Project 2-1: Hash Table Implementation and Test

Topics:

- Interface Specification: hash_table.h
- Implementation: hash_table.c
  - public interface functions vs private helper functions
  - “static” keyword
  - HashTable data structure
  - function pointers
- Guide for incremental design&test: P2-1-incremental design and test.pdf

header file (see hash_table.h)

- `#includes` import definitions from other files
- `#ifndef name`

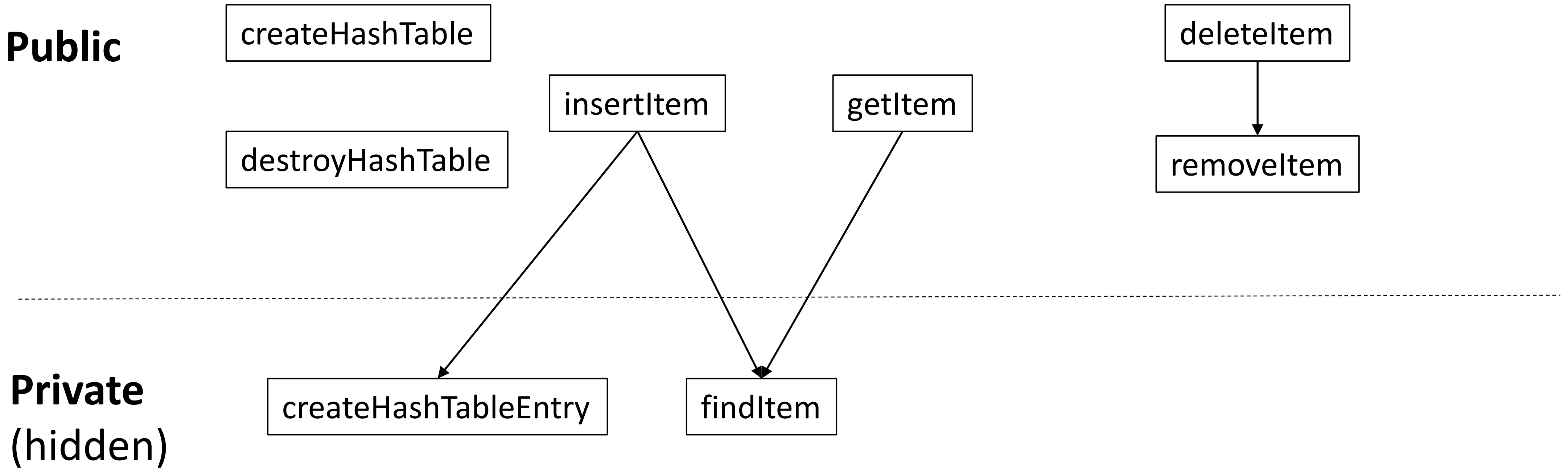
  `#define name`

  . . . // contents of .h file

  `#endif`

  $\Bigg\}$ guards against loading in declarations >1 time
- typedefs
  - HashFunction – a function pointer (will discuss later)
  - nicknames for structs
- function prototypes – define I/O params and types for public functions, and documentation of behavior

