 21]


radice
HEAD
LISTA
5
8
21


radice
TAIL
5
8
21


radice
CONS
5
8
21
11


radice
ELIMINAZIONE DI UN ELEMENTO
5
8
21


radice
INSERIMENTO DI UN ELEMENTO
125
5
8
21

Non c'é un limite massimo alla dimensione della tavola.

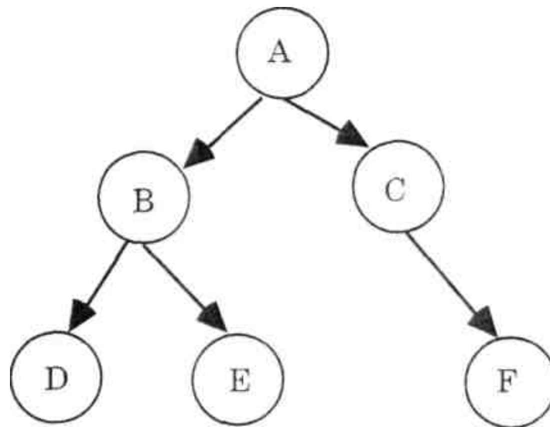
Lo spazio di memoria occupato dipende solo dal numero di elementi della tavola.

Se la lista é ordinata sulla chiave, si puo' applicare la ricerca ordinata (ci si arresta al primo elemento con chiave >=).

Tabelle dei simboli strutturate ad albero

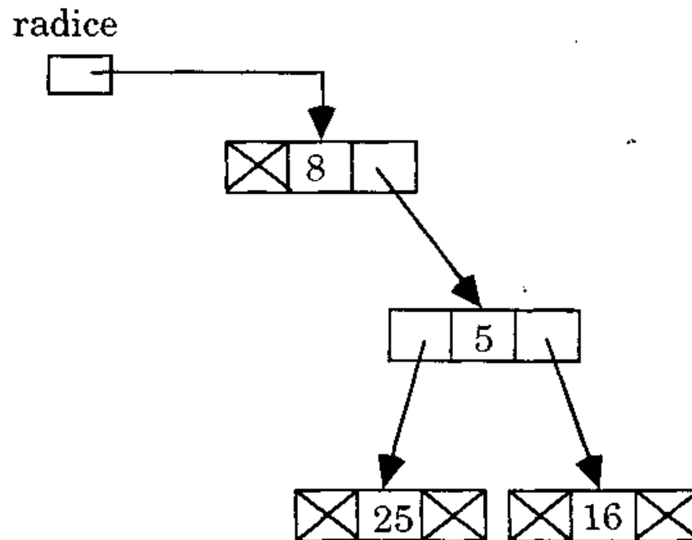
Alberi binari

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



Definizione induttiva:

Un albero binario e' un grafo orientato aciclico che o é vuoto (cioè ha un insieme vuoto di nodi), oppure è formato da un nodo radice e da due sotto-alberi binari (sinistro e destro).
Rappresentazione più efficiente, direttamente attraverso **record** e **pointer** in Pascal:



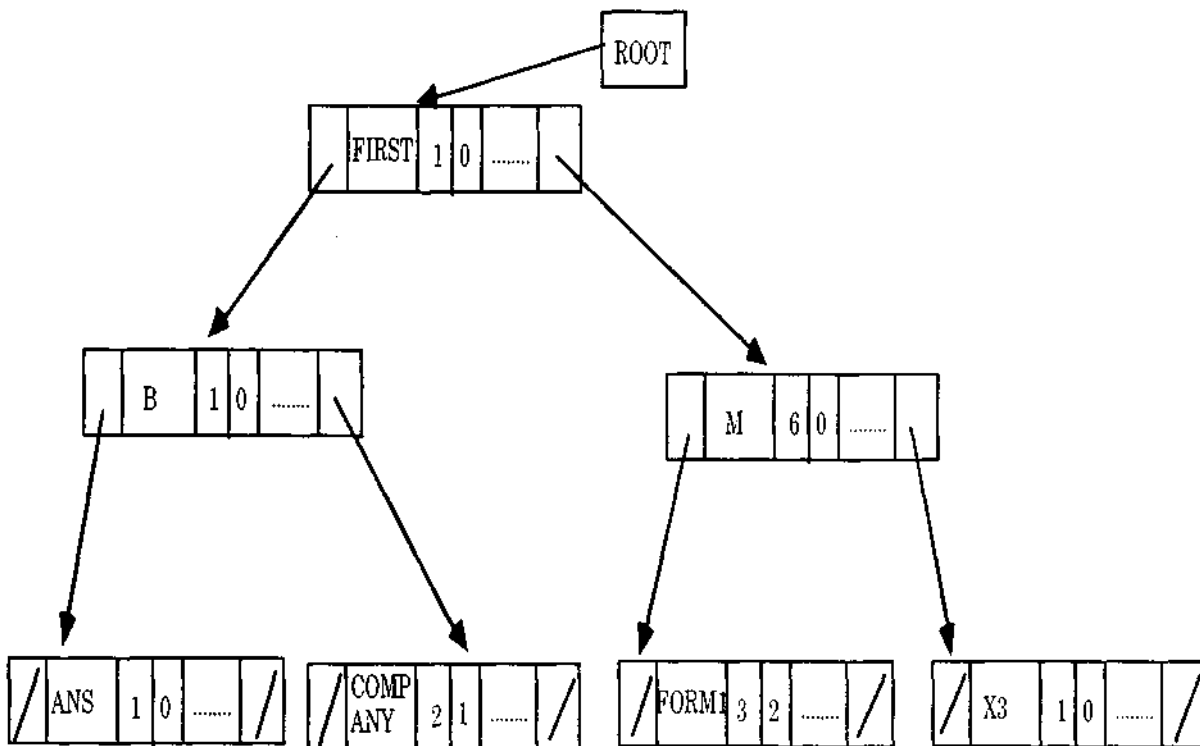
Ogni record ***(nodo)*** ha tre campi: uno per l'informazione associata al nodo, uno per il puntatore al sottoalbero sinistro ed uno per il puntatore al sottoalbero destro.

```
type tree=^node;
     node=record
              value: Element_type;
              left,right: tree
          end;
```

