Symbols Table

Introduction

A compiler needs to collect and use information about the names appearing in the source program. This information is entered into a data structure called a symbol table. The information collected about a name includes:-

1- The string of characters by which it is denoted.
2- It's type (e.g. integer, real, string).
3- It's form (e.g. a simple variable, a structure).
4- It's location in memory.
5- Other attributes depending on the language.

Each entry in the symbol table is a pair of the form (name, information). Each time a name is encountered, the symbol table is searched to see whether that name has been seen previously. If the name is new, it is entered into the table. Information about that name is entered into the table during lexical and syntactic analysis.

The information collected in the symbol table is used during several stages in the compilation process. It is used in semantic analysis, that is, in checking that uses of names are consistent with their implicit or explicit declarations. It is also used during code generation. Then we need to know how much and what kind of run-time storage must be allocated to a name.

There are also a number of ways in which the symbol table can be used to aid in error detection and correction. For example, we can record whether an error message such as "variable A undefined" has been printed out before, and reject from doing so more than once. Additionally, space in the symbol table can be used for code-optimization purposes, such as to flag temporaries that are used more than once.

The primary issues in symbol table design are the format of the entries, the method of access, and the place where they are stored (primary or secondary storage). Block-structured languages impose another problem in that the same identifier can be used to represent distinct names with nested scopes. In compilers for such languages, the symbol table mechanism must make sure that the inner most occurrence of an identifier is always found first, and that names are removed from the active portion of the symbol table when they are no longer active.

The Contents of a Symbol Table

A simple table is a table with two fields, a name field and information field. We require several capabilities of the symbol table. We need to be able to:-

1- Determine whether a given name is in the table.
2- Add a new name to the table.
3- Access the information, associated with a given name.
4- Add new information for a given name.
5- Delete a name or group of names from the table.

In a compiler, the names in the symbol table denote objects of various sorts. There may be separate tables for variable names, labels, procedure names, constants, and other types of names depending on the language. Depending on how lexical analysis is performed, it may be useful to enter keywords into the symbol table initially. If the languages does not reserve keywords (forbid the use of keywords as identifiers), then it is essential that keywords be entered into the symbol table and that they have associated information warning of their possible use as a keyword.

Basic Implementation Techniques

The first consideration of symbol table implementation is how entering and find, store and search for names. Depending on the number of names we wish to accommodate and the performance we desire, a wide variety of implementations is possible:-

- **Unordered List**

Use of an unordered list is the simplest possible storage mechanism. The only data structure required is an **array**, with insertions being performed by adding new names in the next available location. Of course, a **linked list** may be used to avoid the limitations imposed by a fixed array size. Searching is simple using an iterative searching algorithm, but it is impractically slow except for very small tables (no more than 20 items).

- **Ordered List**

If a list of names in an **array** is kept ordered, it may be searched using **binary searched**, which requires O (log (n)) time for a list of n entries. However, each new entry must be inserted in the array in the appropriate location. Insertion in an ordered array is relatively expensive operation. Thus ordered lists are typically used only when the entire set of names in a table is known in advance. They are useful therefore for tables of reserved words.

Binary Search Trees

Binary search trees are a data structure designed to combine the size flexibility and insertion efficiency of a linked data structure with the search speed provided by a binary search. On average, entering or searching for a name in a binary search tree built from random inputs requires O(log(n))time. One compelling argument in favor of binary search trees is their simple, widely known implementation. This implementation simplicity and the common perception of good average case performance make binary search trees a popular technique for implementing symbol tables.

Hash Tables

Hash tables are probably the most common means of implementing symbol tables in production compilers and other system software. With a large enough table, a good hash function, and the appropriate collision-handling technique, Searching can be done in essentially constant time regardless of the number of entries in the table.

-Binary Search Trees

The algorithm for implementing a simple **binary search trees** can be found in Fig. 33.

```
(1) While P ≠ nil do
(2)     If NAME = NAME (P) then      /* Name found, take action on success */
(3)     Else   if NAME< NAME (P) then P:= LEFT (P)      /* visit left child  */
(4)     Else   /* NAME (P) < NAME */ P:= RIGHT (P)      /* visit right child*/

     /* if we fall through the loop, we have failed to find NAME */
```

Fig. 33: binary search tree routine

Of greater concern is how acceptable performance can be ensured using a binary search tree symbol table implementation. If a binary tree is perfectly balanced, the expected search time is O(log(n)). A tree built from random inputs also has an expected search time that is proportional to the log of the number of items in the tree. However, the worst case performance is O (n) and actual occurrence of this worst case is not improbable. For example, entering names in alphabetic order (A, B, C, D, E) results in a tree that is really a linear list, and even random-looking sequences of names can produce the same result (A, E, B, D, C, for example).

