Program Design & Hash Tables

CS 217

Program design

1. Problem statement and *requirements*
   What is the problem?
2. *Specification*
   Detailed description of *what* the system should do, not *how*
3. *Design*
   Explore design space, identify algorithms and key *interfaces*
4. *Programming*
   Implement it in the *simplest* possible way; use libraries
5. *Testing*
   Debug and test until the implementation is *correct* and *efficient enough*
6. *Iterate*
   Do the design and implementation conform to the specification?

Design methodologies

- Two important design methodologies
  - *top-down* design, or stepwise refinement
  - *bottom-up* design
- Reality: use both
  - top-down: what functionality do I need?
    Avoids designing and building useless functionality
  - bottom-up: what functionality do I know how to provide?
    Avoids requiring impossible functionality
- Iterate up and down over the design until everything is both useful and feasible
  - sometimes overlaps with implementation phase

Stepwise refinement

- Top-down design
  starts with a high-level abstract solution
  refines it by successive transformations to lower-level solutions
  refinement ends at programming-language statements
- Key idea: each refinement or *elaboration*
  must be *small* and *correct*
  must move toward final solution
- Accompany refinements with *assertions*
- Refinements use English & pseudocode, but ultimately result in code

Example: library books

1. Problem statement:
   The circulation file has a line of author,title for each checked out book
   Need a program to find books checked out frequently
2. Specification
   Read a text file; print out one copy of any line that appears 10 or more times
3. Design: how many lines are in a typical circulation file?

```
<findfreq> ≡
    <for each line of input>
        <look up the line in the table (add it if not already there)>
        <increment this line's count>
    <for each member of the table>
        <if that member's count ≥ 10>
            <print the line>
```

4. Programming: make forward progress by elaborating chunks

What modules?

- ADT: string table
- Modules:
  - **`main.c`** handle command-line arguments (if any) and top-level loops

```
<findfreq> ≡
   <includes>
   <defines>
   int main(int argc, char *argv[]) {
     <locals>
     <for each line of input>
       <look up the line in the table (add it if not already there)>
       <increment this line's count>
     <for each member of the table>
       <if that member's count ≥ 10>
          <print the line>
     return EXIT_SUCCESS;
   }
```

  - **`symtable.h`** interface for string table
  - **`symtable.c`** implementation for string table

Elaboration

- Some elaborations can be done without defining the ADTs

```
<for each line of input> ≡
  while (fgets(line, MAXLINE, stdin))

<defines> ≡
  #define MAXLINE 512

<locals> ≡
  char line[MAXLINE];
```

ADT: string table

`symtable.h` describes *abstract* operations, *not* implementation; *what*, not *how*

```
typedef struct SymTable *SymTable_T;
SymTable_T SymTable_new(void); /* create a new, empty table. */
int SymTable_put(SymTable_T table, char *key,
                 void *value);
  /* enter (key,value) binding in the table; else return 0 if already there */
void *SymTable_get(SymTable_T table, char *key);
  /* look up key in the table, return value (if present) or else NULL */
void SymTable_map(SymTable_T table,
   void (*f)(char *key, void *value, void *extra),
   void *extra);
  /* apply f to every key in the table ... */
```

This was *top-down* design: specify just those operations necessary for client program

Next step: re-use, if possible

- Avoid some work by searching for an existing module or library that can do the work of SymTable module
- If found, then throw away symtable.h
- Let's pretend we didn't find one

A bit of bottom-up design

- Now that we've committed to create SymTable ADT, add more operations that make it useful in other applications.
- Don't get carried away! You'll end up doing useless work
- This step is optional: you can always do it later as needed.

More of symtable interface

```
void SymTable_free(SymTable_T table);
```
/* Free table */

```
int SymTable_getLength(SymTable_T table);
```
/* Return the number of bindings in table.
It is a checked runtime error for table to be NULL. */

```
int SymTable_remove(SymTable_T table,
                    char *key);
```

/* Remove from table the binding whose key is key. Return 1 if successful, 0 otherwise.
It is a checked runtime error for table or key to be NULL. */

Back to the client

- ADT interface gives enough information to finish the client, main.c

<locals> +≡
```
    SymTable_T table = SymTable_new();
    struct stats *v;
```

<includes> +≡
```
    #include "symtable.h";
```

<global-defs> ≡
```
    struct stats {int count;};
```
(also must define makeStats...)