Vantaggi: non è necessario che l'albero sia completo; si ha un'occupazione di memoria efficiente; operare modifiche è agevole.

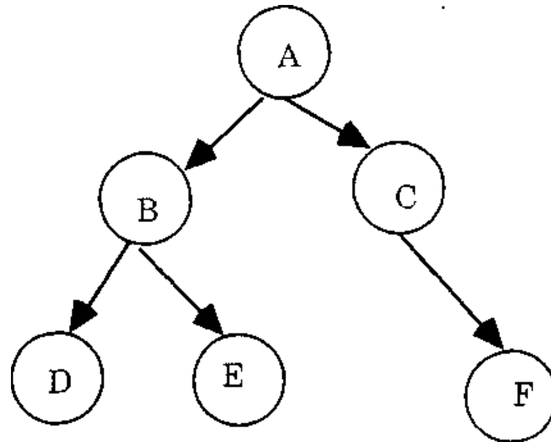
Esempio:

NAME	TYPE	DIMENSION	ATTRIB
ans	1	0	....
b	1	0	....
company	2	1	....
first	1	0	....
forml	3	2	....
m	6	0	....
X3	1	0	....



Algoritmi di visita per alberi binari:

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



Preordine o ordine anticipato:

Analizza radice, albero sinistro, albero destro.

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

A B D E C F

Postordine o ordine ritardato:

Analizza albero sinistro, albero destro, radice.

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

D E B F C A

Simmetrica:

Analizza albero sinistro, radice, albero destro.

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

D B E A C F

Gli alberi binari di ricerca:

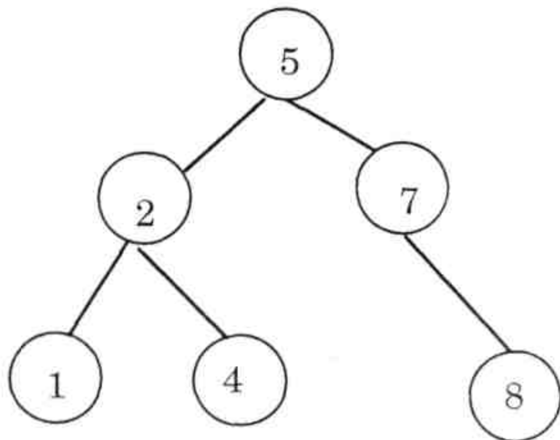
Sono alberi binari (ordinati) utilizzati per memorizzare grosse quantità di dati su cui si esegue spesso un'operazione di ricerca di un dato.

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

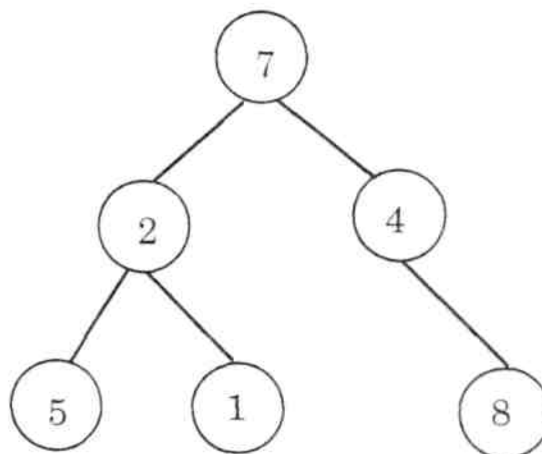
- tutti i nodi del sottoalbero sinistro di N hanno un valore minore o uguale a quello di N e
- tutti i nodi del sottoalbero destro di N hanno un valore maggiore di quello di N.