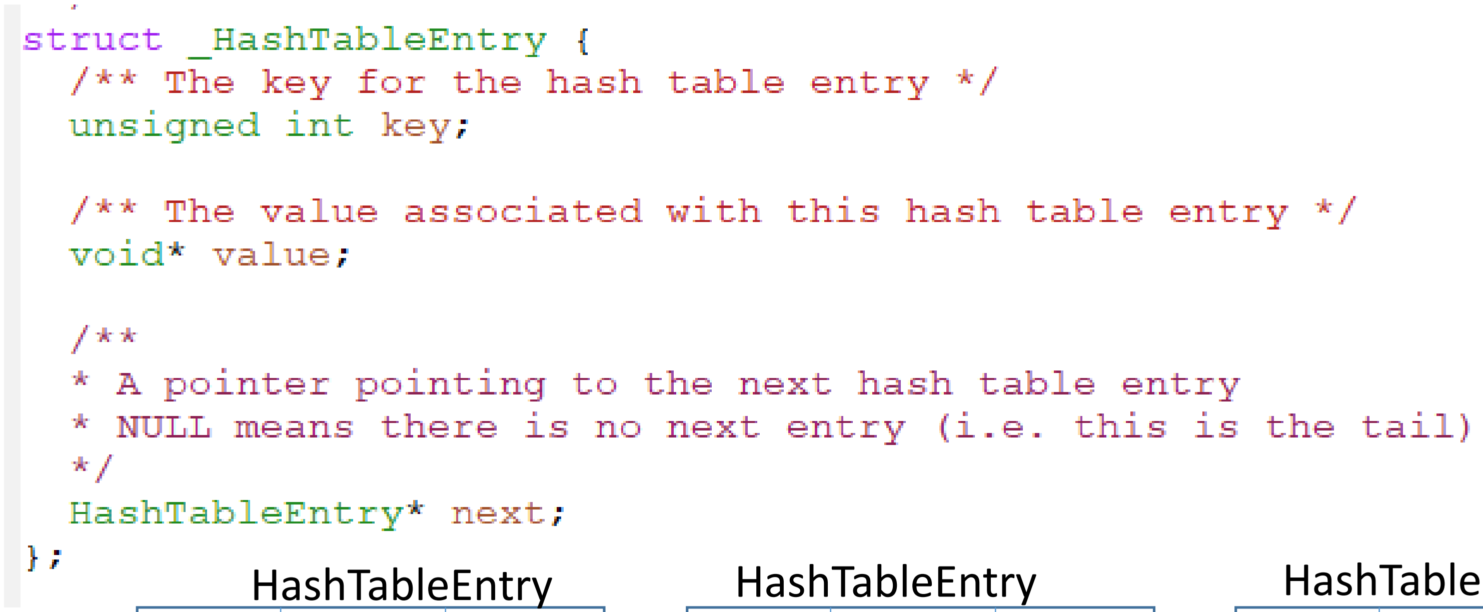
hash_table Functions and Caller/callee Relationships



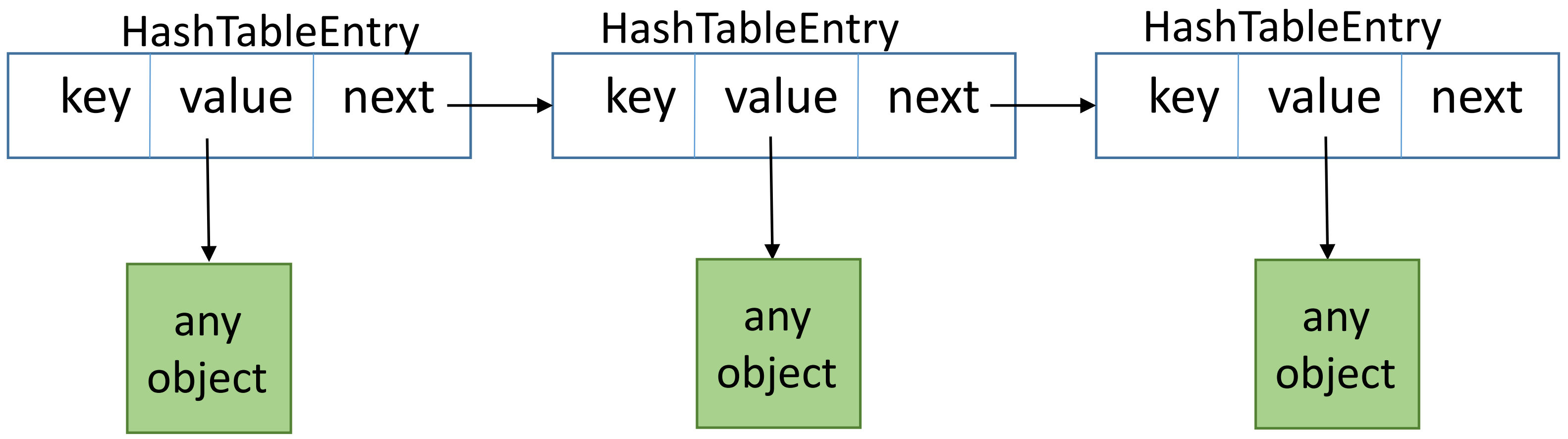
In hash_table.c private definitions preceded by “static” keyword – restricts access to the function only to callers in the file where it is defined.

