Symbol tables

A symbol table associates values or attributes (e.g., types and values) with names

What should be in a symbol table?

- variable names
- procedure and function names
- literal constants and strings
- source text labels

What information might compiler need?

- textual name
- data type
- dimension information
- declaring procedure
- lexical level of declaration
- storage class(base address)
- offset in storage
- if record, pointer to structure table
- if parameter, by reference or by value?
- can it be aliased? to what other names?
- number and type of arguments to functions

Symbol tables

- Interface(abstract definition) –
  the set of operations available to the rest of the compiler
- Implementation – how this functionality is achieved

Interface essentials

- Create and Destroy Symbol table
- Enter symbols and their attributes
- Find symbols and their attributes

Implementation techniques

Unordered list

- use an array or linked list
- O(n) time search
- O(1) time insertion
- simple and compact, but too slow in practice

Ordered list

- O(log n) time with a binary search
- O(n) time insertion
- simple, good if the tables are known in advance

Binary search tree

- O(log n) time expected search, O(n) worst case
- O(log n) time expected insertion
- less simple approach

Hash table

- O(1) time expected for search
- O(1) time expected for insertion
- worst case is very unlikely
- subtle design issues, more programmer effort
- this approach is taken in most compilers

Binary search tree

Complexity

- O(log n) time expected search, O(n) worst case
- O(log n) time expected insertion
- O(n) space

Balanced Trees

- With a balanced tree, we get the expected times
- On an arbitrary input, the tree is not necessarily balanced, What if the variables are alphabetized?
- Use an approximate balancing algorithm
    - If the height of sibling trees will vary by more than 1 after insertion, balance first by moving the deeper subtree to the shallow sibling.
    - Affects insertion performance only slightly
    - Complicates implementation
    - Space overhead is directly proportional to the number of items in the table

Hash table

Complexity

- O(1) time expected for search
- O(1) time expected for insertion
- O(n + m) space where n is the number of symbols and m is the number of hash table entries

Lookup and Insertion

1. Hash into an index
2. If Table[index] is empty
   A. lookup fails
   B. insertion adds at index
3. If Table[index] is full
   A. match implies lookup succeeds
   B. no match or insertion implies pick new index and goto step 2(full table?)

Hash table

Key issues

- hashing function
- table size should be prime(at least odd)
- k and table size should be relatively prime, where k is the number used for rehashing
- how to pick new index?
  - often the use of k=1 works as well as any other choice

Hash Functions: pick h(n) such that

- h(n) depends only on n
- h(n) is cheap to compute
- h(n) is uniform – each output is equally likely
- h(n) is randomizing – similar names no not map to similar values

Sample hash functions:

- $(c_1 + c_2 + \ldots + c_n) \bmod m$
- $(c_1 * c_2 * \ldots * c_n) \bmod m$
- So that all bits of input affect output, avoid modulus by powers of 2.

Collision resolution

A collision occurs when $h(n_1) = h(n_2)$, but $n_1 \neq n_2$.

Collisions are inevitable unless the input is known in advance, in which case a perfect hash function can be found.

Linear Resolution(a.k.a. linear probing)

- If $h(n) = k$ is full, try $(k+1) \bmod m$. If it is full, try $(k+2) \bmod m$, and so on.
- simple
- tends to build long chains(a.k.a. primary clustering)

Collision resolution(cont.)

Techniques for avoiding primary clustering

Add-the-hash rehash

- If h(n) = k is full, try (2*h(n)) mod m, If it is full, try (3*h(n)) mod m and so on.
- on each new probe, add hash code

Quadratic rehash

- $h(n)$, $(h(n) + c_1 * 1^2 + c_2 * 1) \bmod m$,
  $(h(n) + c_1 * 2^2 + c_2 * 2) \bmod m$,
  $(h(n) + c_1 * 3^2 + c_2 * 3) \bmod m$,…
- on each new probe, add polynomial based on probe number

Secondary hash functions

- h(n), (h(n) + h'(n)) mod m, (h(n) + 2h'(n)) mod m,…
- on each new probe, add secondary hash code
- reduces secondary clustering(distributed chains of items that have the same hash code)

