CSCI E-50 WEEK 6

TERESA LEE
[teresa@cs50.net]
March 5 2018

TEST

- Released friday March 09th at noon
- Due monday March 12th at noon
- Weeks 0 through 5 (and Problem Sets 0 through 5)
- About Test
- Past Spring 2017 Test

RESOURCES

- Lecture videos
- Scribe notes
- Reference sheets
- Walkthroughs
- Shorts
- Reference50
- Books

Instructions

you may

- browse and search the Internet,
- email the course's heads at heads@cs50.harvard.edu with questions,
- review books,
- review questions and answers already posted on CS50 Discourse,
- review the course's own materials, and
- use CS50 IDE, but

you may not

- receive or solicit directly or indirectly any help from anyone other than the course's heads.

Take care to review the course's policy on academic honesty in its entirety. Note particularly, but not only, that

- looking at another individual's work during the test is **not reasonable** and
- turning to humans (besides the course's heads) for help or receiving help from humans (besides the course's heads) during the test is **not reasonable**.

Test Content

Week 0

- Binary
- ASCII
- Bytes
- Algorithms
- Scratch

Week 1

- Loops
- Conditions
- Variables
- Compiling
- Data Types
- Overflow
- Imprecision

Test Content

Week 2

- Bugs
- Cryptography
- Strings (Arrays)
- Typecasting
- Reference Tools
- Command Line Arguments

Week 3

- Arrays
- Searching
- Sorting
- Time Complexity
- Recursion

Test Content

Week 4

- Call stack
- Pointers
- Strings
- Dynamic Memory Allocation
- Valgrind
- Buffer overflow

Week 5

- Structures
- Linked lists
- Hash tables
- Tries
- Stacks
- Queues

Week 0 & 1

Supersection (Fall 2017) Slides & Video

Week 2

Example

`ex1_verifier.c`

Week 3

Computational Complexity

- Complexity? Time & Space
- Algorithm's running time
  - O (upper bound)
  - Ω (lower bound)
  - ϴ upper and lower bounds are the same

n	$f(n) = n^3$	$f(n) = n^3 + n^2$	$f(n) = n^3 - 8n^2 + 20n$
1	1	2	13
10	1,000	1,100	400
1,000	1,000,000,000	1,001,000,000	992,020,000
1,000,000	1.0×10^{18}	1.000001×10^{18}	9.99992×10^{17}

Computational Complexity

Computational Complexity

$O(1)$	constant time
$O(log\ n)$	logarithmic time
$O(n)$	linear time
$O(n\ log\ n)$	linearithmic time
$O(n^2)$	quadratic time
$O(n^c)$	polynomial time
$O(c^n)$	exponential time
$O(n!)$	factorial time
$O(\infty)$	infinite time

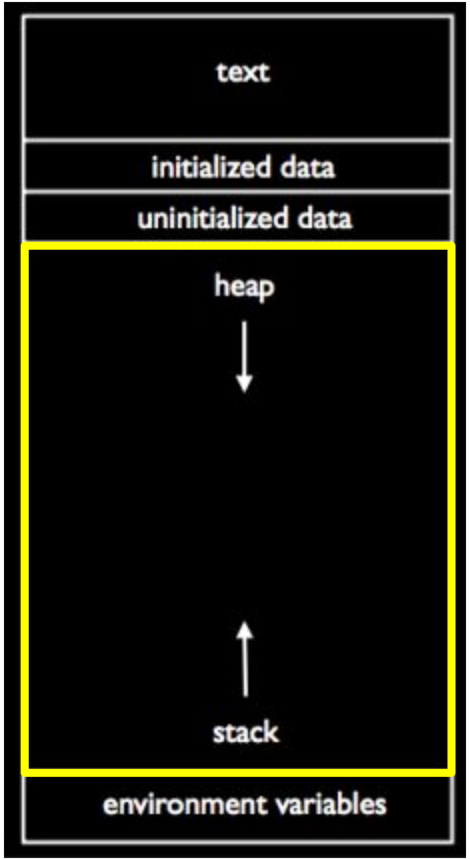
Week 4

Memory

Stack is a contiguous block of memory set aside when a program starts running:

- Metadata
- Any variables held in `read-only` memory
- All local variables - each function has its own stack frame and its variables are protected from other functions. The size of a function's stack frame is dependent largely on its local variables

Heap is essentially a region of unused memory that can be dynamically allocated



Stack frames

- When you call a function system sets aside space in memory for that function to do its job
- More than one frames can be open but only one can ever be active
- When a new function is called, a new frame is pushed onto the top of the stack and becomes active frame
- When a function finishes its work, its frame is popped off of the stack and the frame immediately below it becomes the new active function on the top of the stack.

Memory

- A huge array of 8-bit wide bytes
- Memory is limited!
- Each data type takes up a certain amount

Data Type	**Size (in bytes)**
int	4
char	1
float	4
double	8
long long	8
string	?

Pointers

- If we have some variable we know of, particularly one that lives on the stack and has a name, we can find its address by prepending a &. E.g., &num
- To access the data at an address, we need to dereference it, using the * operator.

Pointers: Let’s Look at an Example

Example 3

	c		date					sound			
	A		15					B	O	O	/0
0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB

point_to_date

Int date = 15;
<type> *<variable>
Int *point_to_date;
Point_to_date = &date

*point_to_date = 17;

Pointers

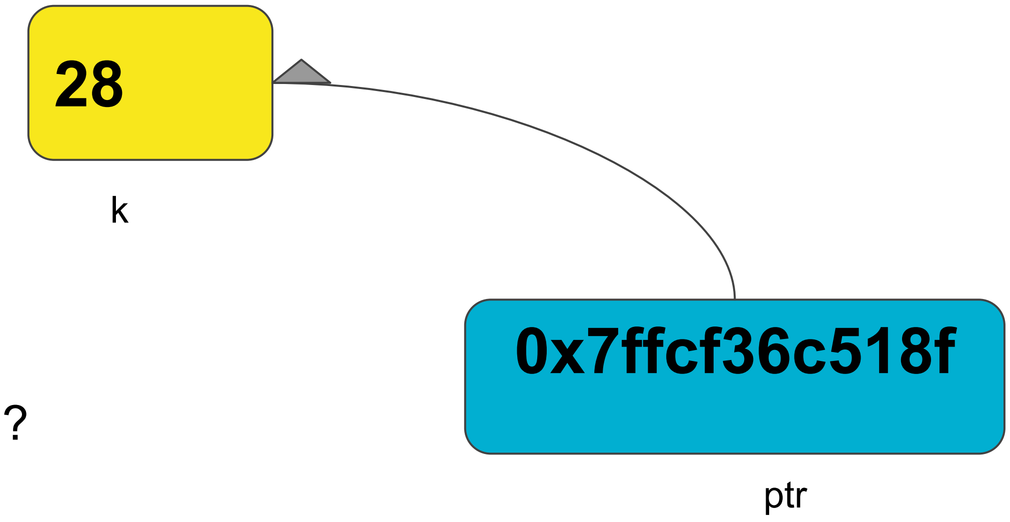
int k;

k = 25;

int *ptr;

ptr = &k;

*ptr = 28; what would happen?



Pointers

Gives you another way of passing data between functions.

Allows us to modify or inspect the location to which it points (* dereferencing the pointer)

Pointers

- The simplest pointer available to us in C is the NULL pointer.
  - What would this point to? **NOTHING!**
- When you create a pointer and you don’t set its value immediately, you should **always** set the value of the pointer to NULL.
- You can check whether a pointer is NULL using the equality operator (==).Ex) if (point_to_file == NULL)
  - What if you try to dereference NULL pointer? Sementation fault

Pointers

& is the reference, or address-of, operator. It returns the [redacted].