HashTableEntry

```
typedef struct _HashTableEntry HashTableEntry;
```

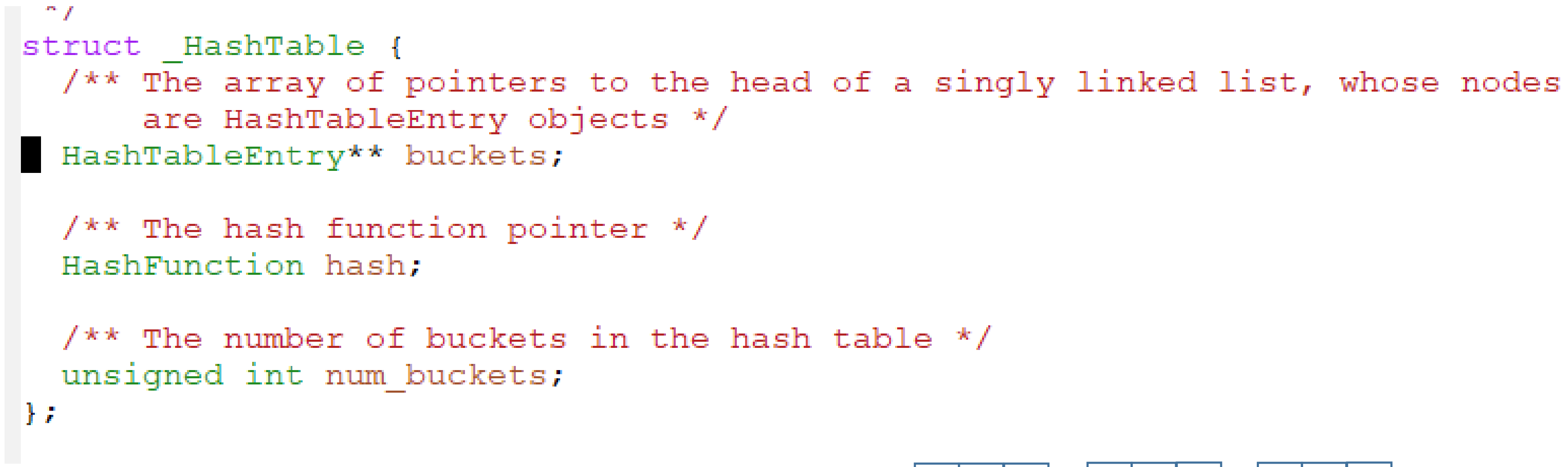


```
struct _HashTableEntry {
  /** The key for the hash table entry */
  unsigned int key;

  /** The value associated with this hash table entry */
  void* value;

  /**
  * A pointer pointing to the next hash table entry
  * NULL means there is no next entry (i.e. this is the tail)
  */
  HashTableEntry* next;
};
```

