TABELLA DEI SIMBOLI

TABELLA DEI SIMBOLI

Il compilatore, nelle sue varie fasi, deve tenere traccia:

- Degli identificatori utilizzati nel codice sorgente;
- Dei loro attributi (tipo e caratteristiche).

Queste informazioni sono memorizzate in un'apposita struttura dati detta **Tabella dei simboli (Symbol table - ST)** che viene consultata ogni volta che si incontra un identificatore:

- Se esso non è presente nella tabella (prima occorrenza), allora si introduce come nuovo elemento con una serie di attributi;
- se esso è già presente, allora se ne aggiornano, eventualmente i suoi attributi.

QUANDO COSTRUIRE ED INTERAGIRE CON LA ST

La ST è consultata in tutte le fasi del processo di compilazione.

La costruzione della ST avviene, almeno in linea teorica, a partire dall'analisi lessicale e completata nelle fasi successive.

- **Nell'analisi lessicale**:

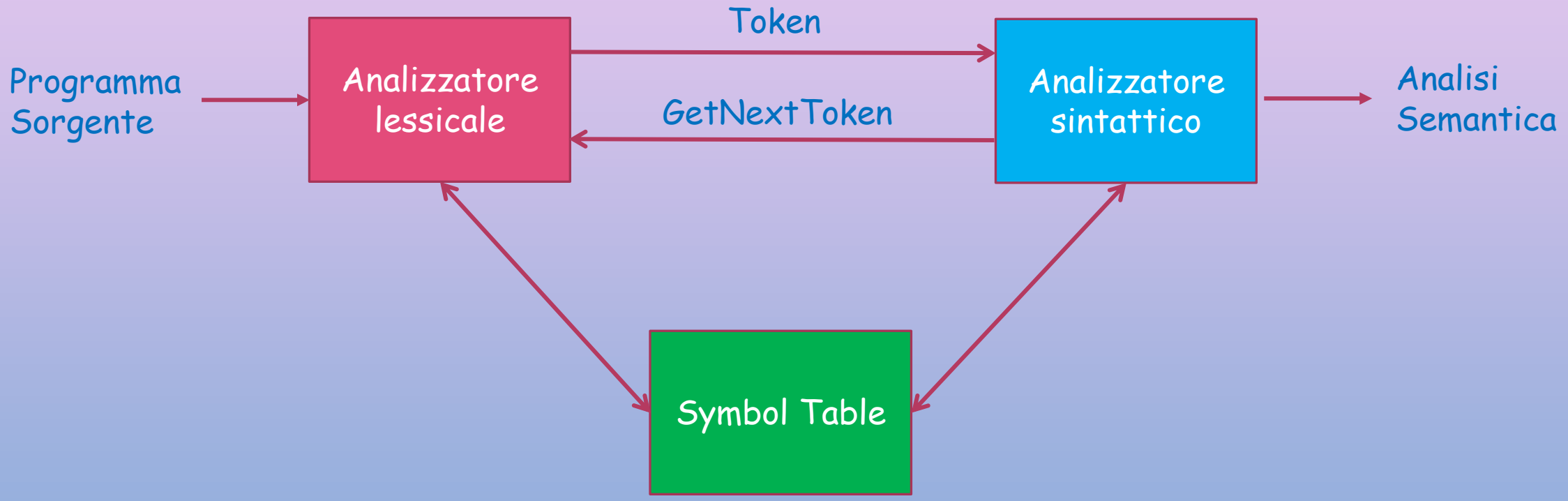
  È creata una entry nella tabella per ogni nuovo identificatore incontrato (variabili, tipi, nomi di funzioni, etichette, campi, ….); a questa entry non è associato alcun valore: di solito non si conosce né il tipo né le caratteristiche.

- **Nell'analisi sintattica e semantica**:

  La ST viene riempita con informazioni sui tipi e sulle caratteristiche degli identificatori.

  Viene coinvolta anche nella fase di **generazione del codice**

INTERAZIONI FRA SCANNER E PARSER



STRUTTURA DELLA TABELLA DEI SIMBOLI

La ST è una struttura dati astratta che serve a rappresentare insiemi di coppie

< Nome_id, lista_attributi >

Ove:

Nome_id, che di solito è una stringa, rappresenta la chiave per accedere a

Lista_attributi cioè alle informazioni associate in modo univoco all'identificatore stesso.

OPERAZIONI PRINCIPALI SULLA ST

La ST, che può essere considerata una struttura dati Dizionario, ha tre operazioni di base:

- **Insert** (inserimento)

  Per memorizzare le informazioni associate ad un identificatore.

- **Lookup** (accesso e/o ricerca)

  Per ritrovare le informazioni associate ad un determinato identificatore.

