

Complex Libraries

Using Hash Dictionaries

Playing Hash Table

*You are the new produce manager of the local grocery store.
You want to use a dictionary to track your fruit inventory.*

Entries have the form

(“banana”, 20)

where

- “banana” is the **key**
 - 20 is associated data, like the number of cases in stock
-
- Let’s observe your initial interactions with a hypothetical hash dictionary library

This is your side

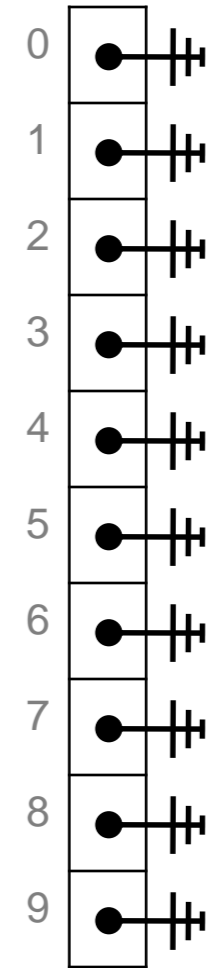
You begin by creating a *new dictionary*

Client | **Implementation**

Create a new hash dictionary

Here you go!

This is what is going on in the library



- This library uses separate-chaining hash tables to implement dictionaries
- It decides on an initial capacity of 10
 - it's probably self-resizing

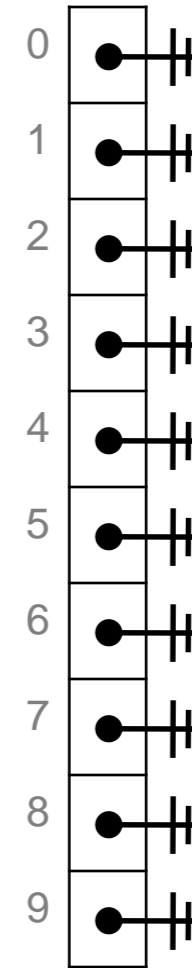
Client | Implementation

Insert A = ("apple", 20)

What's the key of (A)?

✓ new dictionary

Next, you *insert*
A = ("apple", 20)



- Why is the library asking this?
 - it does not know what entries are
 - (A) is just a pointer to some struct
 - no sense of what's in it
- ***You need to tell it***

Client Implementation

Insert A = ("apple", 20)

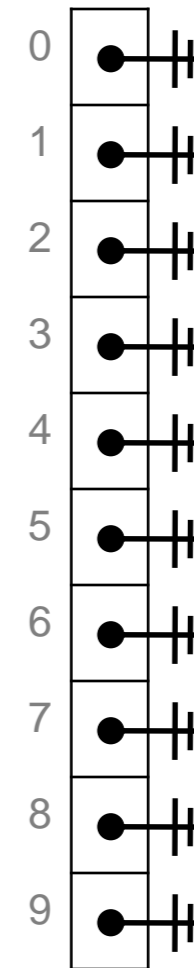
What's the key of (A)?

"apple"

What's its hash value?

✓ new dictionary

Next, you *insert*
A = ("apple", 20)



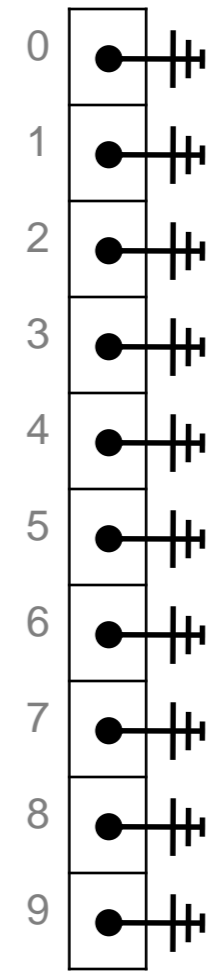
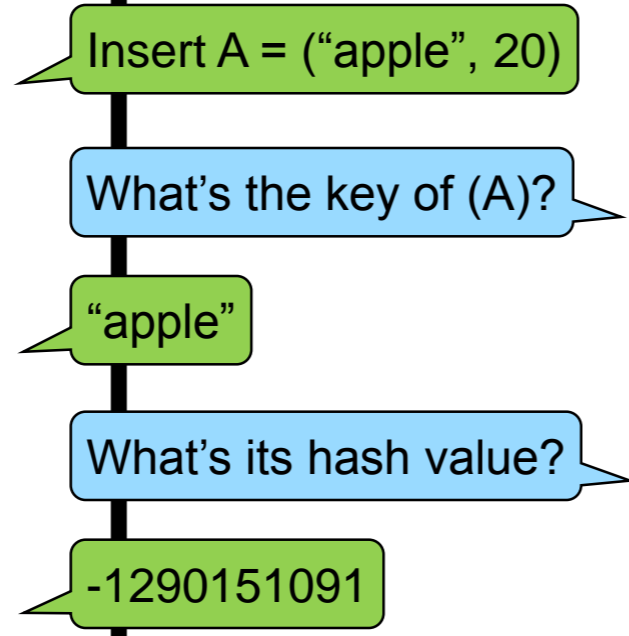
- Why is the library asking this?
 - it does not know the type of keys
 - even if it did, there are many ways to hash them
- ***You need to tell it***

Key	Hash
"apple"	-1290151091
"berry"	-514151789
"banana"	207055587
"grape"	-581390202
"lemon"	-665562942
"lime"	2086736531
"pumpkin"	-1189657311

✓ new dictionary

Next, you *insert*
A = ("apple", 20)

Client Implementation



- $-1290151091 \% 10$ is -1 in $C0$
 - not a valid array index! **x**
 - the library needs a more robust way to compute the hash index
- Let's say it keeps the last digit

Exercise!

Key	Hash
"apple"	-1290151091
"berry"	-514151789
"banana"	207055587
"grape"	-581390202
"lemon"	-665562942
"lime"	2086736531
"pumpkin"	-1189657311

✓ new dictionary

Next, you *insert*
A = ("apple", 20)

Client Implementation

Insert A = ("apple", 20)

What's the key of (A)?

"apple"

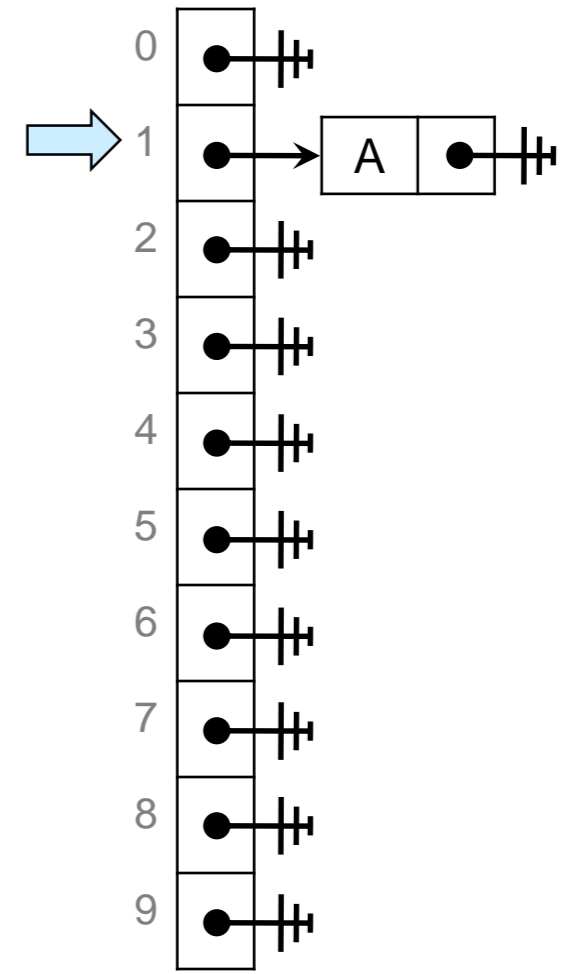
What's its hash value?

-1290151091

Ok. The hash index is 1.

This chain is empty. I can insert entry (A) there.

Done



- The library asked for
 - the key of the entry
 - the hash value of the key

Funny! Libraries didn't ask for anything in the past

Key	Hash
"apple"	-1290151091
"berry"	-514151789
"banana"	207055587
"grape"	-581390202
"lemon"	-665562942
"lime"	2086736531
"pumpkin"	-1189657311

Client Implementation

Insert B = ("banana", 10)

What's the key of (B)?

"banana"

What's its hash value?

207055587

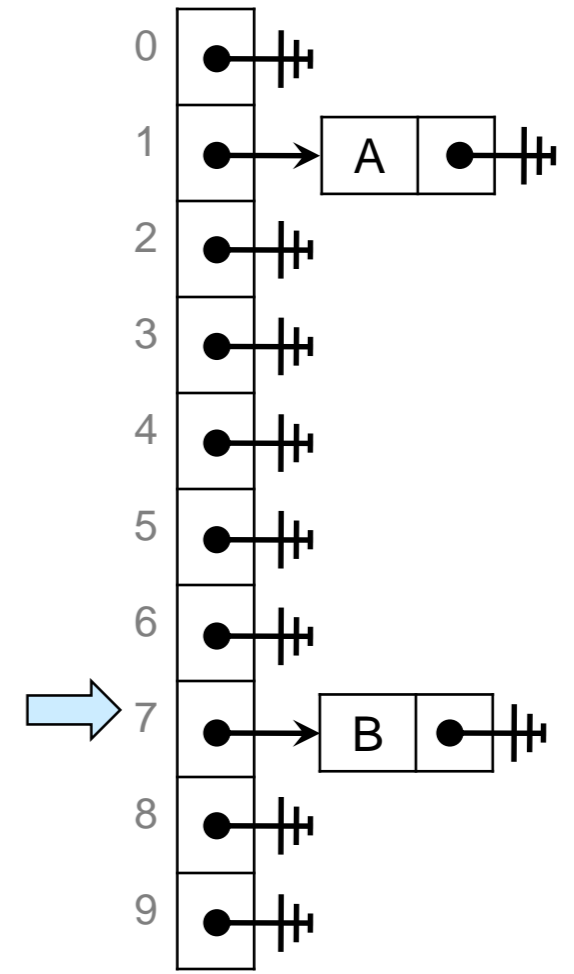
Ok. The hash index is 7.

This chain is empty. I can insert entry (B) there.

Done

- ✓ new dictionary
- ✓ insert A = ("apple", 20)

Next, you *insert* B = ("banana", 10)



Same as for (A)

Key	Hash
"apple"	-1290151091
"berry"	-514151789
"banana"	207055587
"grape"	-581390202
"lemon"	-665562942
"lime"	2086736531
"pumpkin"	-1189657311

Client Implementation

Insert C = ("pumpkin", 50)

What's the key of (C)?

"pumpkin"

What's its hash value?