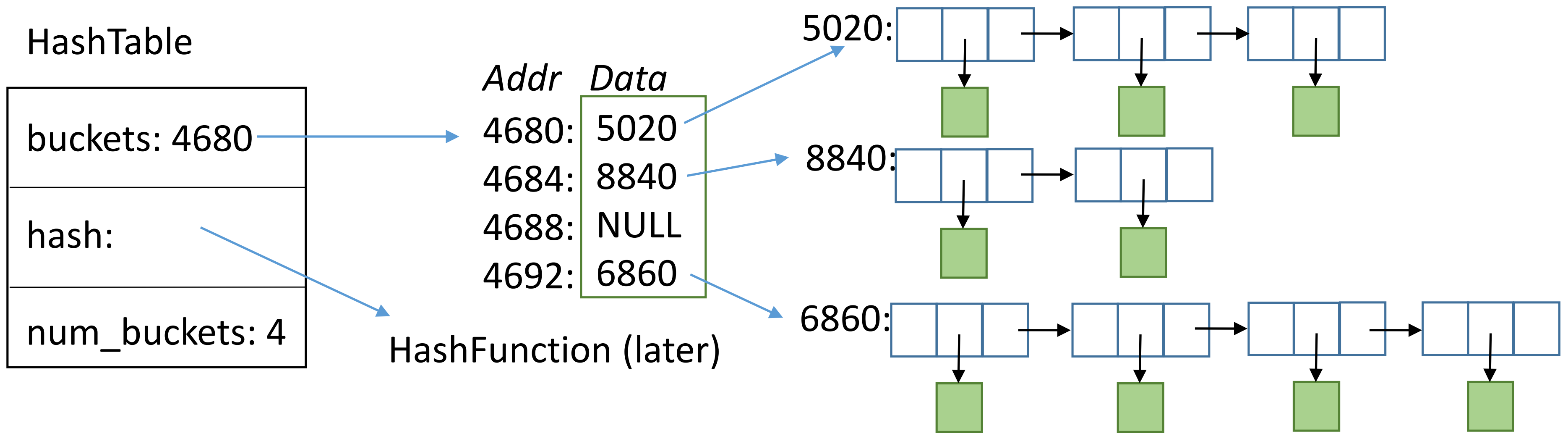


HashTable

```c
typedef struct _HashTable HashTable;
```

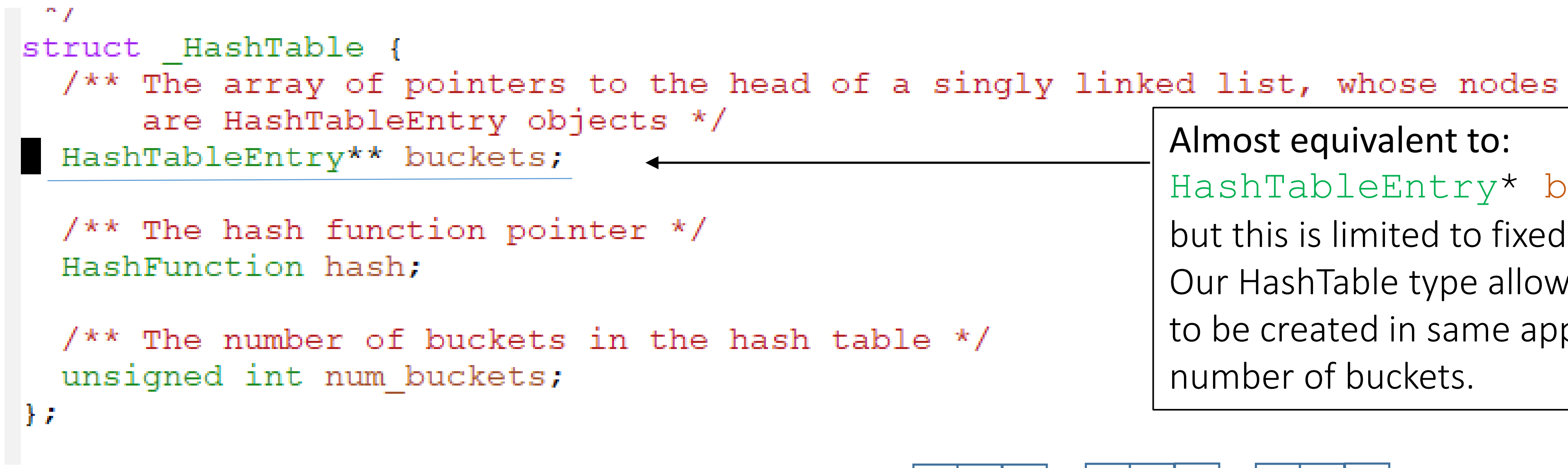


```c
 */
struct _HashTable {
  /** The array of pointers to the head of a singly linked list, whose nodes
      are HashTableEntry objects */
  HashTableEntry** buckets;

  /** The hash function pointer */
  HashFunction hash;

  /** The number of buckets in the hash table */
  unsigned int num_buckets;
};
```