- **Delete** (cancellazione)

  Per rimuovere le informazioni associate ad un determinato identificatore quando esso non è più visibile.

PROBLEMI

Il problema principale è quello di individuare l'organizzazione più idonea della struttura dati che implementa la ST.

In genere, due sono i principali fattori che possono influenzare la scelta dell'organizzazione di una struttura dati:

- Lo spazio di memoria occupato
- La velocità di accesso

Poiché la ST normalmente è consultata migliaia di volte nel corso della compilazione, la velocità di accesso avrà maggiore priorità rispetto allo spazio di memoria occupato.

Bisogna considerare, inoltre, che anche le specifiche del linguaggio sorgente possono influenzare la scelta della struttura dati per l'organizzazione della ST.

REALIZZAZIONE DELLA TABELLA DEI SIMBOLI

Esistono diversi modi per potere implementare la ST. Anche le caratteristiche del linguaggio sorgente possono influenzare tali strutture. La ST può essere organizzata come:

- Un semplice array
- Una lista concatenata
- Un albero binario di ricerca
- Una tabella hash

LA TABELLA DEI SIMBOLI COME *ARRAY*

Vantaggi:

- Semplicità di implementazione
- Facile accesso alle sue componenti

Svantaggi:

- Dimensione della struttura da fissare a priori (ciò implica di fissare un limite sul numero di identificatori che si possono gestire)
- Ricerche non particolarmente efficienti
- La struttura ad array è raramente usata tranne che per compilatori molti semplici.

LA ST COME *LISTA CONCATENATA*

In questo caso la struttura è dinamica.

La ricerca resta lineare ma è possibile ottimizzarla:

- Mantenendo la lista ordinata
- Utilizzando il metodo della **riorganizzazione dinamica**

  Secondo questo metodo le voci a cui si accede sono spostate via via verso la testa della lista (move-to-front) cosicché i simboli più usati avranno un tempo di accesso inferiore (si presume che le istruzioni vicine nel sorgente potranno farvi riferimento).

Gestione semplice, ma inefficiente per grosse Tabelle.

LA TS COME ALBERI BINARI DI RICERCA (ABR)

Gli ABR sono particolari alberi binari tali che per ogni nodo mantengono un ordinamento ponendo nel sottoalbero di sinistra gli elementi con chiave minore e in quello di destra tutti gli elementi con chiave maggiore del nodo considerato.

Si dimostra che se l'albero è costruito in modo casuale ha un'altezza in media di 1.39log n, dove n è il numero dei nodi.

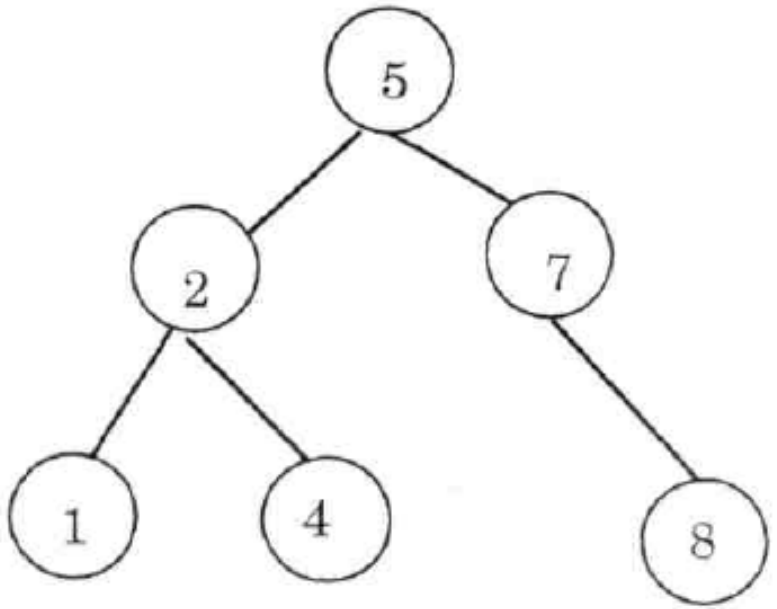
Le tre operazioni di dizionario hanno un costo proporzionale all'altezza (quindi logaritmica).

Frequenti cancellazioni possono sbilanciare l'albero.