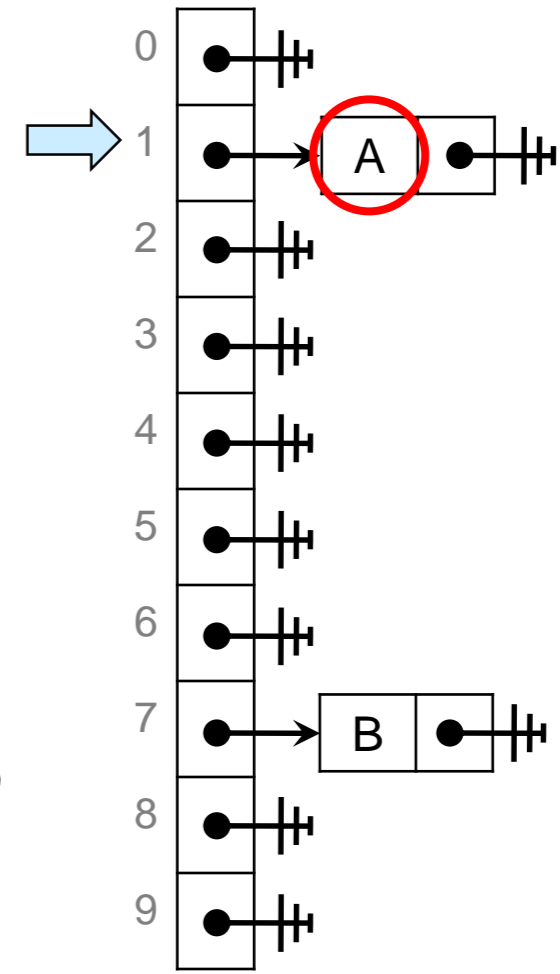
-1189657311

Ok. The hash index is 1. It points to a node for entry (A)

What's the key of (A)?

- ✓ new dictionary
- ✓ insert A = ("apple", 20)
- ✓ insert B = ("banana", 10)

Next, you *insert* C = ("pumpkin", 50)



- Why is the library asking this?
 - it does not know what entries are
 - (A) is just a pointer to some struct
 - no sense of what's in it
- *You need to tell it*

Key	Hash
"apple"	-1290151091
"berry"	-514151789
"banana"	207055587
"grape"	-581390202
"lemon"	-665562942
"lime"	2086736531
"pumpkin"	-1189657311

Client Implementation

Insert C = ("pumpkin", 50)

What's the key of (C)?

"pumpkin"

What's its hash value?

-1189657311

Ok. The hash index is 1. It points to a node for entry (A)

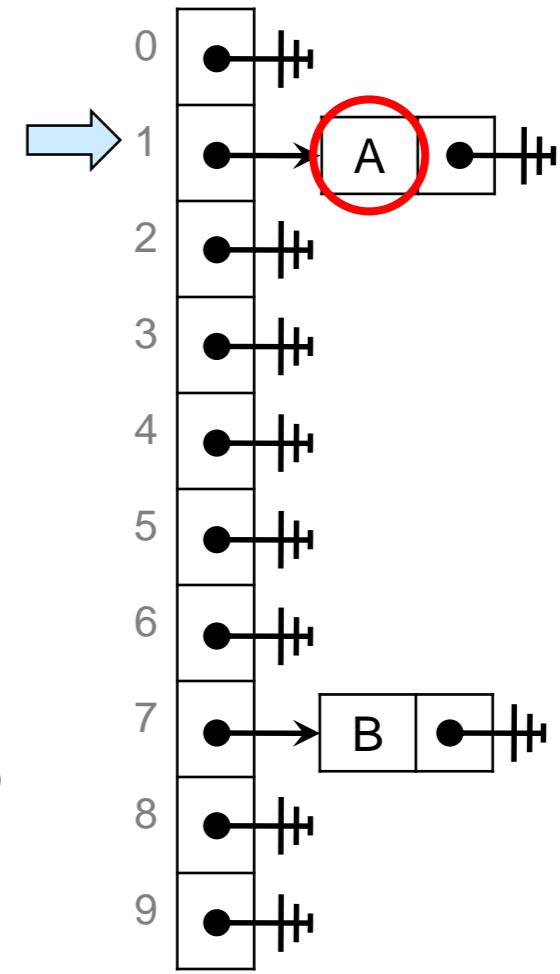
What's the key of (A)?

"apple"

Is it the same as "pumpkin"?

- ✓ new dictionary
- ✓ insert A = ("apple", 20)
- ✓ insert B = ("banana", 10)

Next, you *insert* C = ("pumpkin", 50)



- Why is the library asking this?
 - it does not know the type of keys
 - even if it did, there are many ways to compare them
- *You need to tell it*

Key	Hash
"apple"	-1290151091
"berry"	-514151789
"banana"	207055587
"grape"	-581390202
"lemon"	-665562942
"lime"	2086736531
"pumpkin"	-1189657311

- ✓ new dictionary
- ✓ insert A = ("apple", 20)
- ✓ insert B = ("banana", 10)

Next, you *insert* C = ("pumpkin", 50)

Client Implementation

Insert C = ("pumpkin", 50)

What's the key of (C)?

"pumpkin"

What's its hash value?

-1189657311

Ok. The hash index is 1. It points to a node for entry (A)

What's the key of (A)?

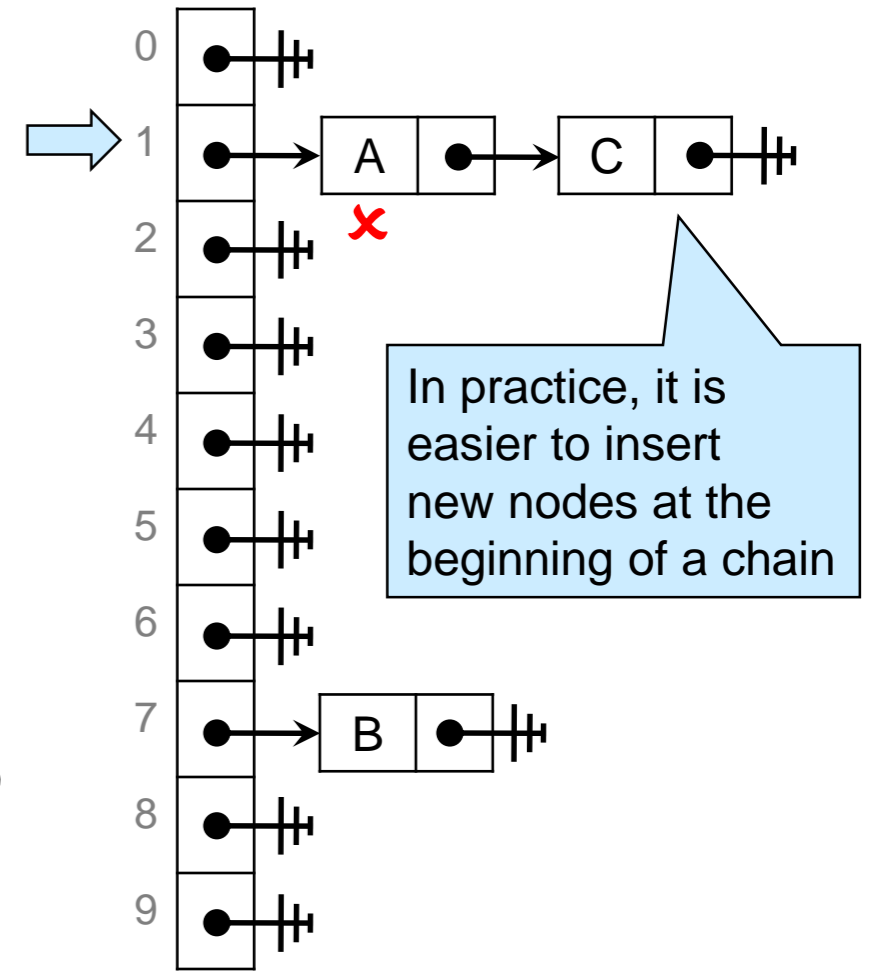
"apple"

Is it the same as "pumpkin"?

No

There is no next node. I can insert entry (C) there.

Done



- The library asked for
 - the key of the entry
 - the hash value of the key
 - whether two keys are the same

Key	Hash
"apple"	-1290151091
"berry"	-514151789
"banana"	207055587
"grape"	-581390202
"lemon"	-665562942
"lime"	2086736531
"pumpkin"	-1189657311

- ✓ new dictionary
- ✓ insert A = ("apple", 20)
- ✓ insert B = ("banana", 10)
- ✓ insert C = ("pumpkin", 50)

Next, you look up "apple"

Client Implementation

Look up "apple"

What's its hash value?

-1290151091

Ok. The hash index is 1. It points to a node for entry (A)

What's the key of (A)?

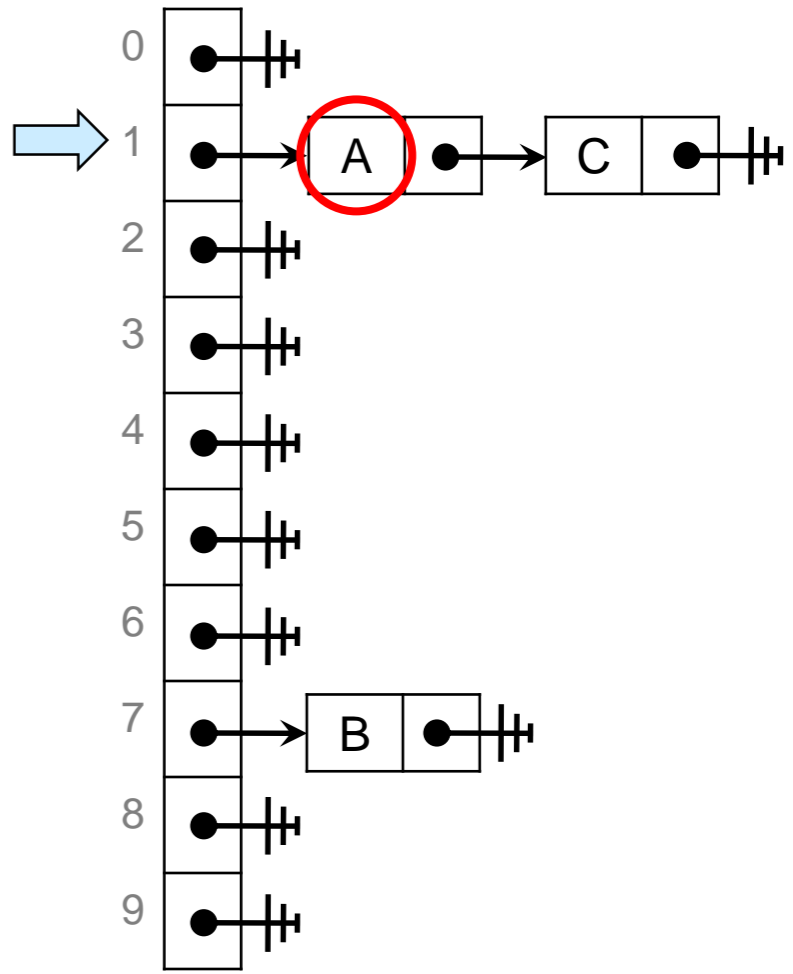
"apple"

Is it the same as "apple"?