Esempio:

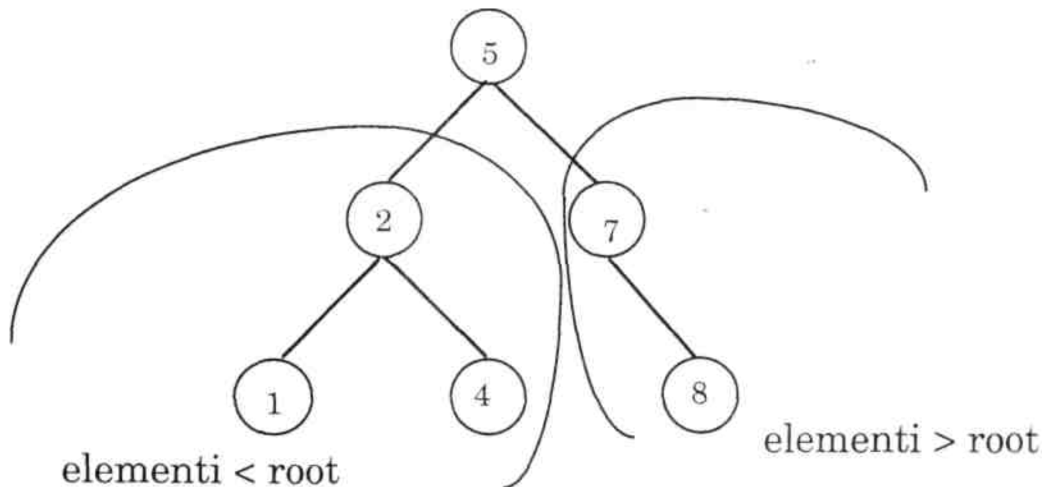
Albero binario di ricerca:



Albero binario:



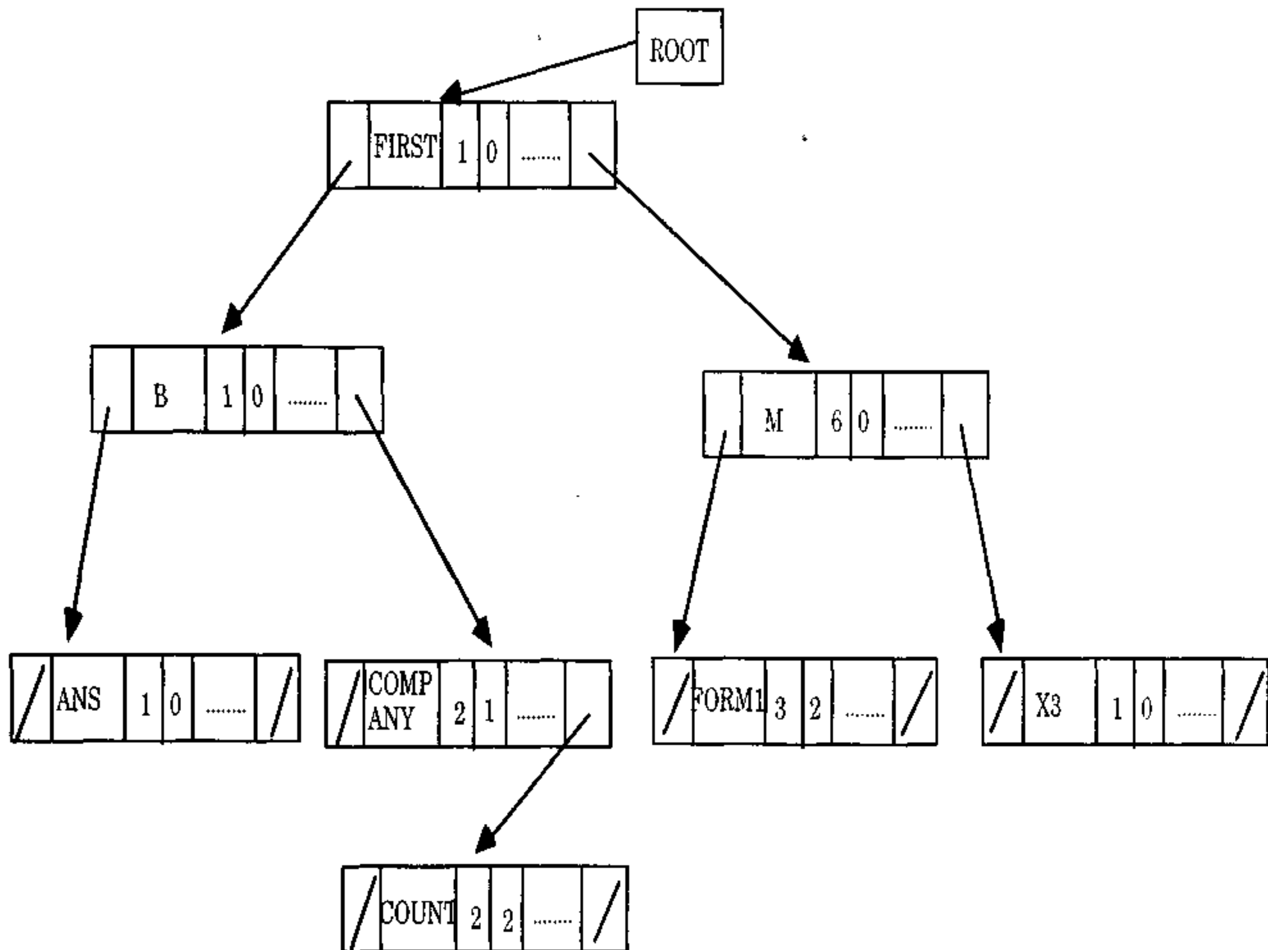
Ricerca binaria in un albero binario di ricerca:




```
function alb_ric (e:Element_type;
t:tree):boolean;
{ricerca binaria}
begin
if empty(t)
then alb_ric:=false
else
  begin
  if  isequal (e,root(T))
  then alb_ric:= true
  else
    if  isless(e,root(t))
    then alb_ric   := alb_ric(e,left(t))
    else alb_ric   := alb_ric(e,right(t))
end;
```

II numero di confronti è (nel caso peggiore) proporzionale alla profondità dell'albero. Ad ogni passo si dimezza il problema, eliminando metà dei nodi (più efficiente).