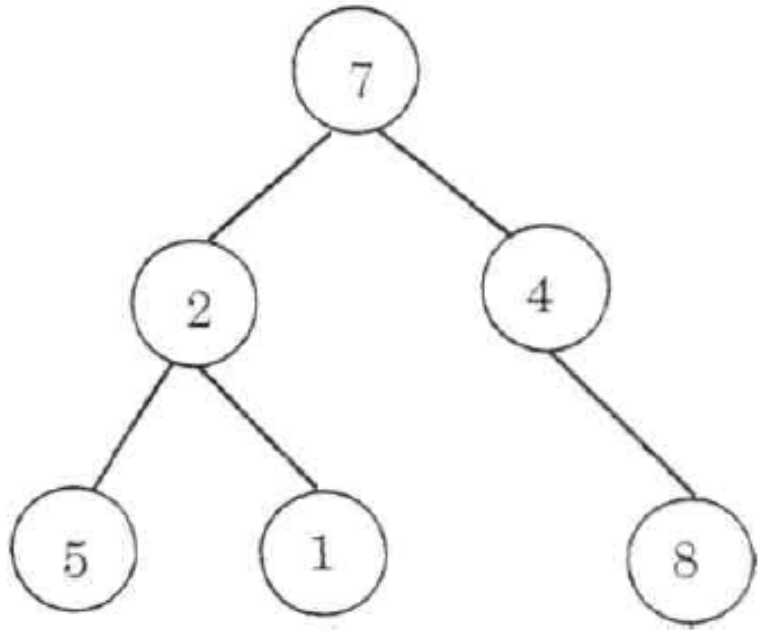
Supportano efficienti operazioni di ordinamento.

ABR

Albero binario di ricerca:



Albero binario:



LA ST AD ACCESSO DIRETTO

Si suppone che l'universo delle chiavi associate agli n elementi da memorizzare siano interi in [0..m-1] con n<=m

Assumiamo che le chiavi siano tutte distinte. Si definisce un array v di dimensione m tale che se un certo elemento x ha chiave k, allora v[k]=x

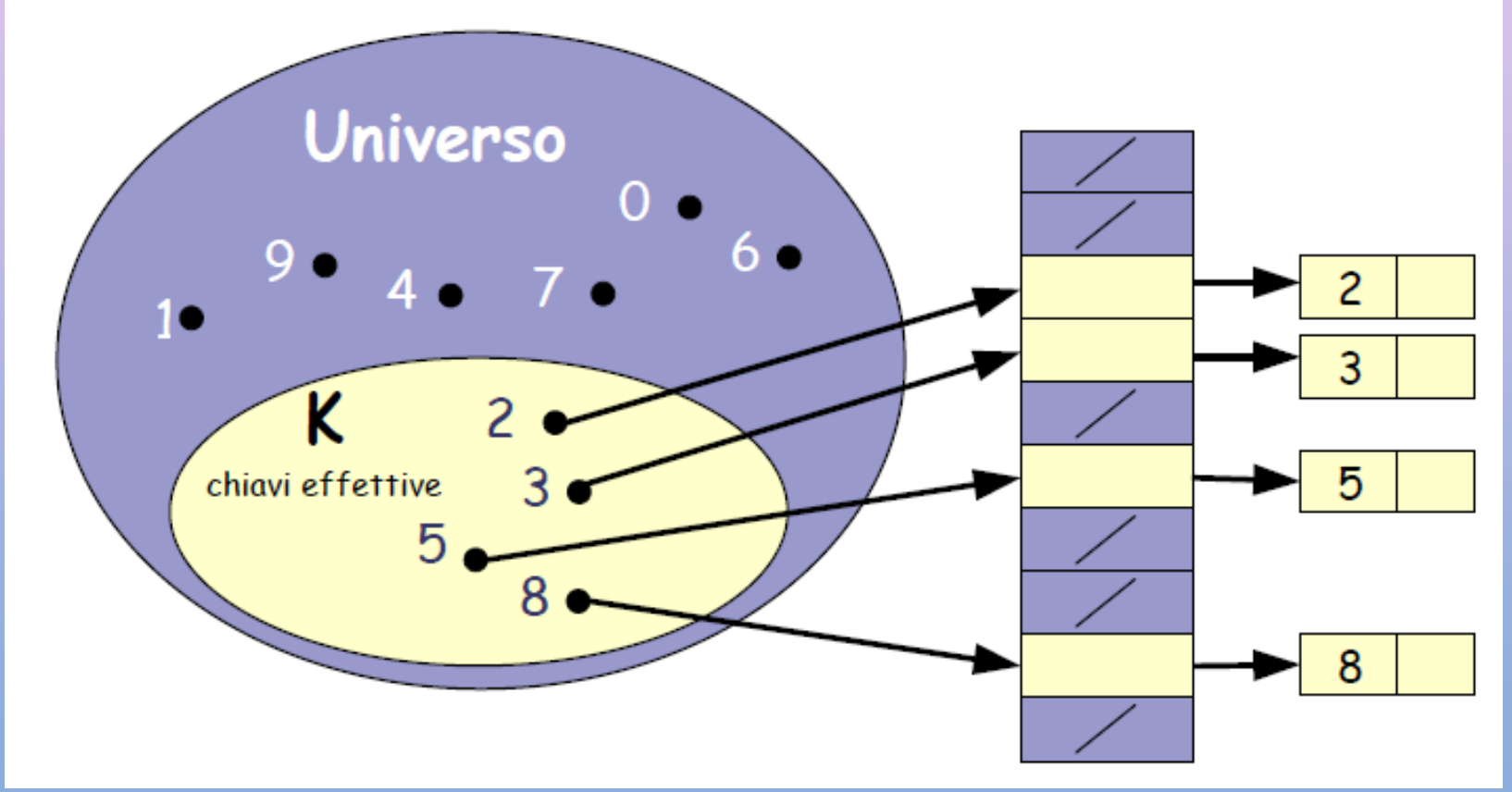
Le caselle che non corrispondono ad alcuna chiave avranno valore NULL