<look up the line in the table (add it if not already there)> ≡
```
    v = (struct stats *)SymTable_get(table, line);
    if (!v) {
       v = makeStats(0);
       SymTable_put(table, line, (void *)v);
    }
```

Finishing the client

<for each member of the table> ≡

```
   SymTable_map(table, maybeprint, NULL);
```

<if that member's count ≥ 10, print the line> ≡

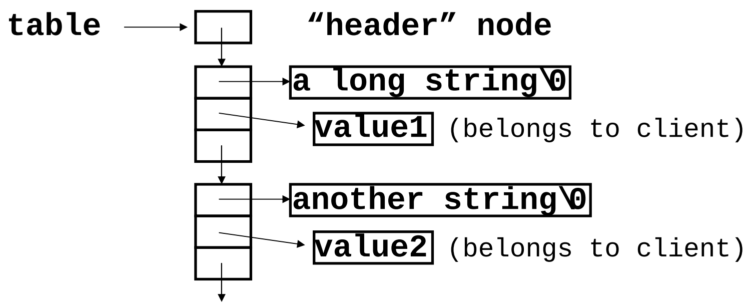
```
void maybeprint(char *key, void *stats,
                void *extra){
 if (((struct stats*)stats)->count >= 10)
    fputs(key, stdout);
}
```

What the client main looks like

```
int main(int argc, char *argv[]) {
  char line[MAXLINE];
  SymTable_T table = SymTable_new();
  struct stats *v;
  while (fgets(line, MAXLINE, stdin)) {
    v = (struct stats *)SymTable_get(table, line);
    if (!v) {
         v = makeStats(0);
         SymTable_put(table, line, (void *)v);
    }
    incrementStats(v,1);
  }
  SymTable_map(table, maybeprint, NULL);
  return EXIT_SUCCESS;
 }
```

ADT implementation

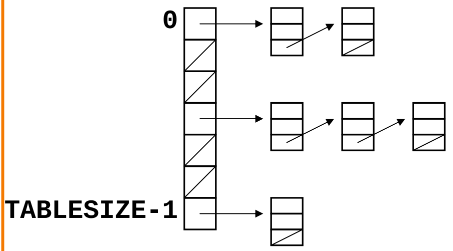
- Now, begin to design the ADT implementation
- Start with a simple algorithm / data structure
  - It's good for debugging and testing the interface
  - Maybe it's good enough for the production system -- that would save the work of implementing a clever algorithm



Testing

- 5. Testing: findfreq works, but runs too slowly on large inputs. Why?
  - Improve symtable's implementation; don't change its interface
- Solution: use a hash table

  A symtable will be a pointer to an array of TABLESIZE linked lists
  "Hash" the string into an integer h
  let i = h % TABLESIZE
  search the ith linked list for the string, or
  add the string to the head of the ith list



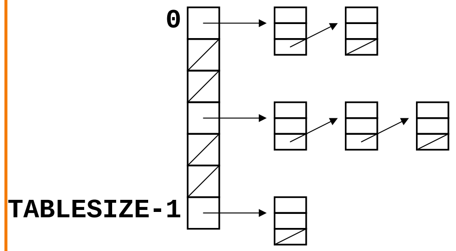
How large an array?

Array should be long enough that “bucket” size is 1.

If the buckets are short, then lookup is fast.

If there are some very long buckets, then average lookup is slow.

This is OK:



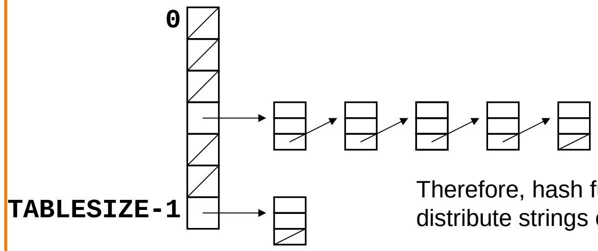
The need for a good hash function

Array should be long enough that average “bucket” size is 1.

If the buckets are short, then lookup is fast.

If there are some very long buckets, then average lookup is slow.

This is not so good:



Therefore, hash function must evenly distribute strings over integers 0..TABLESIZE

A reasonable hash function

How to hash a string into an integer?