Inserimento e cancellazioni devono essere fatte in modo da mantenere l'albero bilanciato.



Alberi binari (di ricerca): inserzione con bilanciamento

Nodo perfettamente bilanciato:
l'altezza dei suoi due sottoalberi è la stessa

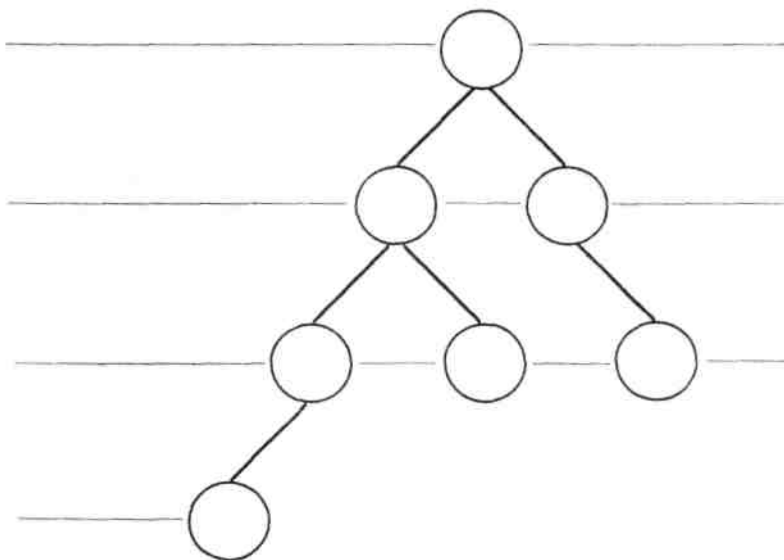
Nodo bilanciato a sinistra:

height(left(n)) = height(right(n))+1

Nodo bilanciato a destra:

height(left(n)) = height(right(n))-1

Albero bilanciato (in altezza): quando l'altezza del sottoalbero sinistro e destro di ogni nodo differisce al più di 1.

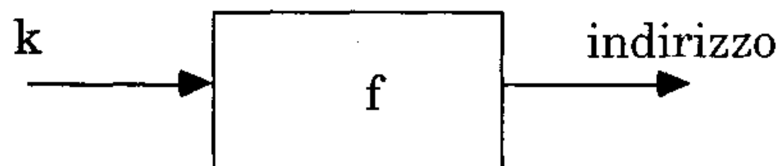


Tavole: rappresentazione a funzione di accesso

Obiettivo: permettere il recupero *con pochi accessi* di una registrazione, nota la sua chiave.

Idea-base: definire una qualche forma di *corrispondenza* fra i valori delle chiavi (primarie) e le posizioni delle registrazioni nell'archivio in modo da rintracciare velocemente la registrazione richiesta.

Ogni elemento è memorizzato in una posizione che dipende dal valore della chiave.

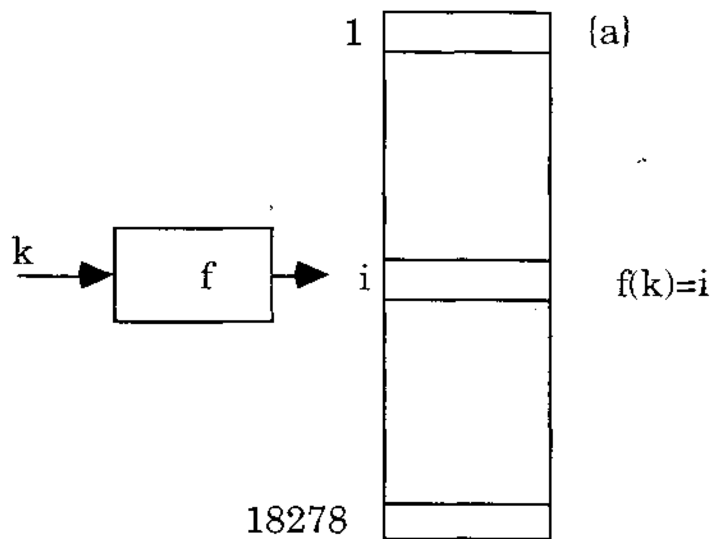


Esempio:

<key> ::= <lettera> {<lettera>}^{2}

Quindi esistono 26 chiavi di lunghezza 1, 26^2 chiavi di lunghezza 2, 26^3 chiavi di lunghezza 3.