* is the dereference operator. A pointer's value is a memory address. When the dereference operator is applied to a pointer, it returns [redacted].

Dynamic Memory Allocation

- We get this dynamically-allocated memory via a call to the function malloc(), passing as its parameter the number of *bytes* we want. malloc() will return to you a **pointer** to that newly-allocated memory.
- If malloc() can't give you memory (because, say, the system ran out), you get a NULL pointer. ALWAYS CHECK FOR NULL!

```
// Statically obtain an integer
int x;

// Dynamically obtain an integer
int *px = [redacted]
```

Dynamic Memory Allocation

```
// Get an integer from the user
int x = get_int();

// Array of floats on the stack
float stack_array[x];

// Array of floats on the heap
float *heap_array = malloc(x * sizeof(float));
```

Dynamic Memory Allocation

- Dynamically allocated memory is not automatically returned to the system for later use when no longer needed.
- Failing to return memory back to the system when you no longer need it results in a **memory leak**, which compromises your system's performance.
- All memory that is dynamically allocated must be released back by free()-ing its pointer.

Example

---

ex2_substring.c

Week 5

Linked List

- Linked lists are a collection of nodes (themselves just a special struct), where each node contains data and a pointer to another node, which creates a chained (linked) collection of data.
- A linked list **node** is a special type of struct with two fields:
  - Data of some type
  - A pointer to another linked list node.
- How is it different from array?
  - Can grow or shrink as you wish
  - Traverse the pointers to access each element (no more random access!)

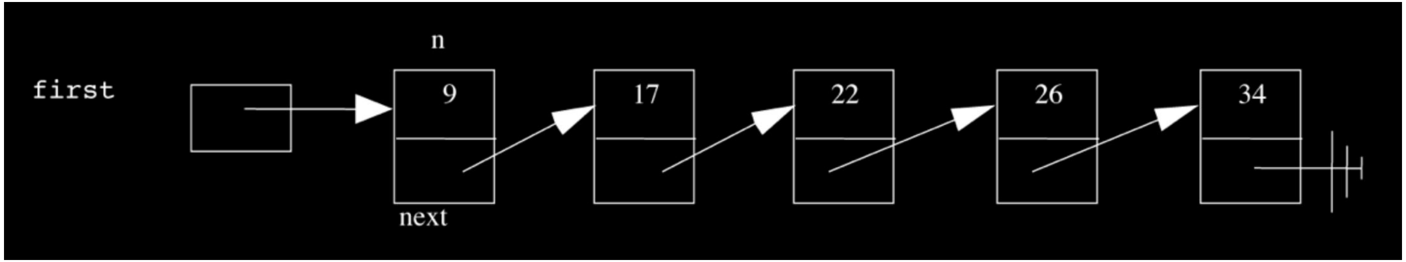
```c
typedef struct node
{
    int n;
    struct node *next;
}
node;
```

Linked list

Tasks

- Create a linked list
- Search an element
- Insert a node
- Delete the entire list
- Delete a single element

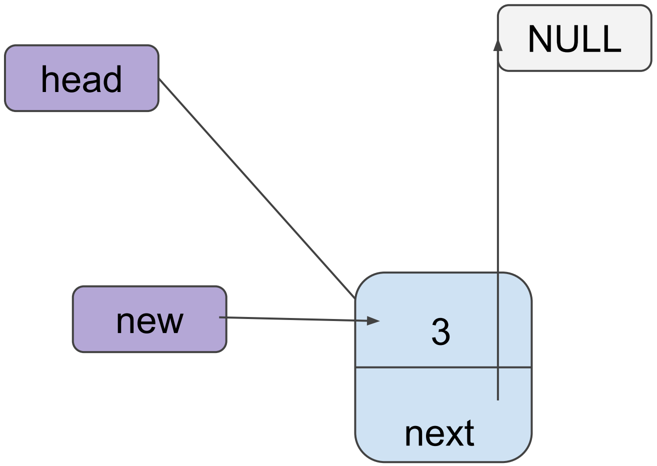
```c
// each element in the linked list
typedef struct node
{
        // data we want to store
        int n;

        //pointer to the next element in the list
        struct node *next;
}
node;
```



Linked List: Starts with a Node

1. Dynamically allocate space for a new node.
2. Check to make sure you didn’t run out of memory.
3. Initialize the value field.
4. Initialize the next field (specifically, to NULL).
5. Return a pointer to your newly created node.