Yes

Found

(A)



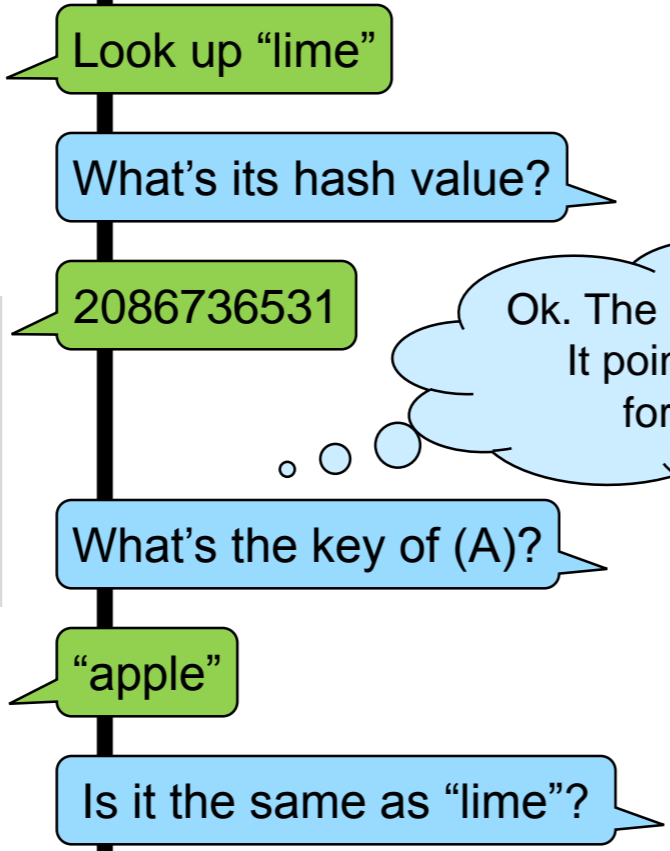
● Looking up a key follows the same steps as inserting an entry

Key	Hash
"apple"	-1290151091
"berry"	-514151789
"banana"	207055587
"grape"	-581390202
"lemon"	-665562942
"lime"	2086736531
"pumpkin"	-1189657311

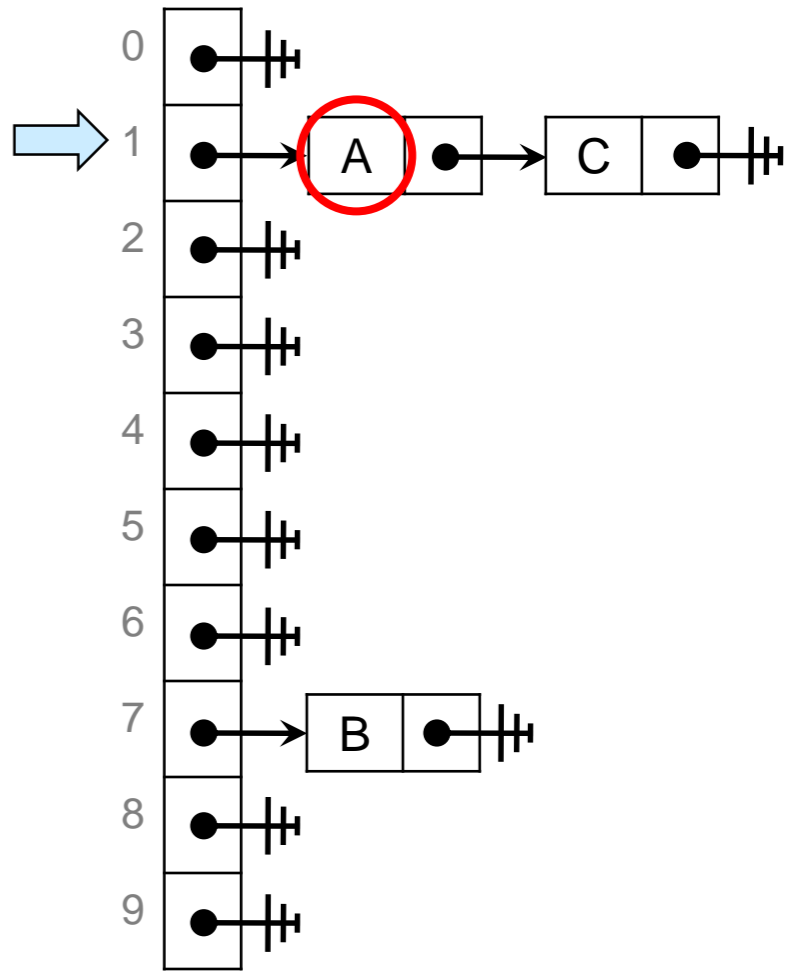
- ✓ new dictionary
- ✓ insert A = ("apple", 20)
- ✓ insert B = ("banana", 10)
- ✓ insert C = ("pumpkin", 50)
- ✓ look up "apple"

Next, you look up "lime"

Client Implementation



Ok. The hash index is 1. It points to a node for entry (A)



● The library goes through the chain node by node

Key	Hash
"apple"	-1290151091
"berry"	-514151789
"banana"	207055587
"grape"	-581390202
"lemon"	-665562942
"lime"	2086736531
"pumpkin"	-1189657311

Client Implementation

Look up "lime"

What's its hash value?

2086736531

Ok. The hash index is 1. It points to a node for entry (A)

What's the key of (A)?

"apple"

Is it the same as "lime"?

No

Ok. The next node has entry (C)

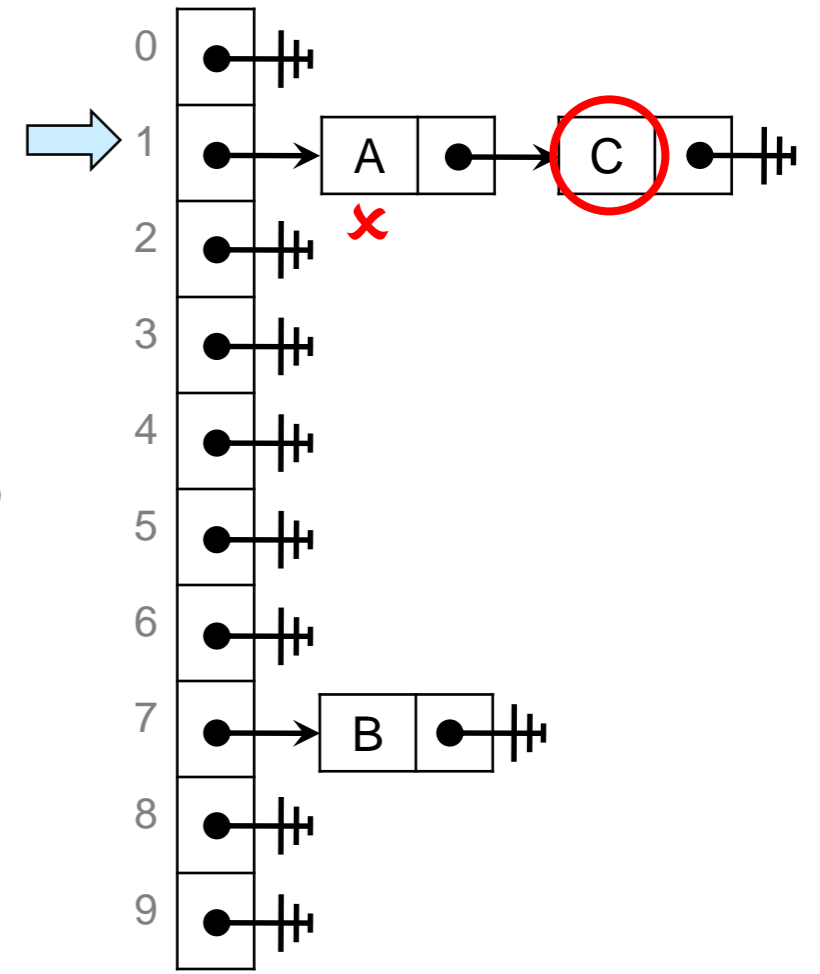
What's the key of (C)?

"pumpkin"

Is it the same as "lime"?

- ✓ new dictionary
- ✓ insert A = ("apple", 20)
- ✓ insert B = ("banana", 10)
- ✓ insert C = ("pumpkin", 50)
- ✓ look up "apple"

Next, you look up "lime"



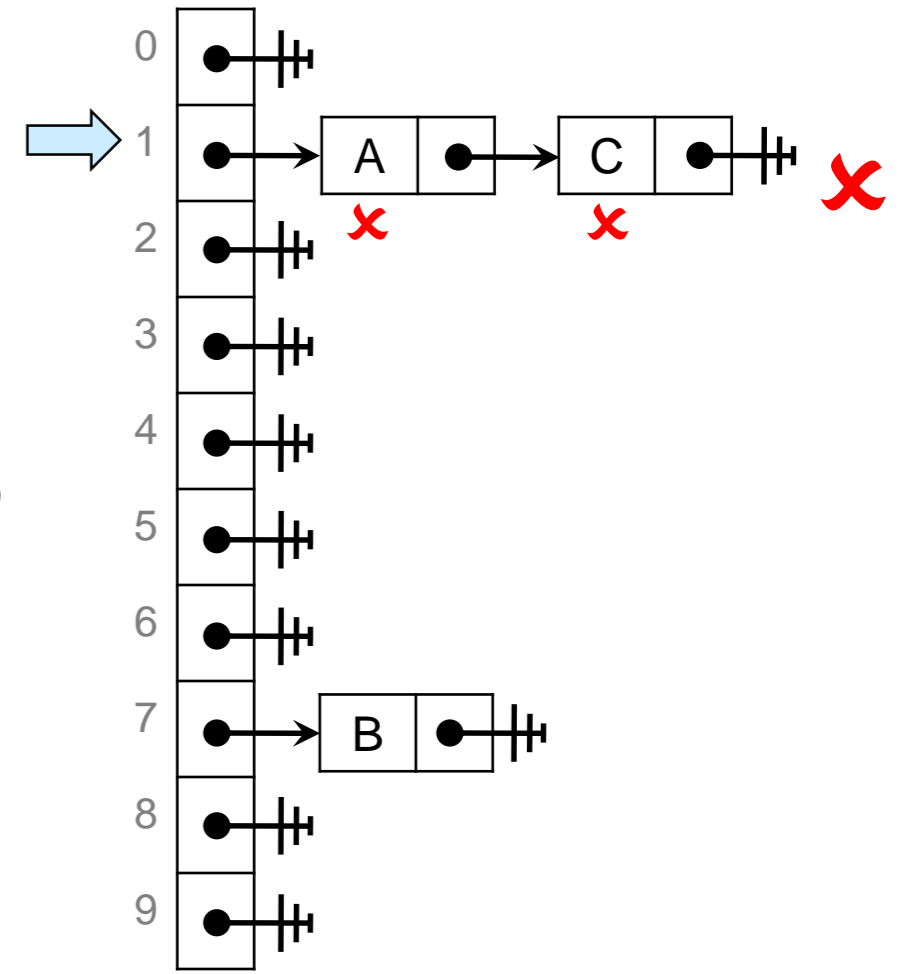
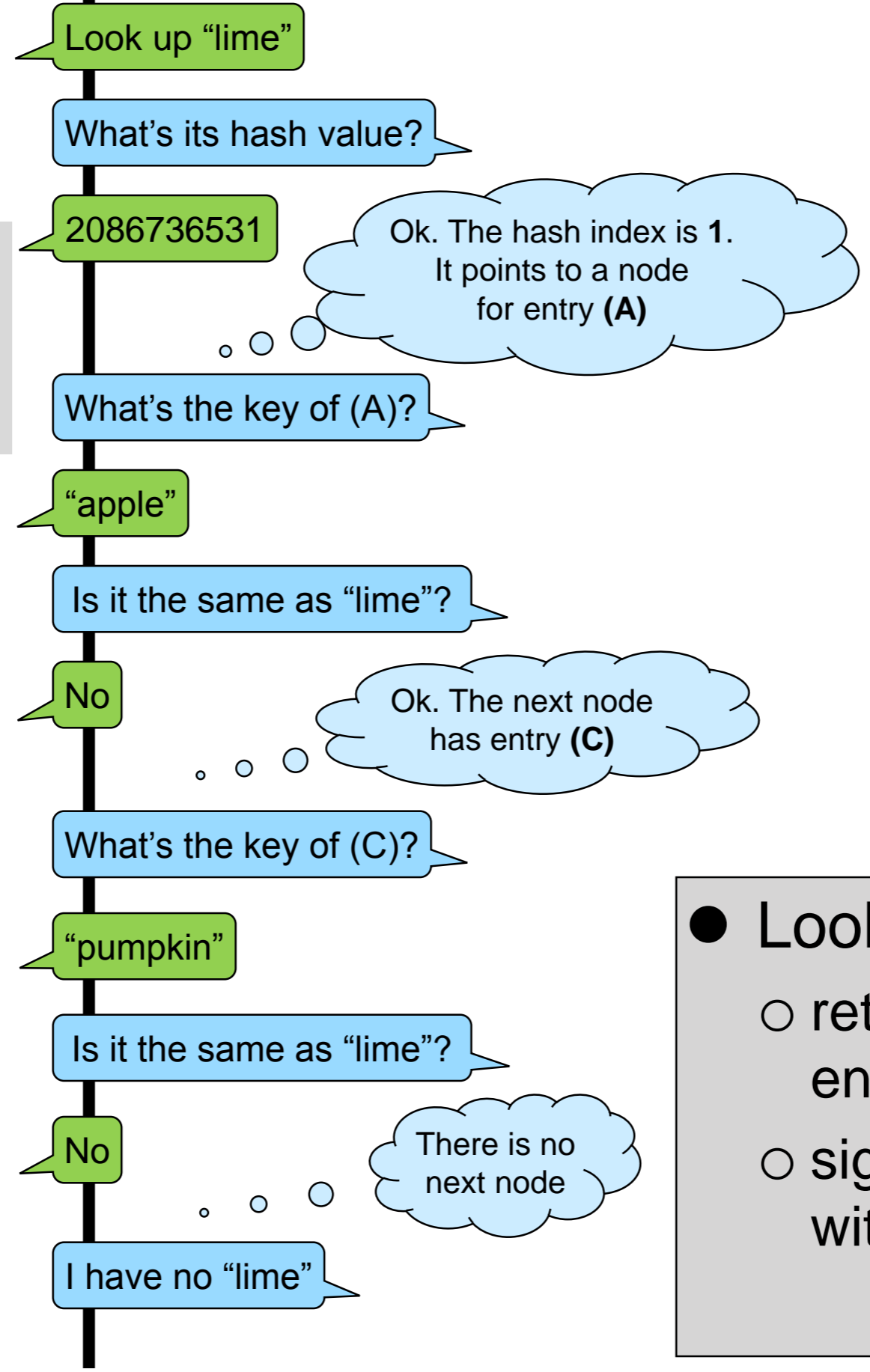
● The library goes through the chain node by node