This problem can be overcome by using an insertion algorithm that keeps the tree approximately balanced. Tree-balancing algorithms are based on the idea of

keeping the height of each sub tree rooted at a node within 1 of the height of its sibling subtree. Entire sub tree are moved to different root node when an insertion would unbalanced a node. The fact that rebalancing can be done by moving sub trees rather than individual nodes keep the insertion cost at O(log(n)).

One significant advantage of binary trees for implementing symbol tables is that their space overhead, for storing the pointers that define the tree, is directly proportional to the number of nodes in the tree. In contrast, hash tables has have a fixed space overhead, storage for the hash table itself, regardless of the number of names that have been entered. One implementation technique is to use many symbol tables to represent various program, components, rather than using one global table.

Hash Table

The central idea of a hash table is to map each of a large space of possible names that might be entered into a symbol table to one of a fixed number of positions in a hash-table. This mapping is done by a hash function.

A hash function is normally assumed to have the following properties:-

1- h(n) depends on n.
2- h can be computed quickly
3- h is uniform and randomizing in mapping names to hash addresses. That is all hash addresses are mapped with equal probability, and similar names don't cluster to the same hash addresses.

Some hash functions treat a name as sequence of words, with some number of characters per word. Names longer than one word are folded together into one word, usually by exclusive operations or by multiplying together two n bit words and keeping the middle n bits of the product. The hash value is then obtained by taking the remainder modulo m, where the hash table has m entries. Note that if **m** is equal to $\mathbf{2^b}$, this division simply isolates the rightmost b bits. Thus, such table sizes should be avoided.

An alternative is to compute a hash value character by character, as a token is scanned. Simple hash function include $(c_1 + c_2 + \ldots + c_n) \bmod m$ or $(c_1 * c_2 * c_3 * \ldots * c_n) \bmod m$, where the token is composed of characters $c_1, c_2, \ldots, c_n$, though care must be taken to avoid or handle overflows in doing such computations.

The Fig.34 below show a hash table.

