How are Symbol Tables Implemented?

There are a number of data structures that can reasonably be used to implement a symbol table:

- An Ordered List
  Symbols are stored in a linked list, sorted by the symbol's name. This is simple, but may be a bit too slow if many identifiers appear in a scope.
- A Binary Search Tree
  Lookup is much faster than in a linked list, but rebalancing may be needed. (Entering identifiers in sorted order can turn a search tree into a linked list.)

- Hash Tables
  The most popular choice.

Implementing Block-Structured Symbol Tables

To implement a block structured symbol table we need to be able to efficiently open and close individual scopes, and limit insertion to the innermost current scope.

This can be done using one symbol table structure if we tag individual entries with a “scope number.”

It is far easier (but more wasteful of space) to allocate one symbol table for each scope. Open scopes are stacked, pushing and popping tables as scopes are opened and closed.

Be careful though—many preprogrammed stack implementations don’t allow you to “peek” at entries below the stack top. This is necessary to lookup an identifier in all open scopes.

If a suitable stack implementation (with a peek operation) isn’t available, a linked list of symbol tables will suffice.

More on Hashtables

Hashtables are a very useful data structure. Java provides a predefined **`Hashtable`** class. Python includes a built-in *dictionary* type.

Every Java class has a **`hashCode`** method, which allows any object to be entered into a Java **`Hashtable`**.

For most objects, hash codes are pretty simple (the address of the corresponding object is often used).

But for strings Java uses a much more elaborate hash function: $\sum_{i=0}^{n-1} c_i \times 37^i$

n is the length of the string, c_i is the i-th character and all arithmetic is done without overflow checking.

Why such an elaborate hash function?

Simpler hash functions can have major problems.

Consider $\sum_{i=0}^{n-1} c_i$ (add the characters).

For short identifiers the sum grows slowly, so large indices won't often be used (leading to non-uniform use of the hash table).

We can try $\prod_{i=0}^{n-1} c_i$ (product of characters), but now (surprisingly) the size of the hash table becomes an issue. The problem is that if even one character is encoded as an even number, the product *must* be even.

If the hash table is even in size (a natural thing to do), most hash table entries will be at even positions. Similarly, if even one character is encoded as a multiple of 3, the whole product will be a multiple of 3, so hash tables that are a multiple of three in size won't be uniformly used.

To see how bad things can get, consider a hash table with size 210 (which is equal to $2\times3\times5\times7$). This should be a particularly bad table size if a product hash is used. (Why?)

Is it? As an experiment, all the words in the Unix spell checker's dictionary (26000 words) where entered. Over 50% (56.7% actually) hit position 0 in the table!

Why such non-uniformity?

If an identifier contains characters that are multiples of 2, 3, 5 and 7, then their hash will be a multiple of 210 and will map to position 0.

For example, in **`Wisconsin`**, **`n`** has an ASCII code of 110 (2×55) and **`i`** has a code of 105 ($7\times5\times3$).

If we change the table size ever so slightly, to 211, no table entry gets more than 1% of the 26000 words hashed, which is very good.

Why such a big difference? Well 211 is *prime* and there is a bit a folk-wisdom that states that prime numbers are good choices for hash table sizes. Now our product hash will cover table entries far more uniformly (small factors in the hash don’t divide the table size evenly).

Now the reason for Java’s more complex string hash function becomes evident—it can uniformly fill a hash table whose size isn’t prime.

How are Collisions Handled?

Since identifiers are often unlimited in length, the set of possible identifiers is infinite. Even if we limit ourselves is short identifiers (say 10 of fewer characters), the range of valid identifiers is greater than 26^{10}.

This means that all hash tables need to contend with *collisions*, when two different identifiers map to the same place in the table.

How are collisions handled?

The simplest approach is *linear resolution*. If identifier **`id`** hashes to position **p** in a hash table of size **s** and position **p** is already filled, we try **(p+1) mod s**, then **(p+2) mod s**, until a free position is found.

As long as the table is not too filled, this approach works well. When we approach an almost-filled situation, long search chains form, and we degenerate to an unordered list.

If the table is 100% filled, linear resolution fails.

Some hash table implementations, including Java's, set a *load factor* between 0 and 1.0. When the fraction of filled entries in the table exceeds the load factor, table size is increased and all entries are rehashed.

Note that bundling of a `hashCode` method within all Java objects makes rehashing easy to do automatically. If the hash function is external to the symbol table entries, rehashing may need to be done manually by the user.

An alternative to linear resolution is *chained resolution*, in which symbol table entries contain pointers to chains of symbols rather than a single symbol. All identifiers that hash to the same position appear on the same chain. Now overflowing table size is not catastrophic—

as the table fills, chains from each table position get longer. As long as the table is not too overfilled, average chain length will be small.