Add up all the characters? (won’t distribute evenly enough)

How about this: $(\Sigma\ a^i x_i) \bmod c$ (best results if a,c relatively prime)

- Choose a = 65599, c = 2^{32}

```
unsigned hash(char *string) {
  int i; unsigned h = 0;
  for (i=0; string[i]; i++)
      h = h * 65599 + string[i];
  return h;
}
```

- How does this implement $(\Sigma\ a^i x_i) \bmod c$?

A reasonable hash function

How to hash a string into an integer?

Add up all the characters? (won’t distribute evenly enough)

How about this: $(\Sigma\ a^i x_i) \bmod c$ (best results if a,c relatively prime)

- Choose a = 65599, c = 2^{32}

```
unsigned hash(char *string) {
  int i; unsigned h = 0;
  for (i=0; string[i]; i++)
      h = h * 65599 + string[i];
  return h;
}
```

$$\begin{aligned} & x_3a^3+x_2a^2+x_1a+x_0 \\ = {} & a(x_3a^2+x_2a+x_1)+x_0 \\ = {} & a(a(ax_3+x_2)+x_1)+x_0 \end{aligned}$$

- How does this implement $(\Sigma\ a^i x_i) \bmod c$?

Hash table in action

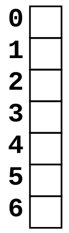
Example: TABLESIZE = 7

Lookup (and enter, if not present) these strings: the, cat, in, the, hat

Hash table initially empty.

First word: the. hash("the") = 965156977. 965156977 % 7 = 1.

Search the linked list table[1] for the string "the"; not found.



Hash table in action

Example: TABLESIZE = 7

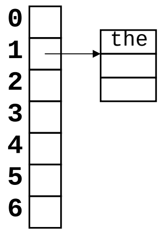
Lookup (and enter, if not present) these strings: the, cat, in, the, hat

Hash table initially empty.

First word: "the". hash("the") = 965156977. 965156977 % 7 = 1.

Search the linked list table[1] for the string "the"; not found

Now: table[1] = makelink(key, value, table[1])

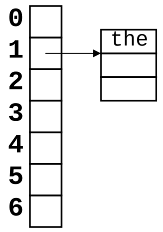


Hash table in action

Second word: "cat". hash("cat") = 3895848756. 3895848756 % 7 = 2.

Search the linked list table[2] for the string "cat"; not found

Now: table[2] = makelink(key, value, table[2])

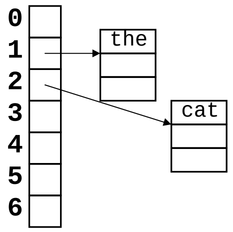


Hash table in action

Third word: "in". hash("in") = 6888005. 6888005% 7 = 5.

Search the linked list table[5] for the string "in"; not found

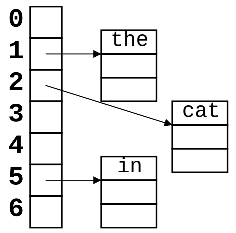
Now: table[5] = makelink(key, value, table[5])



Hash table in action

Fourth word: “the”. hash(“the”) = 965156977. 965156977 % 7 = 1.

Search the linked list table[1] for the string “the”; found it!



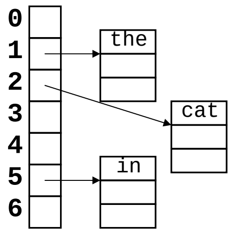
Hash table in action

Fourth word: “hat”. hash(“hat”) = 865559739. 865559739 % 7 = 2.

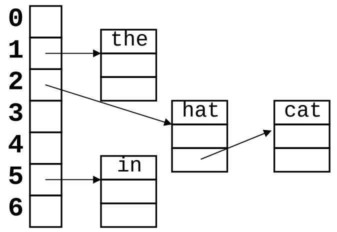
Search the linked list table[2] for the string “hat”; not found.

Now, insert “hat” into the linked list table[2].

At beginning or end? Doesn’t matter.



Hash table in action



Number of buckets

- Average bucket size should be short
- Thus, number of buckets should be (approximately) greater than number of entries in table
- If (approximate) number of entries is known in advance, this is easy to arrange
- If (approximate) number of entries is unpredictable, then one can dynamically grow the hash table
- How to do it; cost analysis; ...

References on hashing

- Kernighan & Pike, *Practice of Programming*, §2.9
- Hanson, *C Interfaces and Implementations,* §3.2