HashTable

```
typedef struct _HashTable HashTable;
```

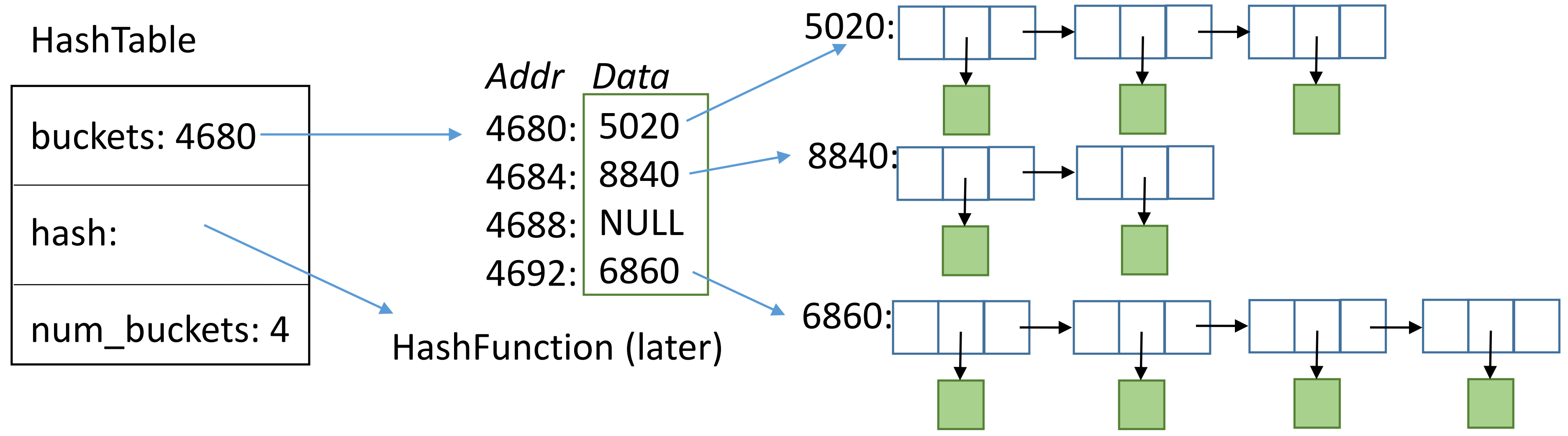


```
 */
struct _HashTable {
  /** The array of pointers to the head of a singly linked list, whose nodes
      are HashTableEntry objects */
  HashTableEntry** buckets;

  /** The hash function pointer */
  HashFunction hash;

  /** The number of buckets in the hash table */
  unsigned int num_buckets;
};
```

Almost equivalent to:

```
HashTableEntry* buckets[4];
```

but this is limited to fixed number of buckets.
Our HashTable type allows multiple HashTables to be created in same application with different number of buckets.



Allocating and initialize new HashTable

```c
// The createHashTable is provided for you as a starting point.
HashTable* createHashTable(HashFunction hashFunction, unsigned int numBuckets) {
  // The hash table has to contain at least one bucket. Exit gracefully if
  // this condition is not met.
  if (numBuckets==0) {
    printf("Hash table has to contain at least 1 bucket...\n");
    exit(1);
  }

  // Allocate memory for the new HashTable struct on heap.
  HashTable* newTable = (HashTable*)malloc(sizeof(HashTable));

  // Initialize the components of the new HashTable struct.
  newTable->hash = hashFunction;
  newTable->num_buckets = numBuckets;
  newTable->buckets = (HashTableEntry**)malloc(numBuckets*sizeof(HashTableEntry*));

  // As the new buckets contain indeterminant values, init each bucket as NULL.
  unsigned int i;
  for (i=0; i<numBuckets; ++i) {
    newTable->buckets[i] = NULL;
  }

  // Return the new HashTable struct.
  return newTable;
}
```

What is # bytes passed to malloc here?

What about here?