Le chiavi sono usate come indici per spostarci nella struttura dati

Vantaggio: Il tempo richiesto per ogni azione è costante

Svantaggio: Se m è molto grande può diventare impraticabile. Questo metodo può comportare un enorme spreco di memoria.

LA ST AD ACCESSO DIRETTO



FATTORE DI CARICO

Misuriamo il grado di riempimento di una tabella introducendo il fattore di carico:

$$A = n/m$$

Dove m è la dimensione della tabella e n è il numero di chiavi effettivamente utilizzate

Esempio:

tabella con nomi di 100 studenti indicizzati da numeri di matricola a 6 cifre

$n = 100$, $m = 10^6$, $\alpha = 0{,}0001 = 0{,}01\%$

Come si evince il fattore di carico è troppo basso. Bisogna trovare una struttura dati che permette di renderlo più alto, ossia fare in modo che non ci sia troppo spreco di memoria

Inoltre, come ci comportiamo se le chiavi non sono intere?

LA ST COME *TABELLA HASH*

Idea: dimensionare la tabella in base al numero di elementi attesi ed utilizzare una speciale funzione (**funzione hash**) per indicizzare la Tabella.

La maggior parte dei compilatori usa questo tipo di organizzazione: in particolare la ST è una hash table con liste di collisione.

Questo tipo di organizzazione consente di implementare le tre operazioni fondamentali con un numero non più costante, ma in ogni caso molto contenuto di accessi indipendente dalla dimensione della tabella, riducendo lo spazio di memoria occupato.

LA TS COME *TABELLA HASH*

Una ricerca basata su hashing è completamente diversa da quella basata su confronti: invece che spostarci nella struttura dati in funzione dell'esito dei confronti fra chiavi, cerchiamo di accedere agli elementi della tabella in modo diretto tramite operazioni aritmetiche che trasformano le chiavi in indirizzi della tabella.

E' necessario determinare la **funzione di hash** che trasforma una chiave in un indirizzo della tabella.

LA FUNZIONE *HASH*

Un indirizzo hash è un possibile valore dell'indice di un vettore di elementi destinati ad ospitare gli elementi della tabella.

La corrispondenza **nome_id – posizione** viene calcolata da un'apposita funzione detta funzione hash, ossia una funzione che trasforma chiavi in indici:

$$h : S \rightarrow \{0 \dots m-1\}$$

dove S è l'insieme delle chiavi

Una funzione hash ideale è facilmente calcolabile e approssima una funzione casuale: per ogni valore di input, i possibili valori di output dovrebbero essere in qualche senso equiprobabili.

LA TS COME TABELLA HASH

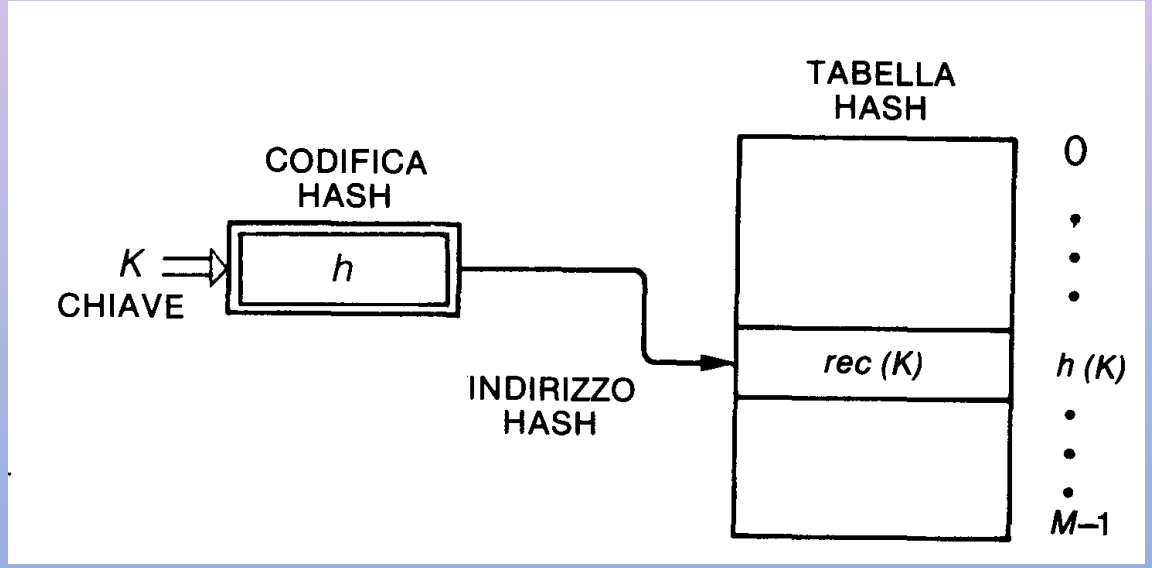
Una tabella hash è una tabella indicizzata tramite una funzione hash.