Possibili chiavi: 26 + 676 + 17576 = 18278



Funzione di accesso biunivoca che data la chiave k restituisce il numero i corrispondente a k.

Occorre una locazione di memoria per ciascuna possibile chiave.

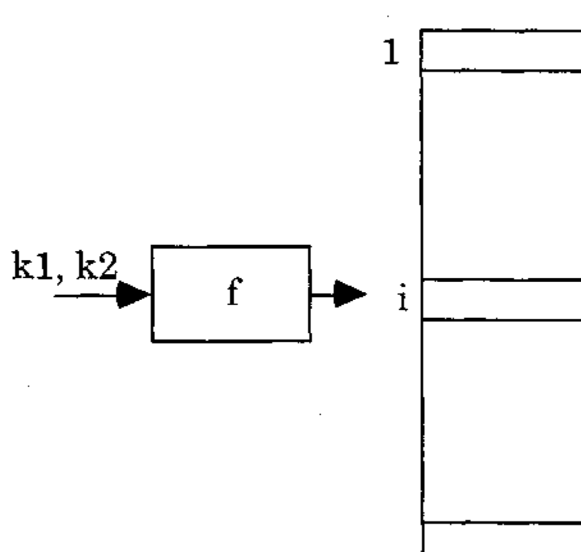
L'accesso è molto efficiente: data la chiave k, l'elemento cercato è all'indirizzo ind=f(k) nello spazio di memoria riservato alla tavola.

Anche l'***inserimento*** e la ***cancellazione*** sono realizzati in modo efficiente.

L'utilizzo di una funzione di accesso biunivoca è possibile in pratica solo se il numero di elementi della tavola è circa uguale al numero di chiavi possibili.

In alternativa, la funzione può calcolare lo stesso indirizzo per chiavi distinte: si possono verificare ***collisioni***.

f(k1)=f(k2)



In questo caso occorre:

1. scegliere una funzione di accesso che riduca il più possibile le collisioni;
2. in caso di collisione, determinare un metodo di scansione della tavola (per ricerca o inserimento).

Tabelle Hash:

Funzioni hash (tritare, mescolare), perchè l'indirizzo calcolato dipende da tutta la chiave, eventualmente spezzata in parti "rimescolate".

Ad esempio, chiave di 15 caratteri. Sia K la sua rappresentazione binaria: si calcola il resto della divisione di K per il massimo numero di elementi N previsto per la tavola.

Esempio:

Chiave: stringa di 3 lettere al massimo. Ogni chiave è associata ad un numero intero. Un intero può essere associato a più chiavi.

Le chiavi sono partizionate in ***classi di equivalenza.***

```
type key: string[3];

function hash (k: key): integer;
var first: char;
begin
    first:=k[1] ;
    hash:=(ord(first)-ord('a')) * length(k)
end;
```

Valore min: first='a' length(k)=... → 0
Valore max: first='z' length(k)=3 → 75

Classi di equivalenza: chiavi di ugual lunghezza che iniziano con la stessa lettera.

Le chiavi di una stessa classe di equivalenza ***collidono.***

Collisioni

La funzione hash si dice *perfetta* se e solo se *non produce collisioni.*

Evitare collisioni non e facile.

Si pensi che, ad esempio, già con 25 persone riunite in una stanza c'e una probabilitä superiore al 50% che due compiano gli anni nello stesso (mese e) giorno, che sale al 90% con 40 persone: e ciò è un esempio di collisione sulla funzione che mette in corrispondenza ogni persona con il suo (mese e) giorno di nascita.

Tuttavia, l'overflow può essere evitato quando si riesce a stabilire una *corrispondenza biunivoca fra l'insieme delle possibili chiavi e l'insieme delle posizioni disponibili* nell'area di memorizzazione.

In generale, però, *le chiavi possibili sono molto più di quelle realmente usate,* e *ciò rende questo approccio impossibile da adottare* perchè drammaticamente inefficiente.

Si pensi ad esempio ai cittadini che fanno parte di una data categoria: essi potrebbero ben essere numerati da 1 a N (ad es., 50.000), ma solo una minima parte farà parte di quella categoria.

Predisporre un archivio con 50.000 posti, la massima parte dei quali destinati a rimanere inutilizzati, sarebbe uno spreco inaccettabile.

Per questo si definisce il concetto di *densità delle chiavi attive.*

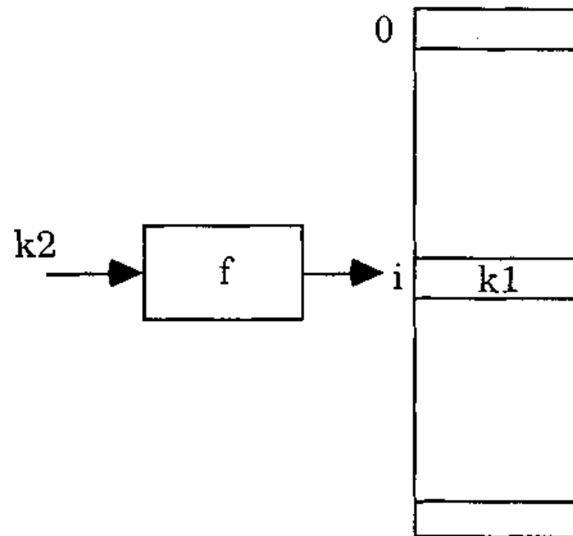
Detto N il numero di registrazioni da archiviare, e M la cardinalità dell'insieme delle chiavi, la densità delle chiavi attive è data dal rapporto N / M.