Key	Hash
"apple"	-1290151091
"berry"	-514151789
"banana"	207055587
"grape"	-581390202
"lemon"	-665562942
"lime"	2086736531
"pumpkin"	-1189657311

- ✓ new dictionary
- ✓ insert A = ("apple", 20)
- ✓ insert B = ("banana", 10)
- ✓ insert C = ("pumpkin", 50)
- ✓ lookup "apple"

Next, you look up "lime"

Client Implementation



- Looking up a key can
 - return the associated entry, or
 - signal there is no entry with this key

Key	Hash
"apple"	-1290151091
"berry"	-514151789
"banana"	207055587
"grape"	-581390202
"lemon"	-665562942
"lime"	2086736531
"pumpkin"	-1189657311

Client Implementation

Insert D = ("banana", 20)

What's the key of (B)?

"banana"

What's its hash value?

207055587

Ok. The hash index is 7. It points to a node for entry (B)

What's the key of (B)?

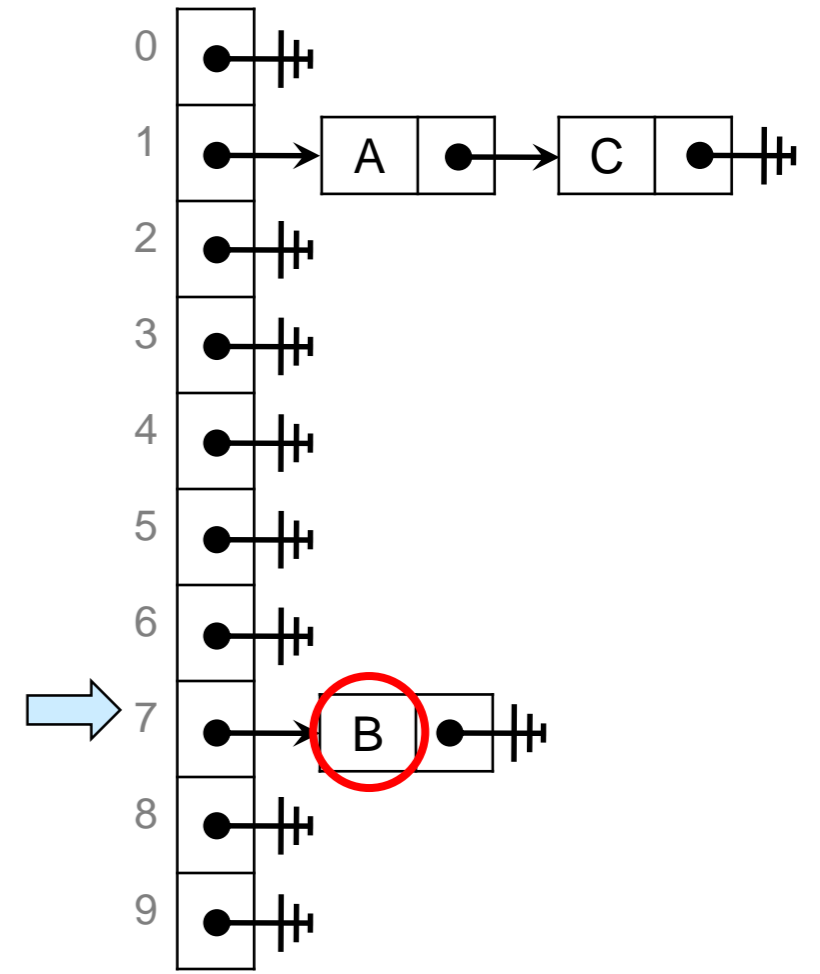
"banana"

Is it the same as "banana"?

Yes

- ✓ new dictionary
- ✓ insert A = ("apple", 20)
- ✓ insert B = ("banana", 10)
- ✓ insert C = ("pumpkin", 50)
- ✓ look up "apple"
- ✓ look up "lime"

Next, you *insert* D = ("banana", 20)



● What to do if the key is already there?

Key	Hash
"apple"	-1290151091
"berry"	-514151789
"banana"	207055587
"grape"	-581390202
"lemon"	-665562942
"lime"	2086736531
"pumpkin"	-1189657311

- ✓ new dictionary
- ✓ insert A = ("apple", 20)
- ✓ insert B = ("banana", 10)
- ✓ insert C = ("pumpkin", 50)
- ✓ look up "apple"
- ✓ look up "lime"

Next, you *insert* D = ("banana", 20)

Client Implementation

Insert D = ("banana", 20)

What's the key of (B)?

"banana"

What's its hash value?

207055587

Ok. The hash index is 7. It points to a node for entry (B)

What's the key of (B)?

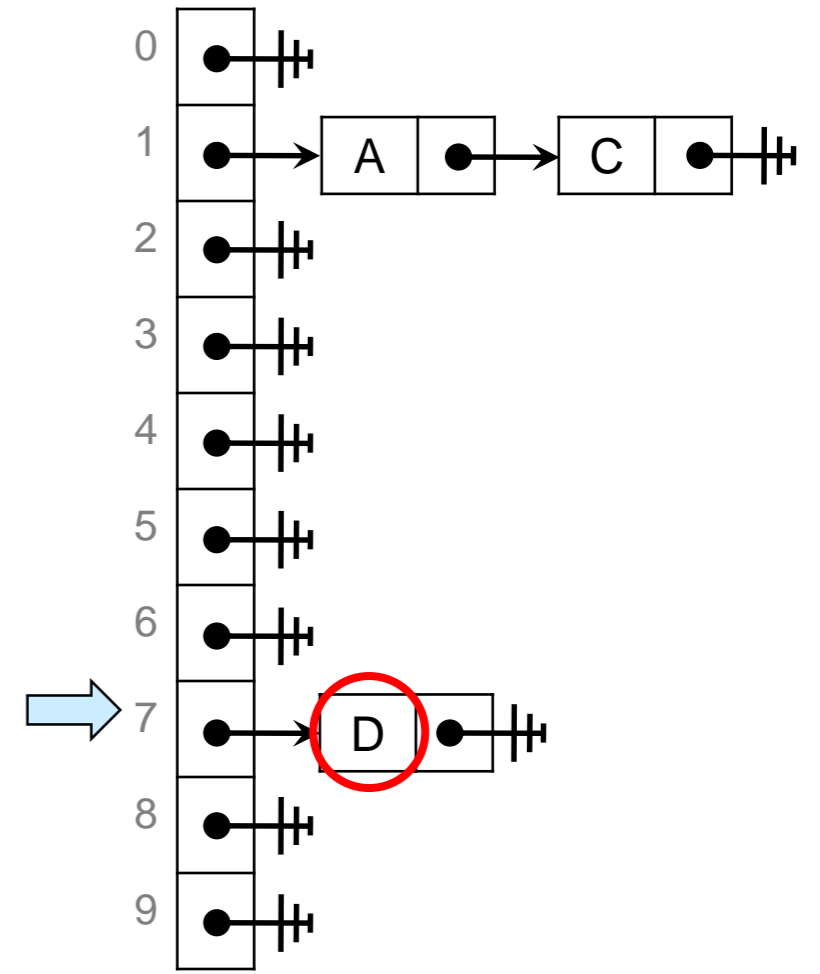
"banana"

Is it the same as "banana"?

Yes

Ok. This is where to insert (D)

Done



- What to do if the key is already there?
- **Overwrite** the stored entry

What Have we Learned?

- The library needs information from the client to do its job
 - the key of an entry
 - the hash value of a key
 - whether two keys are the same
- How shall the client provide this information?
 - Back and forth like we did?
 - Too cumbersome
 - we want to just call lookup and get a result
 - Supply **functions** the library can use to find this information
 - a function that returns the key of an entry
 - a function that computes the hash value of a key
 - a function that determines whether two keys are the same

Hash Dictionary Interface

What the Library Provides

- A type for using dictionaries

- `hdict_t`

`hdict_t` because we will be implementing it using hash tables

- Some operations

- creating a new dictionary

- `hdict_new`

- looking up a key in a dictionary

- `hdict_lookup`

- inserting an entry into a dictionary

- `hdict_insert`

Real dictionary libraries provide many more operations.
Let's keep it simple

Let's write
the interface
of this library

Creating a Dictionary

By now we anticipate this will be a pointer ...