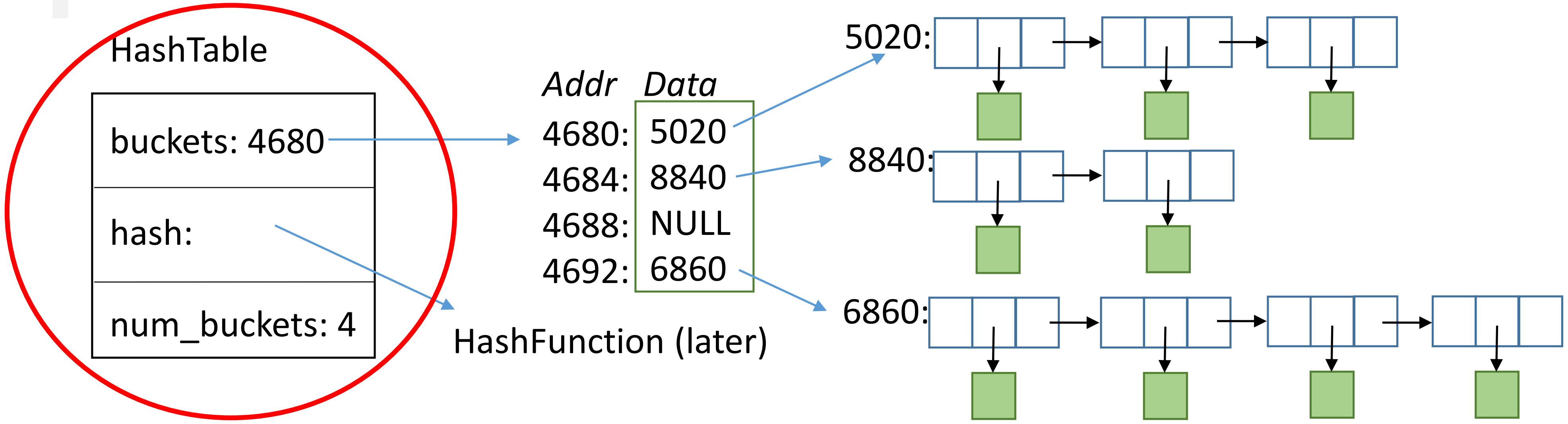
What does this loop do?

HashTable

```
typedef struct _HashTable HashTable;
```

```
 */
struct _HashTable {
  /** The array of pointers to the head of a singly linked list, whose nodes
      are HashTableEntry objects */
  HashTableEntry** buckets;

  /** The hash function pointer */
  HashFunction hash;

  /** The number of buckets in the hash table */
  unsigned int num_buckets;
};
```

`sizeof(HashTable):`
|ptr| + |ptr| + |int|
= 12 bytes (if 32-bit sys)



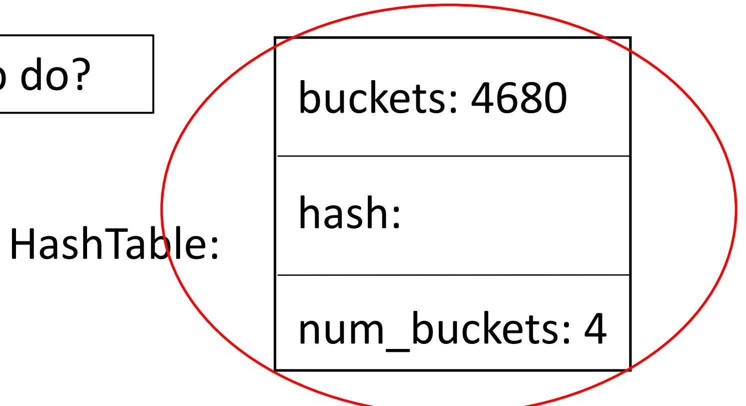
Allocating and initialize new HashTable

```c
// The createHashTable is provided for you as a starting point.
HashTable* createHashTable(HashFunction hashFunction, unsigned int numBuckets) {
  // The hash table has to contain at least one bucket. Exit gracefully if
  // this condition is not met.
  if (numBuckets==0) {
    printf("Hash table has to contain at least 1 bucket...\n");
    exit(1);
  }

  // Allocate memory for the new HashTable struct on heap.
  HashTable* newTable = (HashTable*)malloc(sizeof(HashTable));

  // Initialize the components of the new HashTable struct.
  newTable->hash = hashFunction;
  newTable->num_buckets = numBuckets;
  newTable->buckets = (HashTableEntry**)malloc(numBuckets*sizeof(HashTableEntry*));

  // As the new buckets contain indeterminant values, init each bucket as NULL.
  unsigned int i;
  for (i=0; i<numBuckets; ++i) {
    newTable->buckets[i] = NULL;
  }

  // Return the new HashTable struct.
  return newTable;
}
```