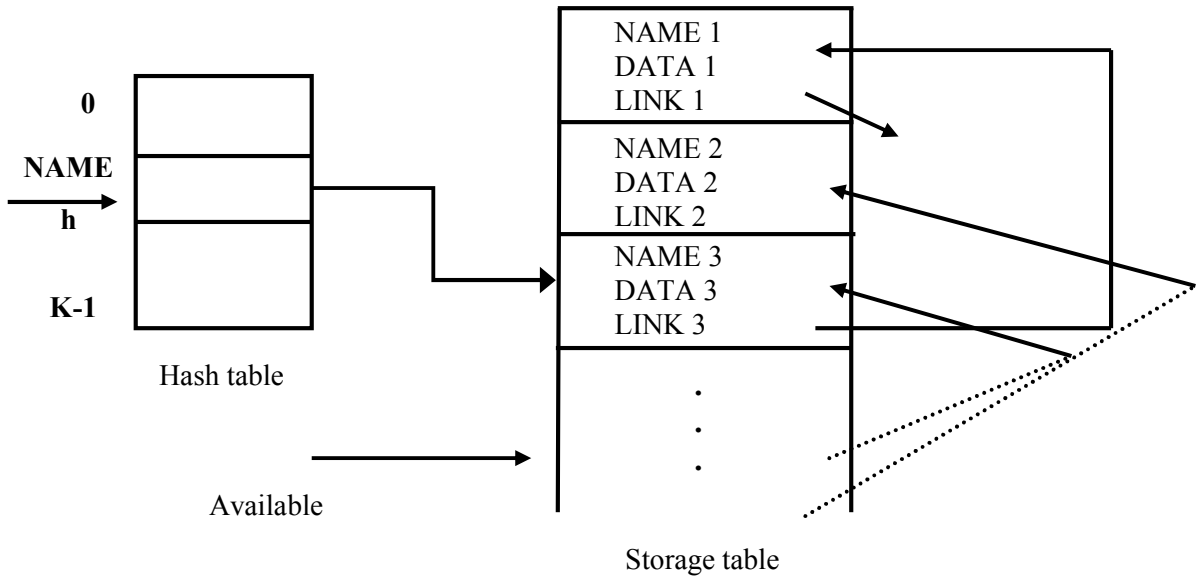


Fig.34: Hash Table

Fig.34 shows two tables, a hash table and a storage table. The hash table consist of k words, numbered 0, 1 … k-1. These words are pointers into the storage table to the heads of k separate linked lists (some lists may be empty). Each record in the symbol table appears on exactly one of these lists

To determine whether NAME is in the symbol table, we apply to NAME a hash function **h** such that h(NAME) is an integer between 0 and k-1. It is on the list numbered h(NAME) that the record for NAME belongs. To inquire about NAME, we compute h(NAME) and search that list only. To enter NAME into the symbol table, we create a record for it at the first available place in the storage table and link that record to the beginning of the h(NAME)'th list. Since the average list is n/k records long if there are n names in the table, we have cut our searching work down by a factor of **k**. As **k** can as large as we like, we can choose **k** sufficiently large that **n/k** will be small for even very large programs.

Resolving Collisions

Because the number of the possible names that can be entered into a symbol table is usually much larger than the number of hash addresses, collisions can occur. That is for names n1 and n2 ($n1 \neq n2$) but h(n1) =h(n2), when such a collisions occurs, a number of collisions-handling techniques are possible :-

1- Linear Resolution

If position h(n) is occupied, try **(h(n)+1)** mod **m** , **(h(n)+2)** mod **m** , and so on. If any table positions are free, they will be found eventually. The main problem with this technique is that as the table fills, long chains tend to form.

2- Add-the-Hash Rehash

If h(n) is occupied, try **(2*h(n))** mod **m**, **(3*h(n))** mod **m** and so on. This helps prevent long chains, but **m** must be prime if all hash positions are to be eventually tried.

3- Quadratic Rehash

If h(n) is occupied, try **(h(n)+1^2)** mod **m**, **(h(n)+2^2)** mod **m**, and so on.

4- Collision Resolution by Chaining

Names are not placed in the hash table at all, but rather records for all names that hash to a given value, are chained together on a linked list. Only list headers are stored in the hash table itself as shown in Fig.35.

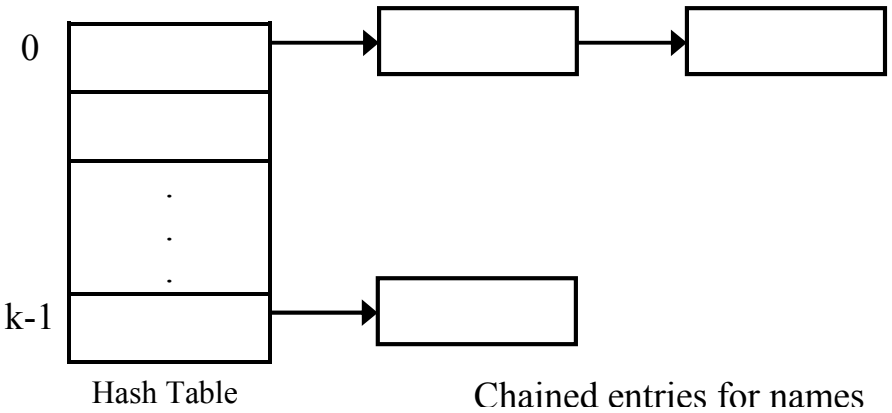
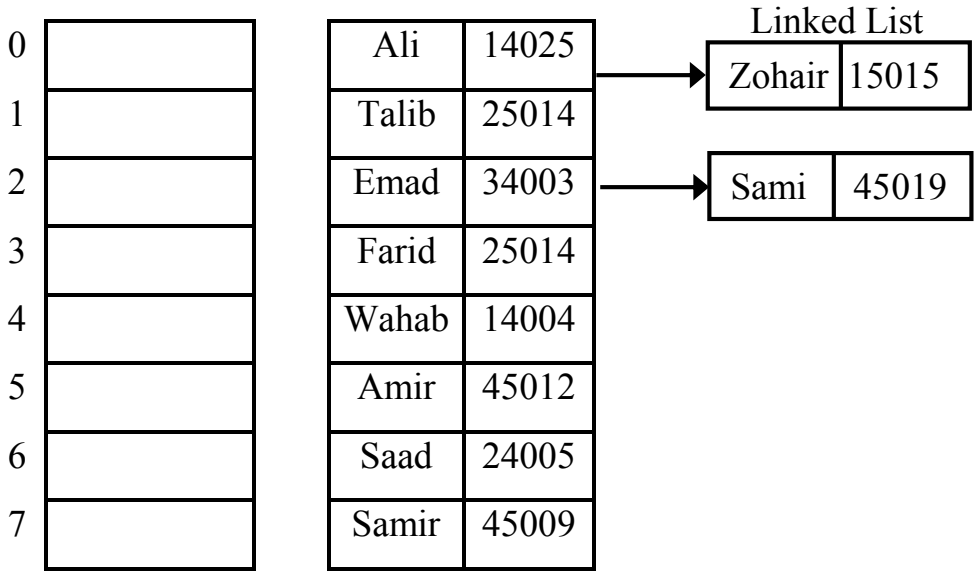