Con l'hashing, un elemento con chiave k viene memorizzato nella cella h(k), dove h è il nome della funzione hash.

Vantaggi: riduciamo lo spazio necessario per memorizzare la tabella

Svantaggi:

1. perdiamo la corrispondenza tra chiavi e posizioni in tabella
2. Le tabelle hash possono soffrire del fenomeno delle collisioni



LA TS COME TABELLA HASH

OPERAZIONI:

```
INSERT(ITEM E; KEY K);
ST[H(K)]=E
```

$T(n)=O(1)$

```
DELETE(KEY K);
ST[H(K)]=NULL;
```

$T(n)=O(1)$

```
ITEM SEARCH(KEY K);
RETURN ST[H(K)]
```

$T(n)=O(1)$

LA TS COME *TABELLA HASH*: COLLISIONI

La situazione ideale si ha quando la funzione h è iniettiva, ossia $k_1 <> k_2$ implica $h(k_1) <> h(k_2)$.

In questo caso la funzione di hash si dice perfetta: ogni chiave ha il suo indirizzo hash distinto da quello di tutte le altre chiavi: ogni elemento della tabella può essere raggiunto con un solo accesso.

Naturalmente si richiede che m sia non minore del numero delle possibili chiavi. Ciò potrebbe non sempre essere possibile, per cui si accetta il fatto che due o più chiavi possano collidere, cioè abbiano lo stesso indirizzo hash.

LA ST COME TABELLA HASH: COLLISIONI

Le chiavi possibili sono in genere moltissime e non note a priori (Per esempio le variabili che un utente inserirà in un programma, o i cognomi dei clienti di una ditta), quindi in genere si ha |insieme delle possibili chiavi| >> m.

Dunque è inevitabile che nell'applicazione si presentino due o più chiavi k_1, k_2 ... con lo stesso indirizzo hash, cioè $h(k_1)=h(k_2)=...$

Nasce quindi un **problema di collisioni**: solo una delle chiavi (tipicamente la prima che si è presentata, diciamo k_1) potrà essere allocata in $ST[h(k_1)]$, e le altre saranno poste altrove.

Si devono quindi affrontare tre problemi:

1) come si sceglie la dimensione m;
2) come si calcola la funzione h;
3) come si risolvono le collisioni.

UNIFORMITÀ SEMPLICE

Per funzionare bene una funzione hash deve distribuire uniformemente le chiavi nello spazio degli indici {0, …, m-1}, tenendo conto della loro probabilità di essere usate.

Data una chiave k, sia P(k) la probabilità per k di essere presente nel dizionario.

Sia $Q(i) = \sum_{k:h(k)=i} P(k)$ la probabilità che prendendo una chiave a caso, questa finisca nella cella i.

Una funzione hash gode della proprietà di uniformità semplice se per ciascun indice $i \in \{0, \dots, m-1\}$, Q(i)=1/m, ossia tutte le celle hanno la stessa probabilità di essere occupate e non ci sono fenomeni di agglomerazione, dove più chiavi tendono a collidere su alcune celle più che in altre

ESEMPIO Se U=[0,1] e ogni chiave $k \in U$ ha la stessa probabilità di essere scelta, $H(k)=\lfloor km \rfloor$ soddisfa la proprietà dell'uniformità semplice.

OSSERVAZIONI

1. In genere le probabilità delle chiavi non sono note a priori. In questi casi si suppone una distribuzione di probabilità uniforme P(k)=1/|U| su tutte le chiavi
2. Si può considerare come generale il caso in cui le chiavi siano interi non negativi. In caso contrario (stringhe o numeri non interi) si considera la rappresentazione binaria della chiave $y_0 y_1 \dots y_l$ e si associa alla chiave il numero

$$\sum_{i=0}^{l} y_i 2^i$$

ESEMPIO: METODO DELLA DIVISIONE

Nel metodo della divisione la funzione hash è definita come

$$h(k)=k \bmod m$$

k la chiave e m la dimensione della tabella hash.