(Se le chiavi sono stringhe lunghe L definite su un alfabeto di V simboli, la cardinalità M vale V^L . Ad esempio, considerando i nomi delle persone sul normale alfabeto internazionale di 26 lettere (troncati a 15 caratteri), le possibili chiavi sono 26^{15}.)

Gestione delle collisioni

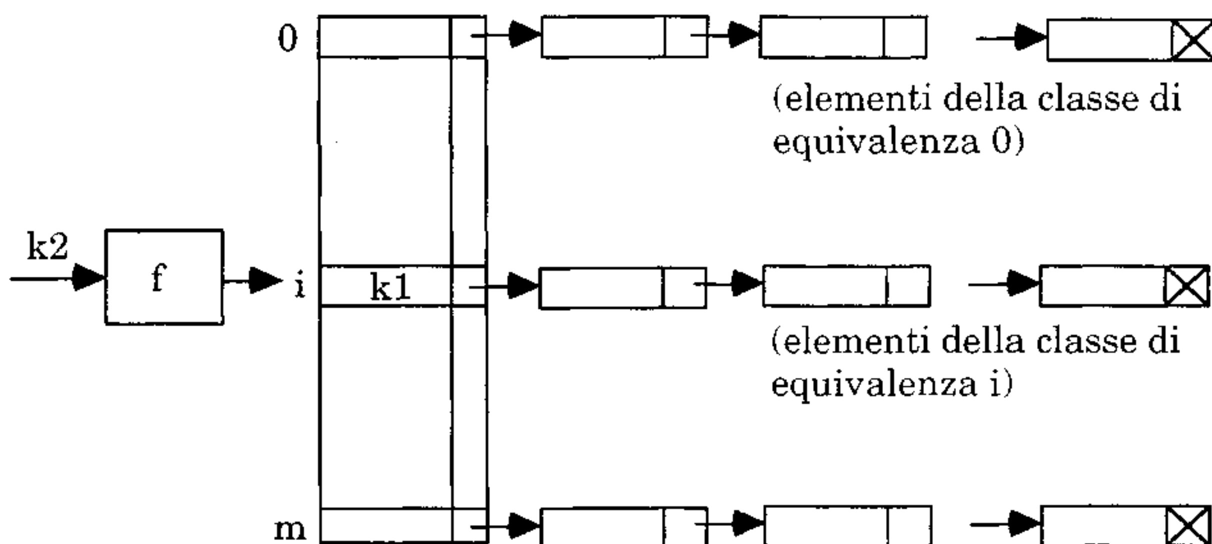
Metodi di scansione, nel caso si verifichino collisioni:

Scansione lineare:



Si passa a considerare per k2 l'indirizzo i+h (h prefissato), poi i+2*h, i+3*h, ...

In alternativa, ***scansione con aree di trabocco:***
Area di trabocco con rappresentazione collegata (lista)



Altri metodi: alberi, tabelle Hash. Può esserci anche un'unica area di trabocco per tutta la tavola.

La funzione Hash

- Una buona funzione hash deve assicurare:
  * una distribuzione *uniforme* delle chiavi nello spazio degli indirizzi
  * una distribuzione *casuale* delle chiavi (cioe tale che valori "vicini" di chiave *non* finiscano in indirizzi "vicini")

- <u>alcune funzioni hash spesso usate</u>:

  a) *metodo del modulo:*

$$H(k) = k \bmod R$$

  con R = massimo numero primo minore o uguale a P (o anche semplicemente numero non primo minore o uguale a P, purche non abbia fattori primi minori di 20).
  Esempio: se P=100, R=97; per rendere H suriettiva, si assume allora P=R=97.

  b) *metodo della somma*

$$H(k) = (pl + p2 + ...pS) \bmod 10^h$$

  dove pl...pS sono "pezzi" della chiave k stessa, che viene suddivisa in parti fatte da un egual numero *h* di caratteri (esclusa eventualmente 1'ultima), pari all'ampiezza dell'indirizzo da generare. Fatta la somma, si ignora la cifra più alta (riporto). Esempio: k=4357652, con indirizzi di *h=2* cifre. Allora: pl=43, p2=57, p3=65, p4=2, la cui somma è 167, H(k) = 67.

c) *metodo dell' EX-OR*

come la precedente, con l'operazione di EX-OR (fatta sulle corrispondenti rappresentazioni binarie) al posto della somma.

d) *metodo del quadrato*

$$H(k) = parte_centrale(k^2)$$

in pratica, si moltiplica k per se stessa, e si prendono le *h* cifre centrali del numero ottenuto.

Esempio: k=5432, h=2 , k^2 = 29506624, H(k) = 06.