... and that a dictionary shall never be NULL

```
Library Interface
// typedef _____* hdict t;
hdict_t hdict_new(int capacity)
/* @requires capacity > 0;          @*/
/* @ensures \result != NULL;      @*/;
// ...
```

- Clients have a sense of how many entries may end up in a dictionary
 - Let them specify an initial capacity
 - whether the implementation is self-resizing or not
 - An initial capacity of 0 makes no sense
 - disallow it in precondition

Looking up a key

- `hdict_lookup` looks up a key in a dictionary ...
 - we need a type `key` of keys
- ... and returns the associated entry ...
 - we need a type `entry` of entries
- .. unless there is no entry with this key in the dictionary
 - it then must signal that no entry was found
 - Arrange so that `entry` is a **pointer type**
 - either a pointer to the entry it found
 - or NULL to represent “not found”

```
Library Interface
// typedef _____* hdict_t;

hdict_t hdict_new(int capacity)
/* @requires capacity > 0;          @*/
/* @ensures \result != NULL;       @*/;

entry hdict_lookup(hdict_t D, key k)
/* @requires D != NULL;            @*/;

// ...
```

Key and Entry Types

```
Client Interface
// typedef _____* entry;
// typedef _____ key;
// ...
```

The client needs to define types `entry` and `key`, and `entry` had better be a pointer

```
Library Interface
// typedef _____* hdict_t;
hdict_t hdict_new(int capacity)
/*@requires capacity > 0;          @*/
/*@ensures \result != NULL;      @*/;
entry hdict_lookup(hdict_t D, key k)
/*@requires D != NULL;           @*/;
// ...
```

- It's the client who decides on what keys and entries are
 - the interface must tell the client to do this
- The interface has two parts
 - the **client interface**: what the client needs to supply to the library
 - the **library interface**: what the library provides to the client

Inserting an Entry

```
Client Interface
// typedef _____* entry;
// typedef _____ key;
// ...
```

```
Library Interface
// typedef _____* hdict_t;
hdict_t hdict_new(int capacity)
/* @requires capacity > 0;          @*/
/* @ensures \result != NULL;      @*/;

entry hdict_lookup(hdict_t D, key k)
/* @requires D != NULL;           @*/;

void hdict_insert(hdict_t D, entry e)
/* @requires D != NULL && e != NULL; @*/;
```

e cannot be NULL

- If NULL stands for an entry that was not found, no entry shall ever be NULL

What about all those Questions?

```
Client Interface
// typedef _____* entry;
// typedef _____key;
key entry_key(entry e)
/*@requires e != NULL;    @*/ ;
int key_hash(key k);
bool key_equiv(key k1, key k2);
```

Entries cannot be NULL

```
Library Interface
// typedef _____* hdict_t;
hdict_t hdict_new(int capacity)
/*@requires capacity > 0;    @*/
/*@ensures \result != NULL; @*/ ;
entry hdict_lookup(hdict_t D, key k)
/*@requires D != NULL;    @*/ ;
void hdict_insert(hdict_t D, entry e)
/*@requires D != NULL && e != NULL;    @*/ ;
```

- The library needs information from the client to do its job
 - Supply **functions** the library can use to find this information
 - a function that returns the key of an entry entry_key
 - a function that computes the hash value of a key key_hash
 - a function that determines whether two keys are the same key_equiv
 - Add their prototype in the client interface!

A Postcondition for `hdict_insert`

```
Client Interface
// typedef _____* entry;
// typedef _____ key;

key entry_key(entry e)
/*@requires e != NULL;    @*/;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

```
Library Interface
// typedef _____* hdict_t;

hdict_t hdict_new(int capacity)
/*@requires capacity > 0;    @*/
/*@ensures \result != NULL;  @*/;

entry hdict_lookup(hdict_t D, key k)
/*@requires D != NULL;    @*/;

void hdict_insert(hdict_t D, entry e)
/*@requires D != NULL && e != NULL;    @*/
/*@ensures hdict_lookup(D, entry_key(e)) == e; @*/;
```

- If we insert an entry and lookup its key, we should find that entry
 - i.e., `hdict_lookup(D, entry_key(e)) == e`
 - lookup returns the very entry `e`
 - not a different entry with the same data

e is a pointer

A Postcondition for `hdict_lookup`

```
Client Interface
// typedef _____* entry;
// typedef _____ key;

key entry_key(entry e)
/* @requires e != NULL;    @*/ ;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

```
Library Interface
// typedef _____* hdict_t;

hdict_t hdict_new(int capacity)
/* @requires capacity > 0;    @*/
/* @ensures \result != NULL;  @*/ ;

entry hdict_lookup(hdict_t D, key k)
/* @requires D != NULL;    @*/
/* @ensures \result == NULL
|| key_equiv(entry_key(\result), k);    @*/ ;

void hdict_insert(hdict_t D, entry e)
/* @requires D != NULL && e != NULL;    @*/
/* @ensures hdict_lookup(D, entry_key(e)) == e; @*/ ;
```

- If we look up a key
 - either we get back NULL
 - `\result == NULL`
 - or the key of the returned entry is our key
 - `key_equiv(entry_key(\result), k)`
- The client interface functions give us a way to write very precise postconditions

The Hash Dictionary Interface

Client Interface

```
// typedef _____* entry;
// typedef _____ key;

key entry_key(entry e)
/* @requires e != NULL;    @*/ ;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

What the library needs
from the client

Library Interface

```
// typedef _____* hdict_t;

hdict_t hdict_new(int capacity)
/* @requires capacity > 0;    @*/
/* @ensures \result != NULL;  @*/ ;

entry hdict_lookup(hdict_t D, key k)
/* @requires D != NULL;    @*/
/* @ensures \result == NULL
           || key_equiv(entry_key(\result), k); @*/ ;

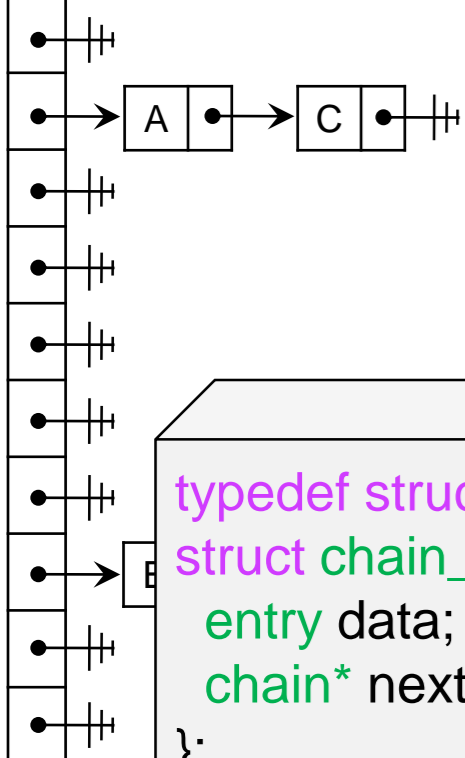
void hdict_insert(hdict_t D, entry e)
/* @requires D != NULL && e != NULL;    @*/
/* @ensures hdict_lookup(D, entry_key(e)) == e; @*/ ;
```

What the library provides
to the client

Hash Dictionary Implementation

```
Library Interface
// typedef _____* hdict_t;
// ...
```

Hash Dictionary Types



```
typedef struct chain_node chain;
struct chain_node {
    entry data; // data != NULL
    chain* next;
};

struct hdict_header {
    int size; // size >= 0
    int capacity; // capacity > 0
    chain*[] table; // \length(table) == capacity
};
typedef struct hdict_header hdict;

// ... rest of implementation

typedef hdict* hdict_t;
```

Implementation

These are expected constraints on the fields

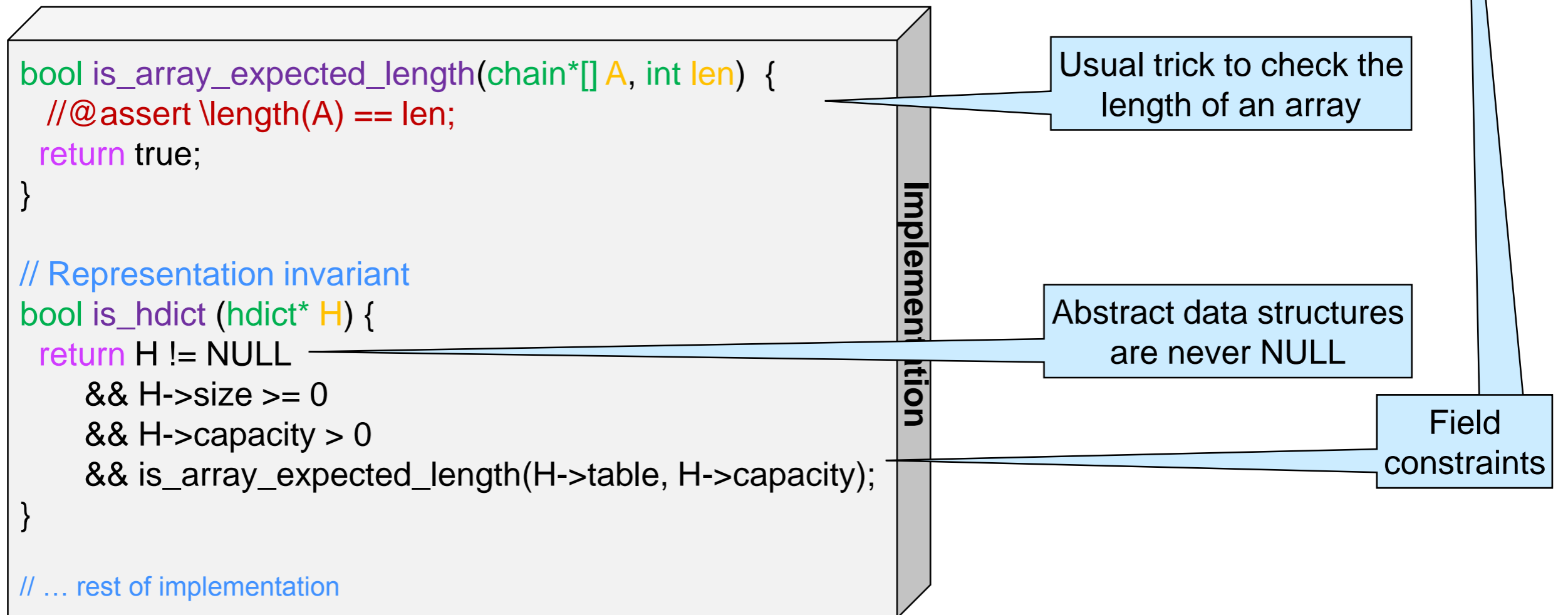
As usual, the abstract client type is a pointer to the concrete implementation type

- Each chain is a NULL-terminated linked list of entries
 - entries can't be NULL
- A dictionary is implemented as a hash table
- We need to keep track of
 - size: the number of entries
 - capacity: the length of the hash table
 - the hash table itself
 - an array of pointers to chain nodes

Representation Invariants

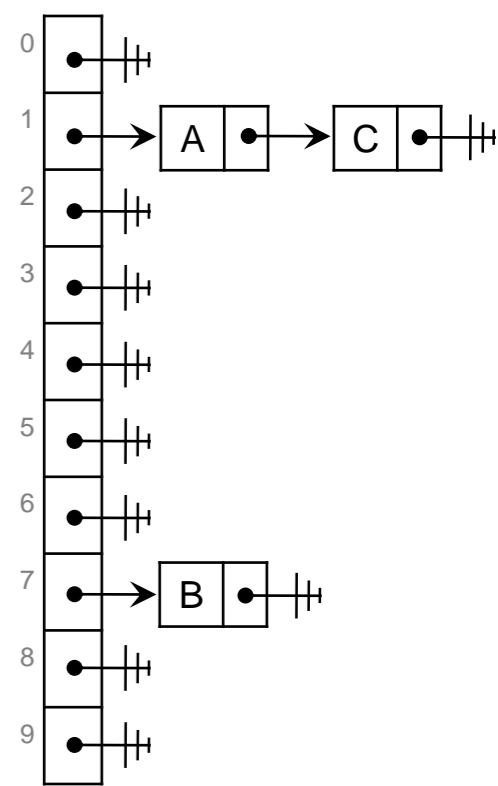
```
typedef struct hdict_header hdict;  
struct hdict_header {  
    int size;           // size >= 0  
    int capacity;      // capacity > 0  
    chain*[] table;    // \length(table) == capacity  
};
```

- We need to capture the field constraints in the type



More Representation Invariants

- Hash tables have a much more involved structure than previous concrete library types
 - the chains are acyclic
 - no two entries have the same key
 - each entry hashes to the right index
 - no entry is NULL
 - the number of entries equals the size field