What is # bytes passed to malloc here?
Answer: 12

What about here?

What does this loop do?



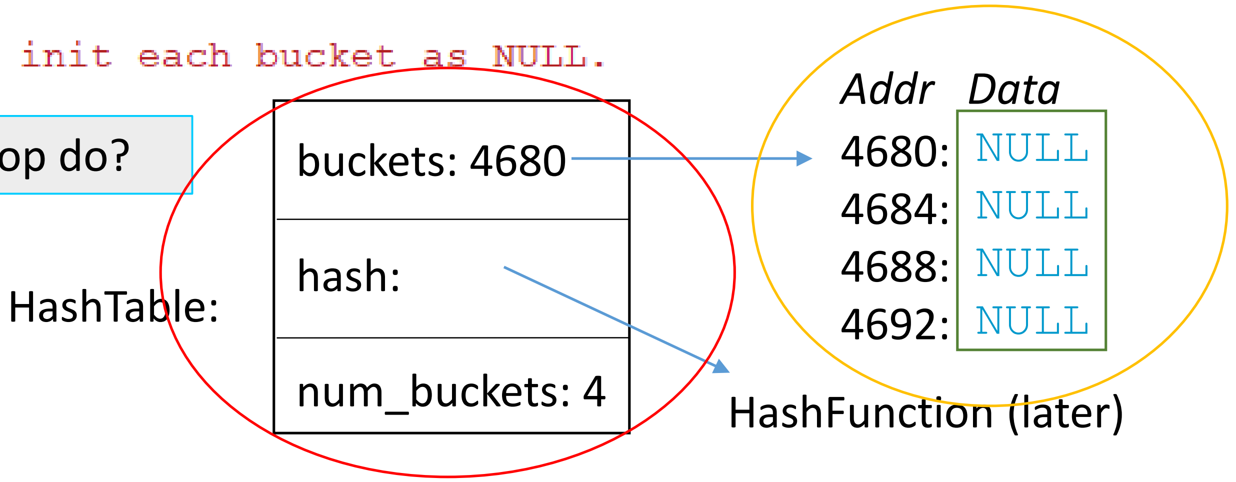
Allocating and initialize new HashTable

```
// The createHashTable is provided for you as a starting point.
HashTable* createHashTable(HashFunction hashFunction, unsigned int numBuckets) {
  // The hash table has to contain at least one bucket. Exit gracefully if
  // this condition is not met.
  if (numBuckets==0) {
    printf("Hash table has to contain at least 1 bucket...\n");
    exit(1);
  }

  // Allocate memory for the new HashTable struct on heap.
  HashTable* newTable = (HashTable*)malloc(sizeof(HashTable));

  // Initialize the components of the new HashTable struct.
  newTable->hash = hashFunction;
  newTable->num_buckets = numBuckets;
  newTable->buckets = (HashTableEntry**)malloc(numBuckets*sizeof(HashTableEntry*));

  // As the new buckets contain indeterminant values, init each bucket as NULL.
  unsigned int i;
  for (i=0; i<numBuckets; ++i) {
    newTable->buckets[i] = NULL;
  }

  // Return the new HashTable struct.
  return newTable;
}
```

What is # bytes passed to malloc here?
Answer: 12

What about here?
Answer: numBuckets * |ptr|
(e.g., 4*4 = 16)

What does this loop do?