In genere il metodo è abbastanza uniforme. In alcuni casi comunque non funziona bene, creando molte collisioni. La situazione estrema sarebbe per esempio che le chiavi fossero tutte multiple di m, per cui la funzione hash varrebbe zero per ciascuna chiave.

In genere la bontà del metodo dipende dalla scelta di m

ESEMPIO

Supponiamo di voler tenere una symbol table degli identificatori di un compilatore usando una tabella hash e il metodo della divisione

Spesso capita nei programmi che diversi identificatori hanno lo stesso prefisso, come temp1, temp2, temp3…

Se scegliamo $m=2^i$ per qualche i, tutte le chiavi con i primi i bit uguali collideranno nella stessa cella (e succederebbe per la rappresentazione binaria di temp1, temp2, temp3, se i non è abbastanza grande).

In genere una buona funzione di hash dovrebbe dipendere da tutti i bit della chiave. La scelta di m come 2^i non era buona. È preferibile prendere un m numero primo e non troppo vicino a una potenza di 2

RISOLUZIONE DELLE COLLISIONI

Per le tabelle hash utilizzate per i compilatori, la risoluzione delle collisioni avviene introducendo le **liste di collisione**

La struttura dati è costruita in maniera tale da associare ad ogni cella della hash table una lista di tutti gli elementi che collidono su quella cella.

Per verificare se una chiave è presente nel dizionario occorre calcolare la sua funzione hash e cercare la chiave nella lista corrispondente.

A lunghezza media di una lista di collisione sarà pari al fattore di carico $A=n/m$

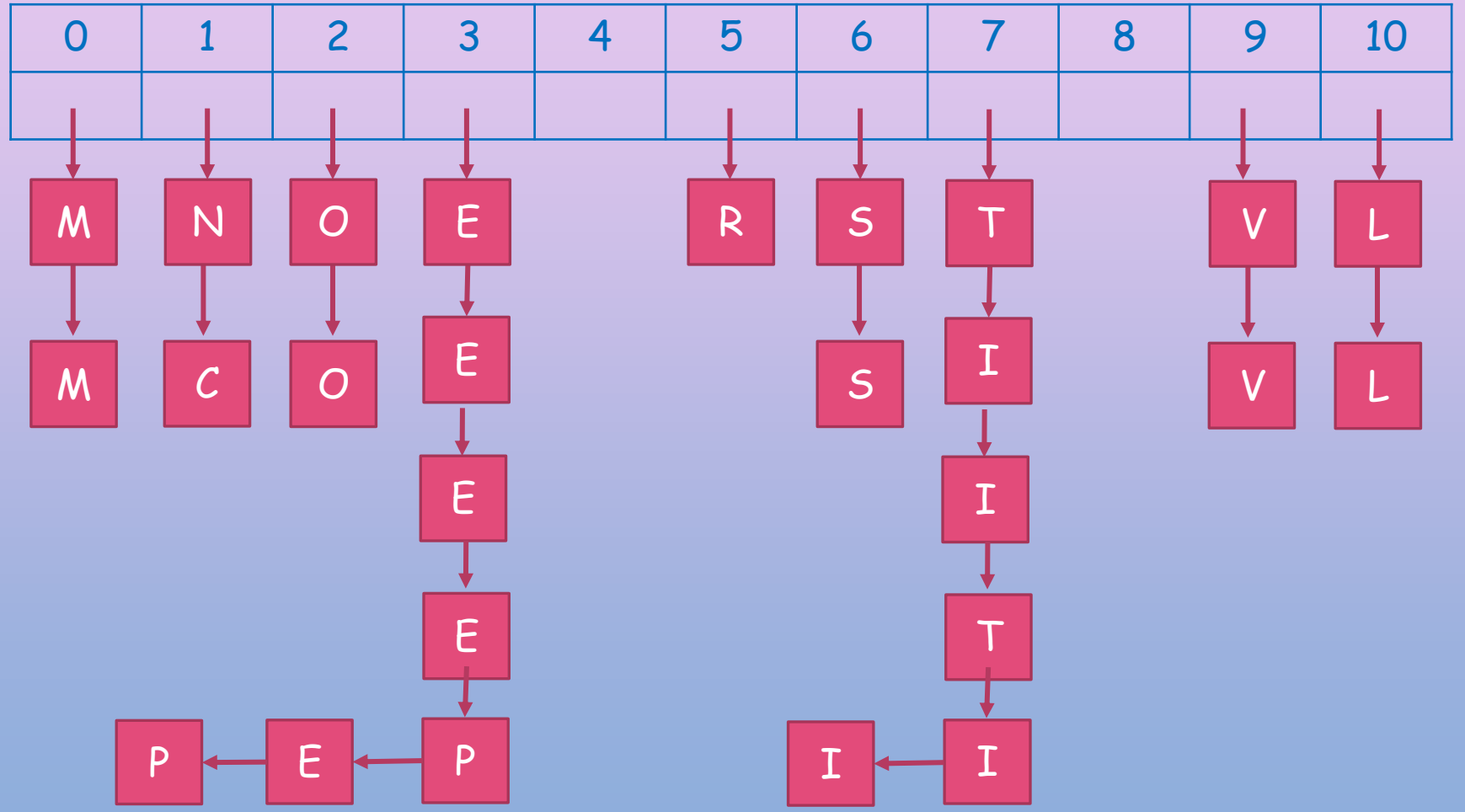
Assumendo l'uniformità semplice per la funzione hash, il tempo di ricerca sarà in media