```
// Representation invariant
bool is_hdict (hdict* H) {
    return H != NULL
        && H->size >= 0
        && H->capacity > 0
        && is_array_expected_length(H->table, H->capacity)
        && is_valid_hashtable(H);
}
```

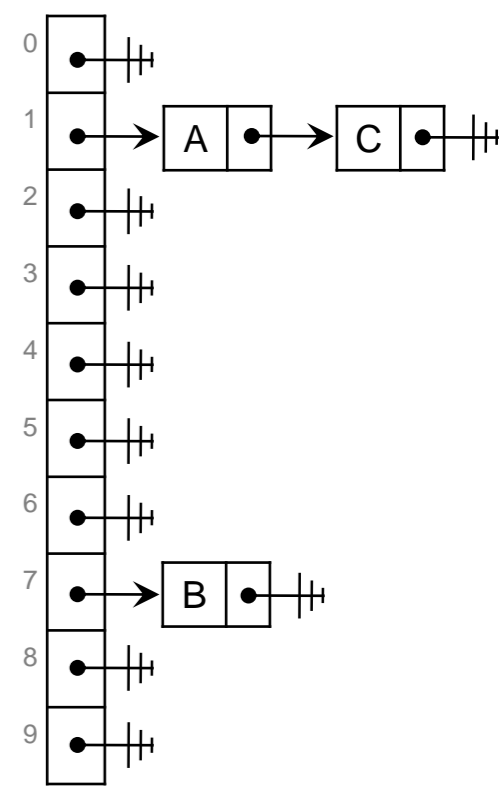
Implementation

This tests all these structural constraints

Exercise!

Invalidating Invariants

- The client can modify the keys **after** they have been inserted in the hash table
 - The chains contain **pointers** to entries
- This can invalidate the data structure invariants
 - `is_dict` fails the next time it is called
 - this is not because of a bug in the library
 - this is because the client manipulated the entries through aliases



- Aliasing is dangerous!

This couldn't happen in any of the data structures we studied so far

Implementing `hdict_lookup`

```
Library Interface
entry hdict_lookup(hdict_t D, key k)
/*@requires D != NULL; @*/
/*@ensures \result == NULL
|| key_equiv(entry_key(\result), k); @*/;
```

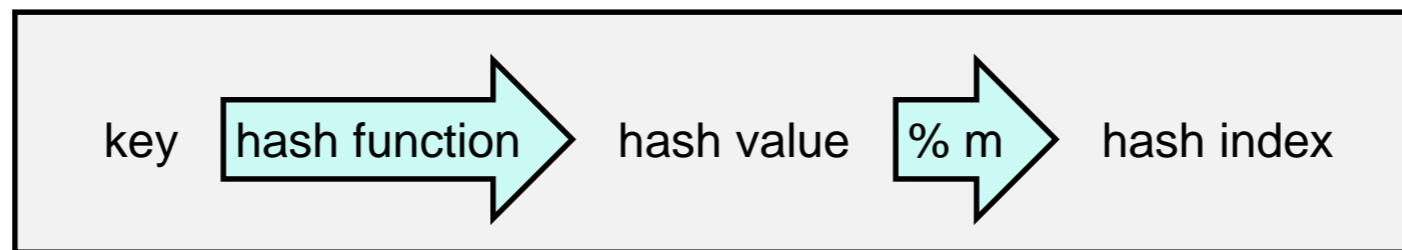
```
Client Interface
// typedef _____* entry;
// typedef _____ key;

key entry_key(entry e)
/*@requires e != NULL; @*/;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

- First we need to find the right bucket
 - determine the hash index of `k`



```
int i = key_hash(k) % D->capacity;
```

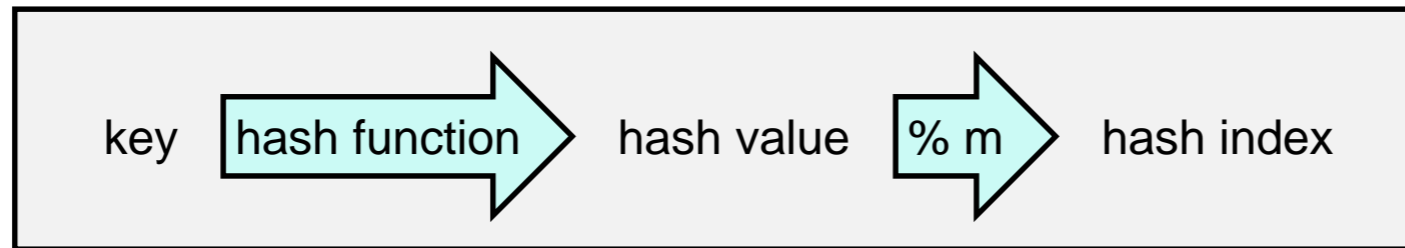
What's its hash value?

The library's questions are answered by the client interface functions

- This won't work if `hash_key(k)` is negative!

```
int i = abs(key_hash(k) % D->capacity);
```

Finding the Right Bucket



```
Client Interface
// typedef _____* entry;
// typedef _____ key;

key entry_key(entry e)
/*@requires e != NULL; @*/;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

➤ determine the hash index of **k**

```
int i = abs(key_hash(k) % H->capacity);
```

- We will need to do the same in **hdict_insert**
 - **factor it out** in a function that computes the hash index of a key

```
int index_of_key(hdict* H, key k)
/*@requires is_hdict(H);
/*@ensures 0 <= \result && \result < H->capacity;
{
    return abs(key_hash(k) % H->capacity);
}
```

Implementation

Implementing `hdict_lookup`

- First we need to find the right bucket
- Then we go through its chain
 - extract the key of each entry
 - check if it is equal to `k`

```

Client Interface
// typedef _____* entry;
// typedef _____ key;

key entry_key(entry e)
/*@requires e != NULL; @*/;

int key_hash(key k);

bool key_equiv(key k1, key k2);
    
```

```

Library Interface
entry hdict_lookup(hdict_t D, key k)
/*@requires D != NULL; @*/
/*@ensures \result == NULL
|| key_equiv(entry_key(\result), k); @*/;
    
```

What's its key?

Is it the same as `k`?

```

entry hdict_lookup(hdict* H, key k)
/*@requires is_hdict(H);
/*@ensures \result == NULL
|| key_equiv(entry_key(\result), k);
{
    int i = index_of_key(H, k);
    for (chain* p = H->table[i]; p != NULL; p = p->next) {
        if (key_equiv(entry_key(p->data), k))
            return p->data;
    }
    return NULL;
}
    
```

Implementation

H must satisfy the representation invariant

`i` is the hash index of `k`

This is the start of the chain

Return the entry if `k` is found ...

... and signal it's not there otherwise

Implementing `hdict_insert`

Client Interface

```
// typedef _____* entry;
// typedef _____ key;

key entry_key(entry e)
/*@requires e != NULL; @*/;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

Library Interface

```
void hdict_insert(hdict_t D, entry e)
/*@requires D != NULL && e != NULL; @*/
/*@ensures hdict_lookup(D, entry_key(e)) == e; @*/;
```

```
void hdict_insert(hdict* H, entry e)
/*@requires is_hdict(H) && e != NULL;
/*@ensures \ hdict_lookup(D, entry_key(e)) == e;
/*@ensures is_hdict(H);
```

```
{
    key k = entry_key(e);
    int i = index_of_key(H, k);
    for (chain* p = H->table[i]; p != NULL; p = p->next) {
        if (key_equiv(entry_key(p->data), k)) {
            p->data = e;
            return;
        }
    }
    chain* p = alloc(chain);
    p->data = e;
    p->next = H->table[i];
    H->table[i] = p;
    (H->size)++;
}
```

Implementation

H must remain valid after the insertion

Check if there is already an entry with the same key (similar code to `hdict_lookup`)

If so overwrite it

Otherwise, prepend a new node containing `e`

Implementing `hdict_new`

```
Client Interface  
  
// typedef _____* entry;  
// typedef _____ key;  
  
key entry_key(entry e)  
  /*@requires e != NULL; @*/;  
  
int key_hash(key k);  
  
bool key_equiv(key k1, key k2);
```

```
Library Interface  
  
hdict_t hdict_new(int capacity)  
  /*@requires capacity > 0; @*/  
  /*@ensures \result != NULL; @*/;
```

```
hdict* hdict_new(int capacity)  
  /*@requires capacity > 0;  
  /*@ensures is_hdict(\result);  
{  
  hdict* H = alloc(hdict);  
  H->size = 0;  
  H->capacity = capacity;  
  H->table = alloc_array(chain*, capacity);  
  return H;  
}
```