Vantaggi e svantaggi dell'approccio hash:

- basso costo medio di ricerca, con elevate prestazioni, specialmente con un'organizzazione dinamica
- semplicità di realizzazione
- non si prestano a una visita dei dati in ordine diverso da quello fisico
- non sono adatte a reperire un sottoinsieme dei dati con chiave primaria che soddisfi una relazione specificata
- costo medio delle principali operazioni basso, ma costo di caso peggiore nettamente più alto

Esercizio:

Realizzare una tavola come tabella hash con area di trabocco gestita secondo una delle modalità discusse.

Linguaggi strutturati a blocchi: organizzazione della tabella dei simboli

Il linguaggio puo' contenere moduli (blocchi) innestati ciascuno contenente delle variabili locali.

Struttura a blocchi, regole di visibilita' degli identificatori e tempo di vita:

Abbiamo già detto che tutti gli identificatori devono essere dichiarati prima del loro uso.

Identificatori, nomi per indicare costanti, variabili, tipi, unità di programma definiti dall'utente.

II significato della dichiarazione è quello di associare - durante l'attivazione di un'unità di programma (main o sottoprogramma) - varie informazioni all'identificatore **(ambiente).** Ad esempio, tipo della variabile, l'indirizzo del codice eseguibile per il nome di una procedura, etc.

II tempo di vita di una di queste associazioni è la durata dell'attivazione dell'unità di programma in cui compare la dichiarazione dell'identificatore.

L'effetto di una **dichiarazione** perdura per **tutto il tempo di attivazione** dell'unità di programma in cui tale dichiarazione si trova.

Regole di visibilità degli identificatori

In Pascal, il campo di azione per gli identificatori segue le seguenti regole:

(1) il campo di azione della dichiarazione di un identificatore è il blocco (unità di programma) in cui essa compare e tutti i blocchi in esso contenuti, a meno della regola (2);

(2) quando un identificatore dichiarato in un blocco P è ridichiarato in un blocco Q, racchiuso da P, allora il blocco Q, e tutti i blocchi innestati in Q, sono esclusi dal campo di azione della dichiarazione dell'identificatore in P.

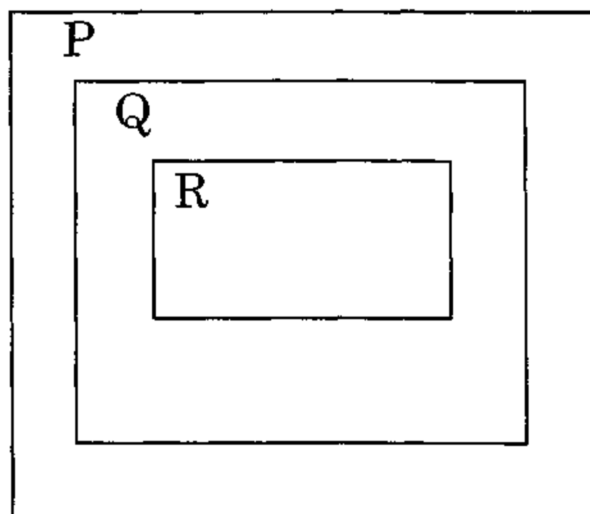
II campo di azione è determinato staticamente, dalla struttura del testo del programma **(regole di visibilità lessicali).**

```
program CampoAzione (...);
var X:integer;
        procedure P1;
        begin
           writeln(X)                    {X del main}
         end;
         procedure P2;
         var X:real;
           begin
              X:=2.14;                     {XdiP2}
              P1
           end;
begin                    {corpo di CampoAzione}
     X:=l;
     P2
end.
```

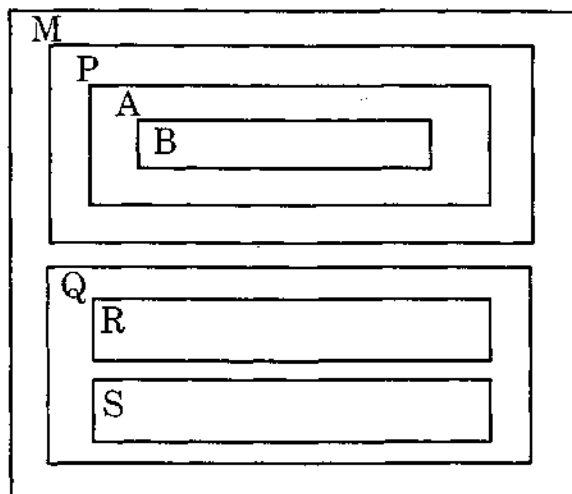
Stampa il valore 1.