Linked List: Starts with a Node




```
// each element in the linked list
typedef struct
{
        // data we want to store
        int n;

        //pointer to the next element in the list
        struct node *next;
}
node;

node *head = NULL;

node *new = malloc(sizeof(node));
if (new == NULL)
        return 1;

new -> n = word;
new -> next = head;
head = new;
```

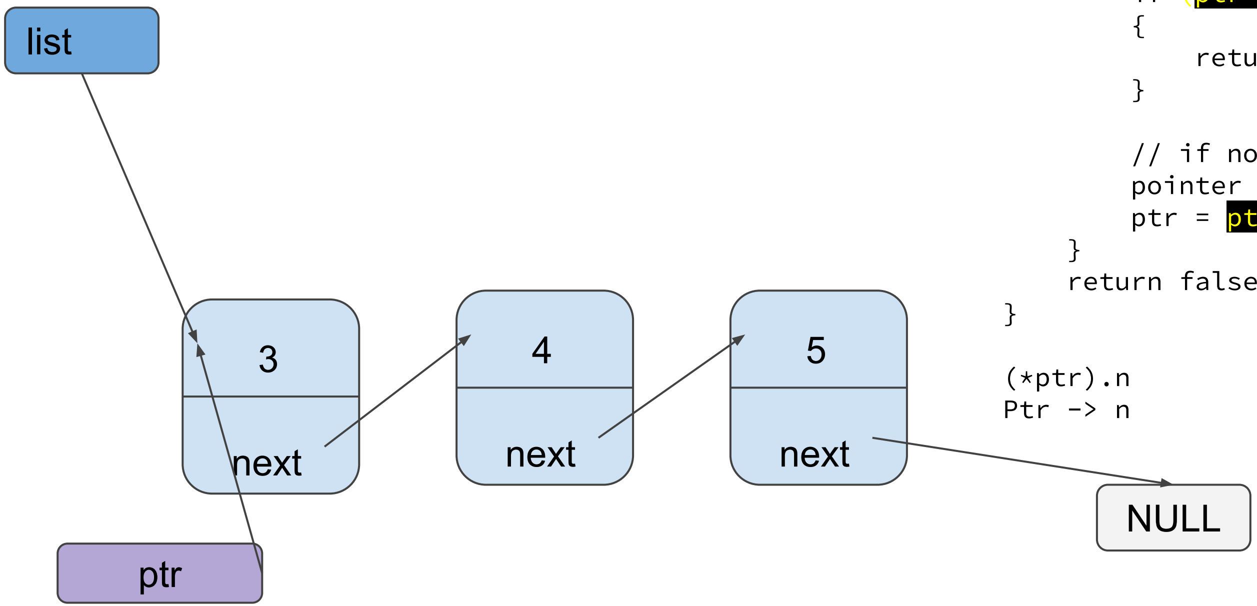
Let’s Look at an Example

`ex2_linked.c`

Linked List: Search an Element

1. Create a traversal pointer pointing to the first element
2. If the current node’s value field is what we’re looking for, return true.
3. If not, set the traversal pointer to the next pointer in the list and go back to the previous step.
4. If you’ve reached the last element of the list, return false.