Implementation

Returned dictionary must be valid

Initialized to default pointer value, NULL

The Hash Dictionary Library

Overall Implementation

Implementation

```

Client Interface
// typedef _____* entry;
// typedef _____ key;

key entry_key(entry e)
/*@requires e != NULL; @*/;

int key_hash(key k);

bool key_equiv(key k1, key k2);
    
```

```

Library Interface
// typedef _____* hdict_t;

hdict_t hdict_new(int capacity)
/*@requires capacity > 0; @*/
/*@ensures \result != NULL; @*/;

entry hdict_lookup(hdict_t D, key k)
/*@requires D != NULL; @*/
/*@ensures \result == NULL
|| key_equiv(entry_key(\result), k); @*/;

void hdict_insert(hdict_t D, entry e)
/*@requires D != NULL && e != NULL; @*/
/*@ensures hdict_lookup(D, entry_key(e)) == e; @*/;
    
```

```

// Implementation-side types
typedef struct chain_node chain;
struct chain_node {
    entry data; // data != NULL
    chain* next;
};
struct hdict_header {
    int size; // size >= 0
    int capacity; // capacity > 0
    chain*[] table; // \length(table) == capacity
};
typedef struct hdict_header hdict;

// Representation invariant
bool is_hdict (hdict* H) {
    return H != NULL
    && H->size >= 0
    && H->capacity > 0
    && is_array_expected_length(H->table, H->capacity)
    && is_valid_hashtable(H);
}

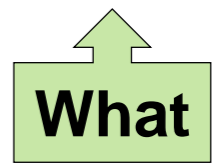
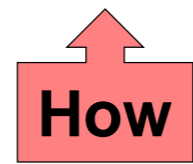
// Implementation of interface functions
int index_of_key(hdict* H, key k)
/*@requires is_hdict(H);
/*@ensures 0 <= \result && \result < H->capacity;
{
    return abs(key_hash(k) % H->capacity);
}

entry hdict_lookup(hdict* H, key k)
/*@requires is_hdict(H);
/*@ensures \result == NULL
|| key_equiv(entry_key(\result), k);
{
    int i = index_of_key(H, k);
    for (chain* p = H->table[i]; p != NULL; p = p->next)
        if (key_equiv(entry_key(p->data), k))
            return p->data;
    return NULL;
}

void hdict_insert(hdict* H, entry e)
/*@requires is_hdict(H) && e != NULL;
/*@ensures \hdict_lookup(D, entry_key(e)) == e;
/*@ensures is_hdict(H);
{
    key k = entry_key(e);
    int i = index_of_key(H, k);
    for (chain* p = H->table[i]; p != NULL; p = p->next) {
        if (key_equiv(entry_key(p->data), k)) {
            p->data = e;
            return;
        }
    }
    chain* p = alloc(chain);
    p->data = e;
    p->next = H->table[i];
    H->table[i] = p;
    (H->size)++;
}

hdict* hdict_new(int capacity)
/*@requires capacity > 0;
/*@ensures is_hdict(\result);
{
    hdict* H = alloc(hdict);
    H->size = 0;
    H->capacity = capacity;
    H->table = alloc_array(chain*, capacity);
    return H;
}

// Client type
typedef hdict* hdict_t;
    
```



Complex Libraries

- The hash dictionary library is a **complex library**
 - it needs the client to supply code and functions
 - so that it can provide its services
- Complex libraries consist of
 - a **client interface**
 - an **implementation**
 - a **library interface**
- The client sees the client and library interfaces
 - but not the implementation

Stacks and queues were **elementary libraries**

They consisted of only an implementation and a library interface

Their client only saw the library interface

Structure of a Complex C0 Library File

NEW

```
/****** CLIENT INTERFACE *****/
// typedef _____ *entry;
...
key entry_key(entry e)
/*@requires e != NULL;          @*/;
...

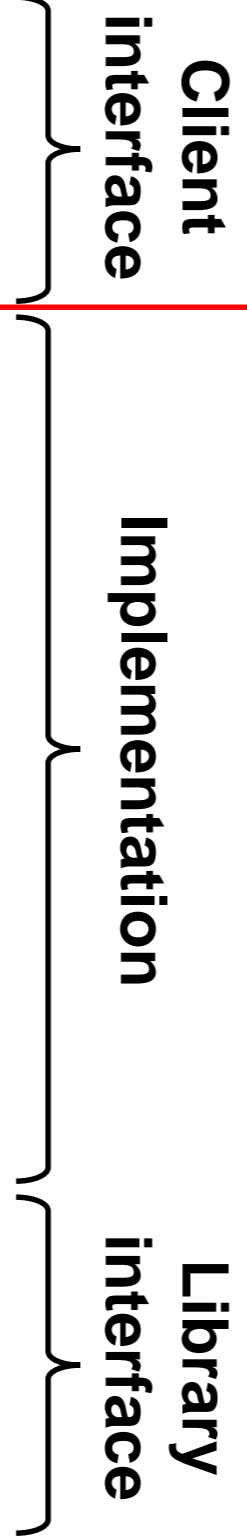
/****** IMPLEMENTATION *****/
// Implementation-side types
struct hdict_header {...};
typedef struct hdict_header hdict;

// Representation invariant
bool is_hdict(hdict* H) { ... }

// Implementation of interface functions
entry hdict_lookup(hdict* H, key k)
/*@requires is_hdict(H);        @*/
/*@ensures ....                @*/
{ ... }
...
// Client type
typedef hdict* hdict_t;

/****** LIBRARY INTERFACE *****/
// typedef _____ *hdict_t;

entry hdict_lookup(hdict_t D, key k)
/*@requires D != NULL;         @*/
/*@ensures ....                @*/;
...
```



- Client interface
 - Client type names
 - Prototype of client functions
- Implementation
 - Concrete type definition
 - Representation invariant function
 - Implementation of interface functions
 - Actual abstract type definition
- Library interface
 - Abstract type name
 - Prototype of exported functions

Structure of a Complex C0 Library File

```
/****** CLIENT INTERFACE *****/
// typedef _____ *entry;
...
key entry_key(entry e)
/*@requires e != NULL;          @*/;
...
/****** IMPLEMENTATION *****/
// Implementation-side types
struct hdict_header {...};
typedef struct hdict_header hdict;

// Representation invariant
bool is_hdict(hdict* H) { ... }

// Implementation of interface functions
entry hdict_lookup(hdict* H, key k)
/*@requires is_hdict(H);        @*/
/*@ensures .....                @*/
{ ... }
...
// Client type
typedef hdict* hdict_t;

/****** LIBRARY INTERFACE *****/
// typedef _____ *hdict_t;

entry hdict_lookup(hdict_t D, key k)
/*@requires D != NULL;          @*/
/*@ensures .....                @*/;
...
```

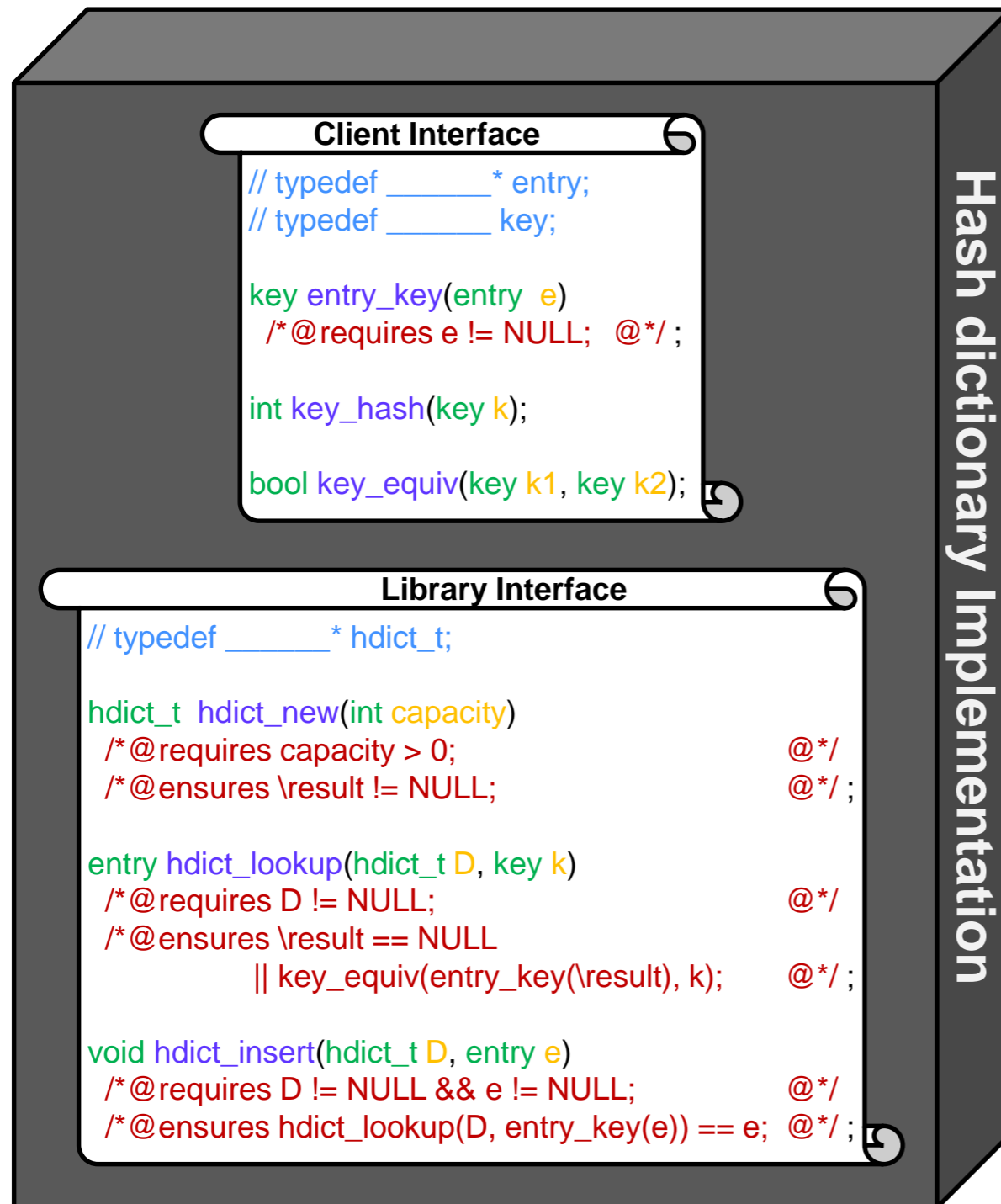
The diagram shows three sections of code grouped by brackets on the right:

- Client interface**: The top section, containing the client interface definition.
- Implementation**: The middle section, containing the implementation of the client interface functions and the client type.
- Library interface**: The bottom section, containing the library interface definition.

- By convention,
 - the client interface is on top
 - because the implementation uses the types and functions it mentions
 - the implementation is in the middle
 - it relies on the concrete client definitions
 - it ends with the definition of the abstract client type
 - the library interface is at the bottom
 - it only mentions the abstract types

Using the Library

Using the Hash Dictionary Library



- The client needs to **define** the types and functions listed in **the client interface**
- It can **use** the types and functions exported by **the library implementation**
- The client must not rely on the implementation details

Implementing our Example

You are the new produce manager of the local grocery store. You want to use a dictionary to track your fruit inventory.

- Defining the **types** requested in the client interface
 - **entries** are inventory items consisting of a fruit and a quantity
 - the fruit name is the **key**

```
// What the client wants to store in the dictionary
struct inventory_item {
    string fruit;    // key
    int quantity;
};
```

```
/****** Fulfilling the library interface *****/
typedef struct inventory_item* entry;
typedef string key;
```

This is the concrete definition of
// typedef _____* entry;

This is the concrete definition of
// typedef _____ key;

Implementing our Example

You are the new produce manager of the local grocery store. You want to use a dictionary to track your fruit inventory.