Hash table construction

Scheme 1 – Simple table

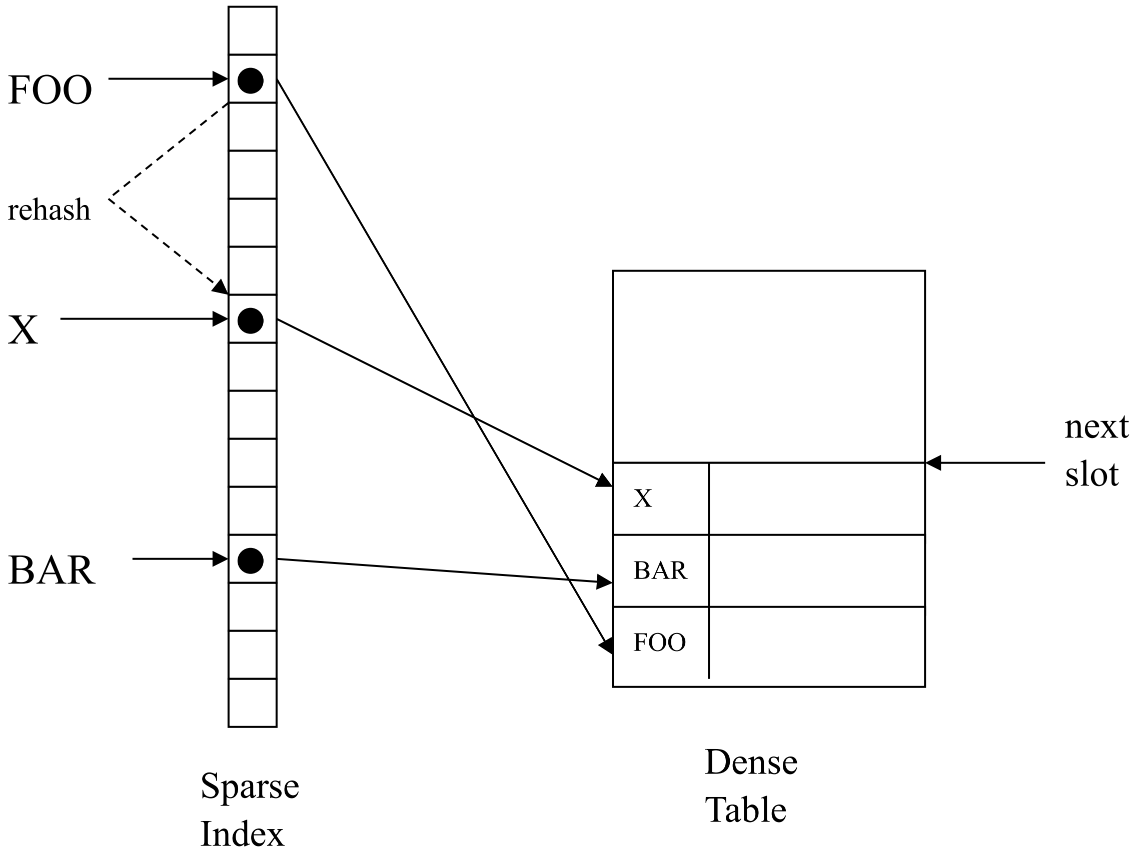
- use a simple, sparse table
- moderately large data structure
- fixed size table
- reallocation is terrible

Scheme 2 – Two-level table

- use a sparse map
- use a dense table
- table growth is easy
- map growth and rehash is simple
- file I/O simplified

Two-level hash table

Example



Bucket hashing

Another approach to hash resolution

- combines sparse index and index list
- hash item into one of the buckets
- walk list of items in bucket
- if item not found, add item to bucket

Bucket hashing complexity

- n number of names, m number of buckets
- Avg search time = $1 + ½*n/m$
- Avg insertion time = $2 + n/m$

For $n \leq km$ with fixed k, time is $O(k) = O(1)$.

- For large n, a better data structure can improve performance. However, this is rarely necessary.
- $O(n + m)$ space, but overhead of m is pretty small
- supports deletion
- not difficult to program

Bucket hashing

Can reorganize items in buckets

Scheme 1

On each lookup, move item to front of bucket list

- capitalize on locality, if possible
- reduce average case search

Scheme 2

On each lookup, move item up by one position

- capitalize on locality, if possible
- limit impact of a single lookup
- reduce average case search

Rivest and Tarjan showed that Scheme 2 does as well as any reorganizing scheme.

Block structure symbol tables

Nested scoping – relative to the current component such as a procedure, module, statement

- current scope – the innermost scope for the current component
- open scope – all scopes surrounding the current scope
- closed scope – all other scopes

Which variables are visible?

- Only variables declared in the open scopes are visible
- Definitions of the same name in an inner scope take precedence over any outer scope
- New declarations are only made to the current scope
  - ⇨ names in closed scopes are inaccessible

Nesting scope example

```
Procedure A
    H1, I, J: integer
    begin
        Procedure B
            X, Y: real
            begin
                …
            end
        Procedure C
            H2, L, M: integer
            begin
                …
            end
```

Visible names in C:

Inaccessible names in C:

Nested scopes

What information is needed?

when we ask about a name, we want the most recent declaration

the declaration may be from the current procedure or some nested procedure

What operations do we need?

- insert(name, p) – create record for name at level p
- lookup(name) – returns pointer or index
- delete(p) – deletes all names declared at level p

Nested scopes

Table per scope

Use one symbol table per scope

Chain together in list based on level of nesting

May use stack to store tables

- insert(name, p) adds to the level p table
  It may need to create the level p table and add it to the chain
- lookup(name) walks chain of tables, looking in each for name. Starts at deepest nesting and works outwards. Returns first occurrence of name
- delete(p) throws away table for level p
  It must be the top table on chain

Nested scopes

Global table

Represent all symbols in one table.

Add nesting level to all items.

Approach 1: build on bucket hashing

- insert(name, p) adds (name, p) to the front of the bucket list. Chain together records declared at level p
- lookup(name) naturally finds lexically closest definition, since first occurrence of name is from the deepest scope
- delete(p) walks the level p chain
  It removes each level p item and fixes up the pointers

Chain reorganization is more complex, but doable

Global table

Approach 2: build on linear rehash scheme

- insert(name, p) hashes by name
  1. If name isn't found, add it
  2. If name is there with wrong level,
     A. create hidden name record
     B. hang it off table slot
     C. supersede information in active slot
  3. Add name to level p chain
- lookup(name) works without change
- delete(p) walks the level p chain for each name on the chain
  1. update the active record from front of chain
  2. deletes the first hidden name record from chain

String storage

Storing identifier strings in symbol table entries require either

- variable sized entries,or
- hard small limit on size, or
- much wasted space

=> store character strings in a string space and refer to them with simple size descriptors

- maintained by symbol table, since parser and semantic routines decide whether or not to make entries permanent
- scanner can make temporary entries at the end of string space