5-1

a.

```
struct bucket {
    string key;
    void *binding;
    struct bucket *next;
    int count;
};

#define INITIAL_SIZE 109
struct bucket **table;
int size = INITIAL_SIZE;
int count = 0;

unsigned int hash(char *s0) {
    unsigned int h = 0;
    char *s;
    for(s = s0; *s; s++)
        h = h * 65599 + *s;
    return h;
}

struct bucket *Bucket(string key, void *binding, struct bucket *next, int count) {
    struct bucket *b = checked_malloc(sizeof(*b));
    b->key = key;
    b->binding = binding;
    b->next = next;
    b->count = count;
    return b;
}

void insert(string key, void *binding) {
    int index = hash(key) % size;
    struct bucket *b = table[index];
    while (b) {
        if (b->key == key) {
            b->binding = binding;
            return;
        }
        b = b->next;
    }
    table[index] = Bucket(key, binding, table[index], 1);
    count++;
    if ((float)count / size > 2) {
        int old_size = size;
        size *= 2;
        struct bucket **old_table = table;
```

```
        table = checked_malloc(sizeof(struct bucket*) * size);
        memset(table, 0, sizeof(struct bucket*) * size);
        count = 0;
        for (int i = 0; i < old_size; i++) {
            struct bucket *b = old_table[i];
            while (b) {
                struct bucket *next = b->next;
                int index = hash(b->key) % size;
                b->next = table[index];
                table[index] = b;
                count++;
                b = next;
            }
        }
        free(old_table);
    }
}

void *lookup(string key) {
    int index = hash(key) % size;
    struct bucket *b = table[index];
    while (b) {
        if (b->key == key)
            return b->binding;
        b = b->next;
    }
    return NULL;
}

void pop(string key) {
    int index = hash(key) % size;
    struct bucket *b = table[index];
    struct bucket *prev = NULL;
    while (b) {
        if (b->key == key) {
            if (prev)
                prev->next = b->next;
            else
                table[index] = b->next;
            free(b);
            return;
        }
        prev = b;
        b = b->next;
    }
}
```

b.

```
struct bucket {
```

```
    string key;
    void *binding;
    struct bucket *next;
};

#define INITIAL_SIZE 109

unsigned int hash(char *s0) {
    unsigned int h = 0;
    char *s;
    for(s = s0; *s; s++)
        h = h * 65599 + *s;
    return h;
}

struct bucket *Bucket(string key, void *binding, struct bucket *next) {
    struct bucket *b = checked_malloc(sizeof(*b));
    b->key = key;
    b->binding = binding;
    b->next = next;
    return b;
}

void insert(struct bucket **table, int size, string key, void *binding) {
    int index = hash(key) % size;
    struct bucket *b = table[index];
    while (b) {
        if (b->key == key) {
            b->binding = binding;
            return;
        }
        b = b->next;
    }
    table[index] = Bucket(key, binding, table[index]);
}

void *lookup(struct bucket **table, int size, string key) {
    int index = hash(key) % size;
    struct bucket *b = table[index];
    while (b) {
        if (b->key == key)
            return b->binding;
        b = b->next;
    }
    return NULL;
}

void pop(struct bucket **table, int size, string key) {
    int index = hash(key) % size;
    struct bucket *b = table[index];
    struct bucket *prev = NULL;
    while (b) {
```

```
        if (b->key == key) {
            if (prev)
                prev->next = b->next;
            else
                table[index] = b->next;
            free(b);
            return;
        }
        prev = b;
        b = b->next;
    }
}
```

6-3

`a, b` : registers; As function arguments, the compiler will typically assign them to registers unless there are too many arguments.

`c` : memory; Because the array size is not known when compiling.

`d` : registers; Because it's a single integer andonly used in a few instructions and its value is not needed across functions calls.

`e` : registers; Similar to `d` . But the return value of `g` should be kept in memory.

6-7

a.

```
mov r1, fp        // move frame pointer to register r1
addi r1, r1, #offset // add offset of `output` variable to r1
ldr r0, [r1]      // load the value of `output` into register r0
```

b.

```
ldr r1, [fp, #display_offset] // load display pointer from current activation record
ldr r1, [r1, #depth_offset]   // load frame pointer for `prettyprint` from display
addi r1, r1, #offset          // add offset of `output` variable to r1
ldr r0, [r1]                  // load the value of `output` into register r0
```