- Defining the **functions** requested in client interface

```
/****** Fulfilling the library interface *****/  
key entry_key(entry e)  
//@requires e != NULL;  
{  
    return e->fruit;  
}  
  
bool key_equiv(key k1, key k2) {  
    return string_equal(k1, k2);  
}  
  
int key_hash(key k) {  
    return lcg_hash_string(k);  
}
```

The key is the fruit field of an inventory item

Two fruit are the same if they have the same name (`string_equals` is defined in `<string>`)

`lcg_hash_string` is a good hash function on strings

Client Interface Implementation

Here's the full definition of `lcg_hash_string`

- This defines every type and function in the client interface

```
Client Interface
// typedef _____* entry;
// typedef _____ key;

key entry_key(entry e)
/*@requires e != NULL; @*/;

int key_hash(key k);

bool key_equiv(key k1, key k2);
```

- We store this code in a file called **produce.c0**

Client definition file

```
#use <string>

int lcg_hash_string(string s) {
    int len = string_length(s);
    int h = 0;
    for (int i = 0; i < len; i++) {
        h = h + char_ord(string_charat(s, i));
        h = 1664525 * h + 1013904223;
    }
    return h;
}

// What the client wants to store in the dictionary
struct inventory_item {
    string fruit; // key
    int quantity;
};

/***** Fulfilling the library interface *****/
typedef struct inventory_item* entry;
typedef string key;

key entry_key(entry e)
/*@requires e != NULL;
{
    return e->fruit;
}

bool key_equiv(key k1, key k2) {
    return string_equal(k1, k2);
}

int key_hash(key k) {
    return lcg_hash_string(k);
}
```

Implementing our Example

You are the new produce manager of the local grocery store. You want to use a dictionary to track your fruit inventory.

- We can now implement the inventory application that uses hash dictionaries

- ✓ new dictionary
- ✓ insert A = ("apple", 20)
- ✓ insert B = ("banana", 10)
- ✓ insert C = ("pumpkin", 50)
- ✓ look up "apple"
- ✓ look up "lime"
- ✓ insert D = ("banana", 20)

- We store this code in a file called **produce-main.c0**

Client application file

```
struct inventory_item* make_inventory_item(string fruit, int quantity) {
    struct inventory_item* x = alloc(struct inventory_item);
    x->fruit = fruit;
    x->quantity = quantity;
    return x;
}

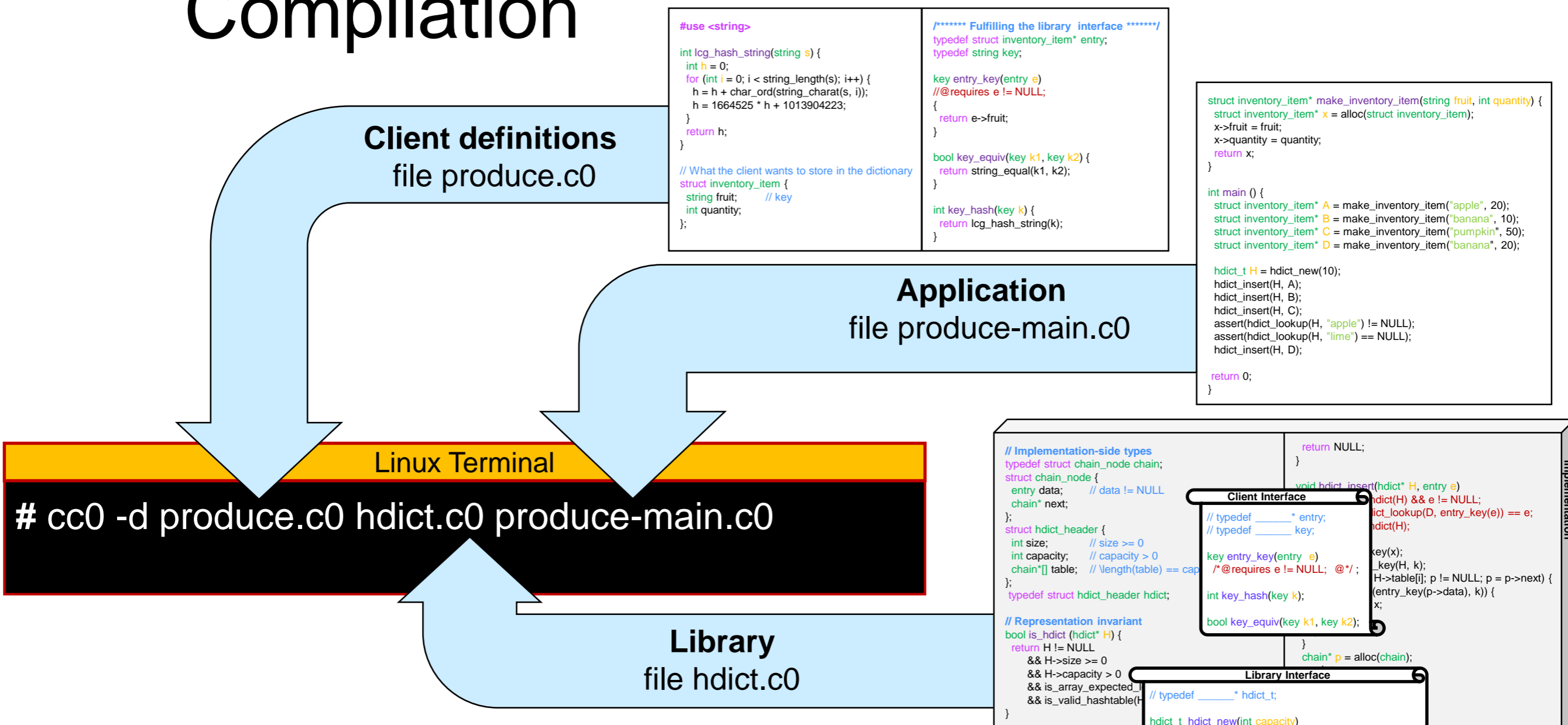
int main () {
    struct inventory_item* A = make_inventory_item("apple", 20);
    struct inventory_item* B = make_inventory_item("banana", 10);
    struct inventory_item* C = make_inventory_item("pumpkin", 50);
    struct inventory_item* D = make_inventory_item("banana", 20);

    hdict_t H = hdict_new(10);
    hdict_insert(H, A);
    hdict_insert(H, B);
    hdict_insert(H, C);
    assert(hdict_lookup(H, "apple") != NULL);
    assert(hdict_lookup(H, "lime") == NULL);
    hdict_insert(H, D);

    return 0;
}
```

Function that creates inventory items

Compilation



```
#use <string>

int lcg_hash_string(string s) {
    int h = 0;
    for (int i = 0; i < string_length(s); i++) {
        h = h + char_ord(string_charat(s, i));
        h = 1664525 * h + 1013904223;
    }
    return h;
}

// What the client wants to store in the dictionary
struct inventory_item {
    string fruit; // key
    int quantity;
};
```

```
/****** Fulfilling the library interface *****/
typedef struct inventory_item* entry;
typedef string key;

key entry_key(entry e)
/*@requires e != NULL;
*/
{
    return e->fruit;
}

bool key_equiv(key k1, key k2) {
    return string_equal(k1, k2);
}

int key_hash(key k) {
    return lcg_hash_string(k);
}
```

```
struct inventory_item* make_inventory_item(string fruit, int quantity) {
    struct inventory_item* x = alloc(struct inventory_item);
    x->fruit = fruit;
    x->quantity = quantity;
    return x;
}

int main () {
    struct inventory_item* A = make_inventory_item("apple", 20);
    struct inventory_item* B = make_inventory_item("banana", 10);
    struct inventory_item* C = make_inventory_item("pumpkin", 50);
    struct inventory_item* D = make_inventory_item("banana", 20);

    hdict_t H = hdict_new(10);
    hdict_insert(H, A);
    hdict_insert(H, B);
    hdict_insert(H, C);
    assert(hdict_lookup(H, "apple") != NULL);
    assert(hdict_lookup(H, "lime") == NULL);
    hdict_insert(H, D);

    return 0;
}
```

```
Linux Terminal
# cc0 -d produce.c0 hdict.c0 produce-main.c0
```

```
// Implementation-side types
typedef struct chain_node chain;
struct chain_node {
    entry data; // data != NULL
    chain* next;
};
struct hdict_header {
    int size; // size >= 0
    int capacity; // capacity > 0
    chain[] table; // \length(table) == capacity
};
typedef struct hdict_header hdict;

// Representation invariant
bool is_hdict (hdict* H) {
    return H != NULL
        && H->size >= 0
        && H->capacity > 0
        && is_array_expected(H->table)
        && is_valid_hashtable(H->table);
}

// Implementation of interface
int index_of_key(hdict* H, key k)
/*@requires is_hdict(H);
  @ensures 0 <= \result &&
  \result < H->capacity;
*/
{
    return abs(key_hash(k) % H->capacity);
}

entry hdict_lookup(hdict* H, key k)
/*@requires is_hdict(H);
  @ensures \result == NULL
  || key_equiv(entry_key(\result), k);
*/
{
    int i = index_of_key(H, k);
    for (chain* p = H->table[i]; p != NULL; p = p->next)
        if (key_equiv(entry_key(p->data), k))
            return p->data;
    return NULL;
}

void hdict_insert(hdict* H, entry e)
/*@requires H != NULL;
  @requires hdict_lookup(D, entry_key(e)) == e;
*/
{
    chain* p = alloc(chain);
    p->data = e;
    p->next = NULL;
    if (H->table[H->size] == NULL)
        H->table[H->size] = p;
    else
        H->table[H->size] = p->next;
    H->table[H->size] = p;
    H->size++;
}

// Client type
typedef hdict* hdict_t;
```

- The definition file comes **before** the library
 - the library needs the definitions it supplies
- The library comes **before** the application file
 - the application needs the functionalities it provides

Compilation

Linux Terminal

```
# cc0 -d produce.c0 hdict.c0 produce-main.c0
```

- *The definition file comes **before** the library*
 - *the library needs the definitions it supplies*
- *The library comes **before** the application file*
 - *the application needs the functionalities it provides*
- The client must split the application code into two files
 - This leads to an unnatural compilation pattern
 - We would like to compile the hash dictionary library just the way we compile a stack library



We will address this shortly

Hash Sets

Towards an Interface

- keys = entries
 - these are the elements of the set
 - a single type **elem** replaces **key** and **entry**
- lookup can simply return true or false
 - this now checks set membership
 - return type is **bool**
 - no need to signal “not found” in a special way
 - **elem** does not have to be a pointer type

What about Sets?

- A **set** can be understood as a special case of a dictionary
 - keys = entries
 - these are the elements of the set
 - **lookup** can simply return true or false
 - this now checks set membership
- A set implemented as a hash dictionary is called a **hash set**

The Hash Set Interface

Client Interface

```
// typedef _____ elem;  
  
int key_hash(elem k);  
  
bool key_equiv(elem k1, elem k2);
```

Library Interface

```
// typedef _____* hset_t;  
  
hset_t hset_new(int capacity)  
/*@requires capacity > 0;          @*/  
/*@ensures \result != NULL;       @*/;  
  
bool hset_contains(hset_t S, elem e)  
/*@requires S != NULL;            @*/;  
  
void hset_insert(hset_t S, elem e)  
/*@requires S != NULL;            @*/  
/*@ensures hset_contains(S, e);   @*/;
```

- A single type **elem** replaces **key** and **entry**
 - it does not need to be a pointer
- lookup checks membership
 - renamed **hset_contains**
 - it returns a **bool**
- Everything else remains the same

*The implementation
is left as exercise*