$$T_{avg}(n,m)=O(1+n/m)$$

OSSERVAZIONI

1. Se si utilizzano le hash table con liste di collisione possiamo benissimo scegliere m<n ed avere un fattore di carico A>1
2. Con questo metodo una chiave può ricorrere più volte nel dizionario
3. Le hash table con liste di collisione sono un ottimo esempio di bilanciamento spazio-tempo. Occorre scegliere il valore di m opportuno affinchè il bilanciamento fra spazio e tempo sia accettabile.

B C D E F G H I J K L
M N O P Q R S T U V W
X Y Z A



Hash table con liste di collisione di «PRECIPITEVOLISSIMEVOLMENTE»

OSSERVAZIONI

- Alcuni compilatori scelgono di utilizzare una sola tabella dei simboli (che contiene qualunque informazione), che è dunque abbastanza complessa.
- Altri compilatori scelgono di separare le informazioni in differenti tabelle (tabella per i tipi, per la portata delle dichiarazioni, etc.)

COMPORTAMENTO E IMPLEMENTAZIONE DELLA ST

Il comportamento di una symbol table dipende pesantemente dalle proprietà delle dichiarazioni del linguaggio che deve essere tradotto;

Può essere usata una sola ST per tutti i tipi di dichiarazione (variabili, costanti, tipi, etc.), diverse ST, una per ogni tipo di dichiarazione ma esistono linguaggi (come il C e il Pascal) che hanno differenti ST per diverse regioni del programma (procedure o funzioni).

Il funzionamento delle operazioni di insert e delete, quando devono essere chiamate, quali attributi inserire, variano da linguaggio a linguaggio.

ATTRIBUTI DI VISIBILITÀ

Nel caso di variabili, un attributo implicito rappresenta la visibilità di una dichiarazione, o la regione del programma dove la dichiarazione si applica.

Un attributo collegato alla visibilità è la locazione di memoria per la variabile dichiarata.

Un altro è il tempo di vita della variabile: in C tutte le variabili esterne a funzioni sono allocate staticamente (prioritaria all'inizio dell'esecuzione). Il tempo di vita è quello del programma.

Nel caso di variabili dichiarate dentro una funzione, il tempo di vita dura quanto la funzione.

VISIBILITÀ E STRUTTURA A BLOCCHI

Nei linguaggi senza dichiarazioni annidate (Fortran) la ST così come l'abbiamo descritta è più che sufficiente.

Molti linguaggi di programmazione moderni sono strutturati a blocchi. Un blocco è un costrutto che può contenere dichiarazioni. E' possibile quindi avere anche dichiarazioni annidate. Pertanto bisogna gestire mediante la Symbol Table anche il fatto che una dichiarazione ha una sua portata (il tempo di vita).

REGOLE DI VISIBILITÀ E STRUTTURA A BLOCCHI

Dichiarazione prima dell'uso: una variabile deve essere dichiarata prima di essere usata.

Ciò consente alla symbol table di essere costruita in fase di analisi lessicale e di generare errori durante la fase di parsing.

Regola del blocco annidato più vicino: date diverse dichiarazioni con lo stesso nome la dichiarazione valida è quella del blocco più interno.

RISOLUZIONE DEL PROBLEMA

Esistono due approcci per tener conto delle regole di visibilità nella ST:

- Avere una ST per ogni ambiente o blocco;
- Avere una ST globale.

LA PILA DEGLI AMBIENTI

E' probabilmente più semplice pensare ad una ST per ogni ambiente. Infatti quando la compilazione di un determinato blocco è finita, la ST contenente l'ambiente locale, può essere cancellata.

Ciò suggerisce di utilizzare una **pila degli ambienti**: ogni volta che si definisce un nuovo ambiente una nuova ST è aggiunta nella pila.