Fig.35: Hash tables with chaining for collision Resolution



Hash Table (b)

Symbol	Value	Hash Code
Ali	14025	0
Amir	45012	5
Emad	34004	2
Sami	45019	2
Samir	45009	7
Zohair	15015	0
Talib	25014	1
farid	25014	3
Wahab	14004	4
Saad	24005	6

(a)

Fig.36: Hash Coding

(a) Symbols, values & hash code derived from symbols.

(b) Eight entry hash table with linked lists of symbols & values.

Examples hash function:

Use Hash table of length (12) to save the following tokens: (ett, tva, tre, fem and fyra). Suppose the hash function is the sum of index of each character in a token. Use Linear Resolution (if necessary).

We simply store it a position $h(x)$-1 of the array

ett = 1+2+3 = 6 mod 12 = 6
tva = 1+2+3 = 6 mod 12 = 6 (collision, 6+1 mod 12 = 7)
tre= 1+2+3 = 6 mod 12 = 6 (collision, 6+1 mod 12 = 7 collision , 6+2 mod 12 = 8)
fem = 1+2+3 = 6 mod 12 = 6 (collision, 6+1 mod 12 = 7 collision , 6+2 mod 12 = 8 collision , 6+3 mod 12 = 9)
fyra = 1+2+3+4= 10 mod 12 = 10

x	h(x)
	0
	1
	2
	3
	4
	5
ett	6
tva	7
tre	8
fem	9
fyra	10
	11

Use Hash table of length (12) to save the following tokens: (sju, atta, nio, tio, elva, and tolv). Suppose the hash function is the sum of index of each character in a token. Use Linear Resolution (if necessary).

We simply store it a position $h(x)$-1 of the array

sju = 1+2+3 = 6 mod 12 = **6**
atta = 1+2+3+4 = 10 mod 12 = **10**
nio = 1+2+3 = 6 mod 12 = 6 (collision, 6+1 mod 12 = **7**)
tio = 1+2+3 = 6 mod 12 = 6 (collision, 6+1 mod 12 = 7 collision , 6+2 mod 12 = **8**)
elva = 1+2+3+4= 10 mod 12 = 10 collision, 10+1 mod 12 = **11**
tolv= 1+2+3+4= 10 mod 12 = 10 collision, 10+1 mod 12 = 11 collision , 10+2 mod 12 = **0**

x	h(x)
tolv	0
	1
	2
	3
	4
	5
sju	6
nio	7
tio	8
	9
atta	10
elva	11

Use Hash table of length (4) to save the following tokens: (Genetic, Image, Token, and Neural). Suppose the hash function is the sum of index of each character in a token. Use Add-the hash-rehash (if necessary).

We simply store it a position $h(x)$-1 of the array

Genetic = 1+2+3+4+5+6+7 = 28 mod 4 = **0**
Image = 1+2+3+4+5 = 15 mod 4 = **3**
Token = 1+2+3+4+5 = 15 mod 4 = 3 (collision, 2*3 mod 4 = **2**)
Neural = 1+2+3+4+5+6 = 21 mod 4 = **1**

x	*h*(*x*)
Genetic	0
Neural	1
Token	2
Image	3