Function Pointers!

```
 /* This defines a type that is a pointer to a function which takes
  * an unsigned int argument and returns an unsigned int value.
  * The name of the type is "HashFunction".
  */
typedef unsigned int (*HashFunction)(unsigned int key);
```

optional arg name

return value type

type nickname

input parameter types,
separated by commas if >1 parameter

Example Use: Application w/ >1 HashTable

```
#define num_moon_categories 10;
#define num_planet_categories 7;
int hash_m(int key){
  return(key*key) % num_moon_categories);
}
int hash_p(int key){
  return(key % num_planet_categories);
}
HashTable* moonDatabase = createHashTable(hash_m, num_moons_categories);
HashTable* planetDatabase = createHashTable(hash_p, num_planet_categories);
moonDatabase->hash(20);   // 0
planetDatabase->hash(20); // 6
```

A pointer variable can hold an address of location in instruction memory (not just data memory).

Function pointer's value can be applied just like any function call.

HashTable Functions Start w/ Hash Function Call

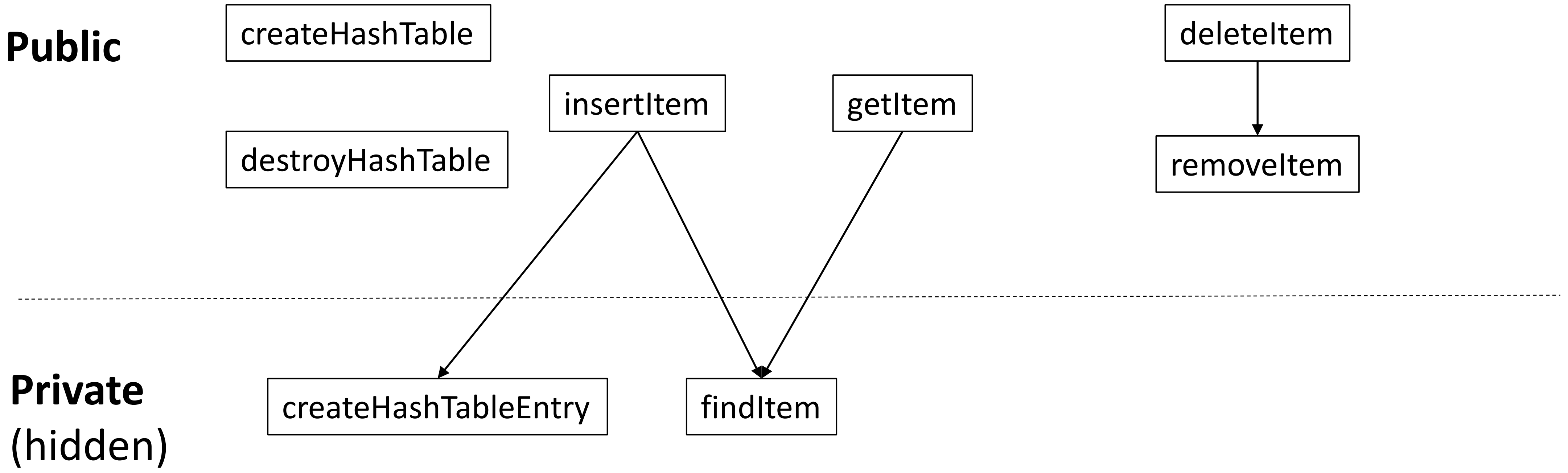
Lookup(HT, key):

1. hash(key) => index into bucket array
2. loop thru bucket list, look for match to key

```
HashTableEntry* Lookup(HashTable* myHashTable, unsigned int key) {
  unsigned int bucketNum = myHashTable->hash(key);  // Get the bucket number.
  HashTableEntry* temp = myHashTable->buckets[bucketNum]; // Get the head entry.

  while (temp!=NULL) {
    if (temp->key == key) return temp; // Return hash table entry if key is found.
    temp = temp->next;  // Otherwise, move to next node.
  }
  return NULL;  // Return NULL if key is not present.
}
```

Typical Associative Search pattern

hash_table Functions and Caller/callee Relationships



Guide for incremental design&test: P2-1-incremental design and test.pdf
gives steps for coding and testing P2-1 w/ support of gtest.