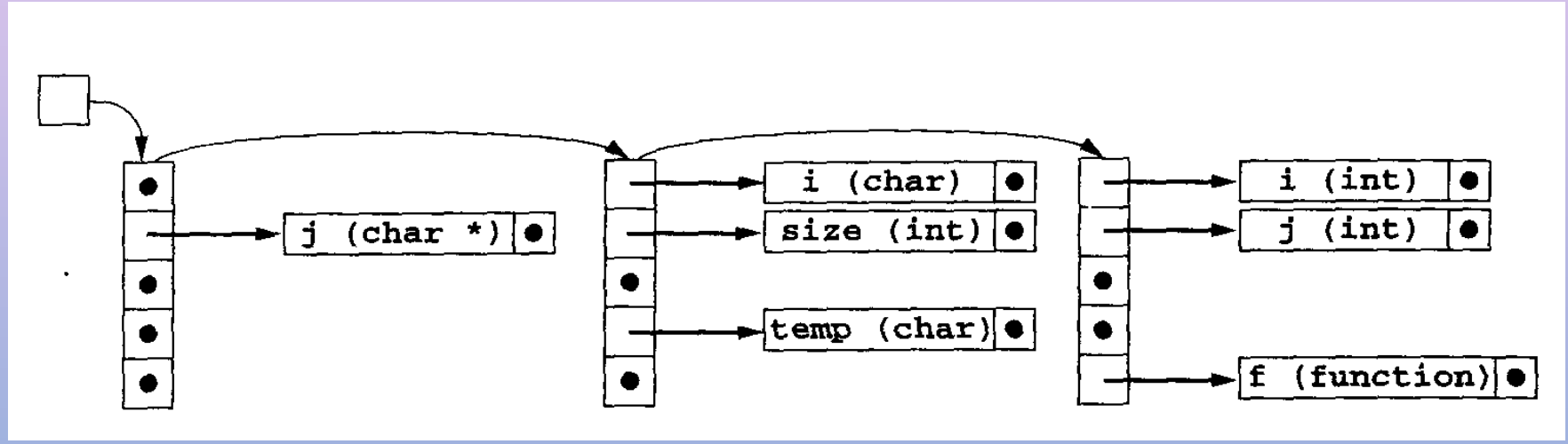
Quando il compilatore termina con tale ambiente, la ST relativa può essere eliminata.

L'uso della pila assicura che vengano rispettate le regole di visibilità.

PILA DEGLI AMBIENTI

```
int i,j;

int f(int size)
{ char i, temp;
   ...
   { char * j;
     ...
   }
}
```



PROBLEMI CON LA PILA DEGLI AMBIENTI

- Nei compilatori a più passate si potrebbe avere la necessità di rivisitare gli ambienti già cancellati;
- Alcuni compilatori creano la "cross-reference table" che contiene per ogni variabile l'informazione su dove è definita e dove è referenziata. Tale tabella è ovviamente costruita alla fine della compilazione e alcune informazioni potrebbero non essere più disponibili. La soluzione potrebbe essere quella di salvare opportunamente gli ambienti cancellati (per es. in un file temporaneo)
- Se sono presenti più ST abbiamo bisogno di più hash table normalmente vengono previste tutte della stessa dimensione: può capitare, quindi, che per alcuni ambienti con pochi identificatori lo spazio di memoria inutilizzato sia veramente tanto.

UNA ST GLOBALE

In alternativa si può costruire una ST globale. In questo caso bisogna associare all'identificatore un'informazione relativa all'ambiente a cui appartiene.

Ciò può essere fatto associando ad ogni ambiente un numero che verrà incrementato ogni volta che inizia un nuovo ambiente.

Ogni nuovo identificatore sarà inserito all'inizio della lista in modo tale che in cima ci saranno gli identificatori dell'ultimo ambiente analizzato e ciò in accordo con le regole di visibilità.

La cancellazione degli ambienti già analizzati non presenta particolari difficoltà, fermo restando le osservazioni fatte in precedenza.

NOME DELL'IDENTIFICATORE

Per ogni elemento della ST occorre memorizzare:

- Nome dell'identificatore
- Elenco degli attributi

In alcuni linguaggi di programmazione la lunghezza degli identificatori è limitata.

In questo caso è possibile memorizzare il nome dell'identificatore in un vettore di caratteri di lunghezza massima pari a quella consentita per la lunghezza degli identificatori stessi.

Lo spreco di memoria risulta irrisorio se il limite per la lunghezza degli identificatori è ragionevole (per es. 8 caratteri).

Tale soluzione non può essere accettata per quei linguaggi che ammettono identificatori di lunghezza arbitraria.

LA SOLUZIONE ALTERNATIVA

I nomi degli identificatori vengono memorizzati, opportunamente delimitati, in un unico vettore di caratteri di grandi dimensione (detto "**array dei nomi**").

Nella ST non sarà presente il nome dell'identificatore ma un puntatore all'array dei nomi.