**Symbol Table Management**

A symbol table is a data structure used by a compiler to keep track of scope binding information about names.

This information is used in the source program to identify the various program elements, like variables, constants, procedures, and the labels of statements.

The symbol table is searched every time a name is encountered in the source text. When a new name or new information about an existing name is discovered, the content of the symbol table changes. Therefore, a symbol table must have an efficient mechanism for accessing the information held in the table as well as for adding new entries to the symbol table

What information is stored in the symbol table?

What items to enter in the symbol table?

* Variable names; defined constants; procedure and function names; literal constants and strings; source text labels; compiler-generated temporaries.

What kind of information might the compiler need about each item:

* textual name, data type, declaring procedure, storage information. Depending on the type of the object, the compiler may want to know list of fields (for structures), number of parameters and types (for functions), etc…

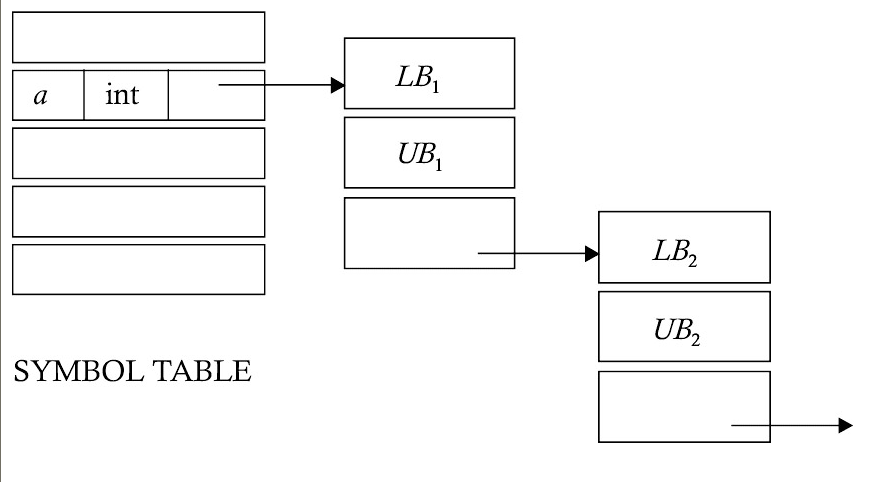
In practice, many different tables may exist.

*Symbol table information is accessed frequently:*

*hence, efficiency of access is critical!*

**IMPLEMENTATION**

Each entry in a symbol table can be implemented as a record that consists of several fields. These fields are dependent on the information to be saved about the name. But since the information about a name depends on the usage of the name**,** the entries in the symbol table records will not be uniform. Hence, to keep the symbol table records uniform, some of the information about the name is kept outside of the symbol table record, and a pointer to this information is stored in the symbol table record, as shown in the following [Figure](mk:@MSITStore:E:\computer\compiler\Algorithms%20for%20Compiler%20Design.chm::/7264final/LiB0048.html#ch07fig01#ch07fig01).

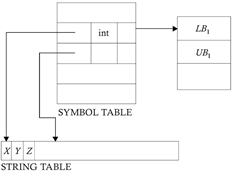


**Entering Information Into The Symbol Table**

Information is entered into the symbol table in various ways. In some cases, the symbol table record is created by the lexical analyzer as soon as the name is encountered in the input, and the attributes of the name are entered when the declarations are processed.

**Where Should Names Be Held?**

If there is a modest upper bound on the length of the name, then the name can be stored in the symbol table record itself. But if there is no such limit, or if the limit is rarely reached, then an indirect scheme of storing name is used. A separate array of characters, called a "string table," is used to store the name, and a pointer to the name is kept in the symbol table record, as shown in the following figure :

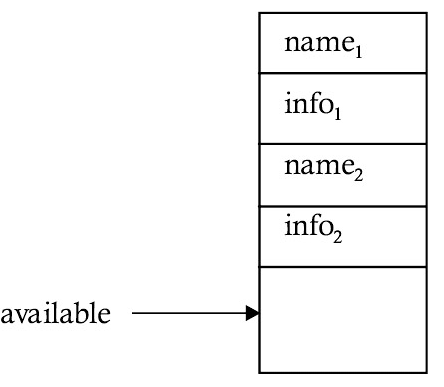


**Various Approaches To Symbol Table Organization**

There are several methods of organizing the symbol table. These methods are discussed below:

**The Linear List**

A linear list of records is the easiest way to implement a symbol table. The new names are added to the table in the order that they arrive. Whenever a new name is to be added to the table, the table is first searched linearly or sequentially to check whether or not the name is already present in the table. If the name is not present, then the record for new name is created and added to the list at a position specified by the available pointer, as shown in the following figure :

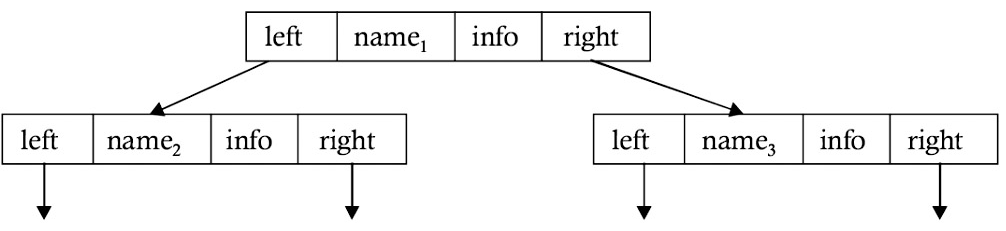


To retrieve the information about the name, the table is searched sequentially, starting from the first record in the table. The average number of comparisons, p, required for search are p = (n + 1)/2 for successful search and p = n for an unsuccessful search, where n is the number of records in symbol table. The advantage of this organization is that it takes less space, and additions to the table are simple. This method's disadvantage is that it has a higher accessing time.

**Search Trees**

A search tree is a more efficient approach to symbol table organization. We add two links, left and right, in each record, and these links point to the record in the search tree. Whenever a name is to be added, first the name is searched in the tree. If it does not exist, then a record for the new name is created and added at the proper position in the search tree.

The expected time needed to enter n names and to make m queries is proportional to (m + n) log2n; so for greater numbers of records (higher n) this method has advantages over linear list organization .



* + An unbalanced tree would have similar behaviour as a linear list (this could arise if symbols are entered in sorted order).
  + A balanced tree (path length is roughly equal to all its leaves) would take *O(log2n)* probes per lookup (worst-case).

**Hash Tables**

A hash table is a table of k pointers numbered from zero to k−1 that point to the symbol table and a record within the symbol table. To enter a name into symbol table, we find out the hash value of the name by applying a suitable hash function.

The hash function maps the name into an integer between zero and k−1, and using this value as an index in the hash table, we search the list of the symbol table records that is built on that hash index. If the name is not present in that list, we create a record for name and insert it at the head of the list.

When retrieving the information associated with the name, the hash value of the name is first obtained, and then the list that was built on this hash value is searched for information about the name

* + Potentially *O(1),* but needs inexpensive function, with good mapping properties, and a policy to handle cases when several names map to the same single index.

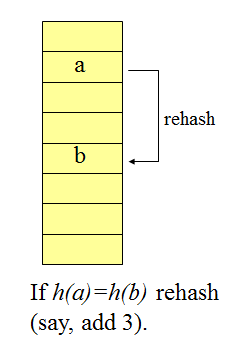
Bucket hashing (open hashing)

* A hash table consisting of a fixed array of *m* pointers to table entries.
* Table entries are organised as separate linked lists called buckets.
* Use the hash function to obtain an integer from *0* to *m-1*.
* As long as *h* distributes names fairly uniformly (and the number of names is within a small constant factor of the number of buckets), bucket hashing behaves reasonably well.

foo...

qq...

i...

****

* Use a single large table to hold records. When a collision is encountered, use a simple technique (i.e., add a constant) to compute subsequent indices into the table until an empty slot is found or the table is full. If the constant is relatively prime to the table size, this, eventually, will check every slot in the table.
* Disadvantages: too many collisions may degrade performance. Expanding the table may not be straightforward.