Con **regole di visibilità dinamiche** (non adottate dal Pascal, ma ad esempio in LISP), stamperebbe il valore 2.14.

```
program P (input, output);
var I,J: integer;
      procedure Q;
      const I=16;
      var K: char;
            procedure R;
            var J: real;
            begin    {locali:   J :real;
                          non locali:    K: char,
                                         I: const = 16,
                                         R,Q: procedure}
            end;       {R}

      begin             {locali:    I: const = 16,
                                    K:char,
                                    R:procedure;
                          non locali:    J : integer;
                                         Q: procedure}
      end;       {Q}
begin                   {locali:   I, J: integer,
                                   Q: procedure}
end.      {P}
```



```
program M ....;
...
   procedure P;
   ...
      procedure A;
      ...
         procedure B;
         ...
         end;  {B}
      end;  {A}
   ...
   end;  {P}

   procedure Q;
   ...
      procedure R;
      ...
      end;  {R}
      procedure S;
      ...
      end;  {S}
   ...
   end;  {Q}
...
end.  {M}
```



Identificatori definiti nel blocco di:	sono accessibili in:
M	M,P,A,B,Q,R,S
P	P,A,B
A	A,B
B	B
Q	Q,R,S
R	R
S	S

a meno che non siano ridefiniti in tali blocchi.

Esempio di riferimento:

```
1)     PROGRAM MAIN
       X, Y: REAL; NAME:STRING
              PROCEDURE M2(J:INTEGER);forward

       2)     PROCEDURE M1(IND:INTEGER);
              X:INTEGER;
              BEGIN {Ml}
              CALL M2(IND+1);
              .......
              END {Ml}
       3)     PROCEDURE M2(J:INTEGER);
              4)     PROCEDURE M3;
                     F:ARRAY 1..10 OF INTEGER;
                     TEST1: BOOLEAN
                     BEGIN {M3}
                     .......
                     END {M3};
              BEGIN {M2}
              M3;
              .......
              END {M2}
       BEGIN {MAIN}
       .......
       CALL M1(X/Y);
       .......
       END. {MAIN}
```

Quando si entra all'interno di un nuovo blocco, si crea una nuova sotto-tabella (operazione di set). Quando si esce dal blocco si elimina la sotto-tabella (operazione di reset).

Traccia delle operazioni di set e reset sulla TS

Operazione	Attive	Inattive
SET BLK1	empty	empty
SET BLK2	M1,M2,NAME,X,Y	empty
RESET BLK2	M2,X,IND,M1,NAME,Y	X
SET BLK3	M2,M1,NAME,Y,X	X,IND
SET BLK4	M3,J,M2,M1,NAME,Y,X	X,IND
RESET BLK4	M3,TEST1,F,J,M2,M1,NAME,Y, X	X,IND
RESET BLK3	J,M2,M1,M3,NAME,X,Y	TEST1,F,X,IND
RESET BLK1	M2,M1,NAME,Y,X	M3,J,TEST1,F,X,IND
fine della comp.	empty	M2,M1,NAME,Y,X,M3, J,TEST1,F,X,IND

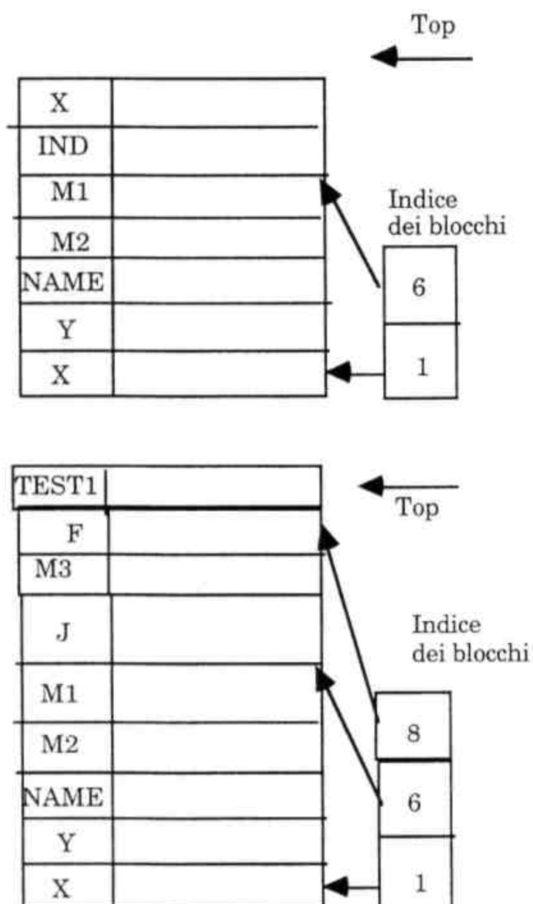
Come si puo' notare le operazioni di set e reset direttamente corrispondono a operazioni di push e pop su uno stack.

TS a Stack

E' l'organizzazione piu' semplice per un linguaggio a blocchi.

I record che contengono gli attributi delle variabili di un blocco B1 vengono messi nello stack quando si incontrano le corrispondenti dichiarazioni (push) ed eliminati al termine del blocco B1 (pop) .

Esempio: situazione prima di completare la compilazione dei blocchi 2 e 4 rispettivamente.



L'operazione di ricerca per un simbolo parte dal top dello stack, e quindi garantisce che i simboli piu' innestati siano trovati per primi (anche se ne esistessero piu' occorrenze).

L'operazione di set, salva il contenuto di top nello stack degli indici, mentre l'operazione di reset lo ripristina, cancellando implicitamente tutti i valori non piu' referenzati.

L'organizzazione si puo' rendere piu' efficiente introducendo nello stack rappresentazioni piu' sofisticate quali quelle ad albero o con funzione di accesso hash.