Linked List: Search an Element




```
bool search(int n, node *list)
{
    // create a traversal pointer
    node *ptr = list;
    the list
    while (ptr != NULL)
    // traverse until the end of )
    {
        // if current n field is what we
        are looking for, return true
        if (ptr->n == n)
        {
            return true;
        }

        // if not, set ptr to the next
        pointer in the list
        ptr = ptr->next;
    }
    return false;
}

(*ptr).n
Ptr -> n
```

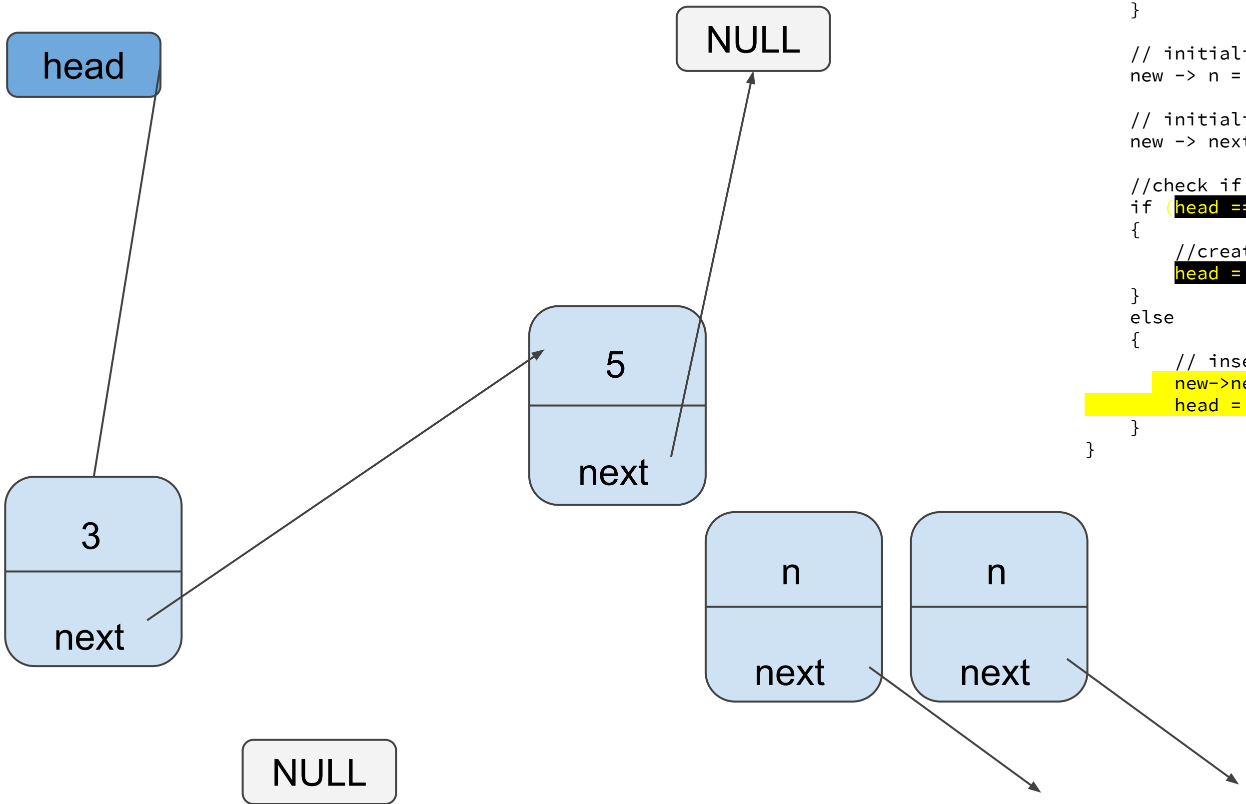
Let’s Look at an Example

`ex3_search_linked.c`

Linked List: Insert a Node

1. Dynamically allocate space for a new linked list node.
2. Check to make sure we didn’t run out of memory.
3. Populate and insert the node at the beginning of the linked list.
    a. So which pointer do we move first? The pointer in the newly created node, or the pointer pointing to the original head of the linked list?
    b. This choice matters!
4. Return a pointer to the new head of the linked list.

Linked List: Insert a Node

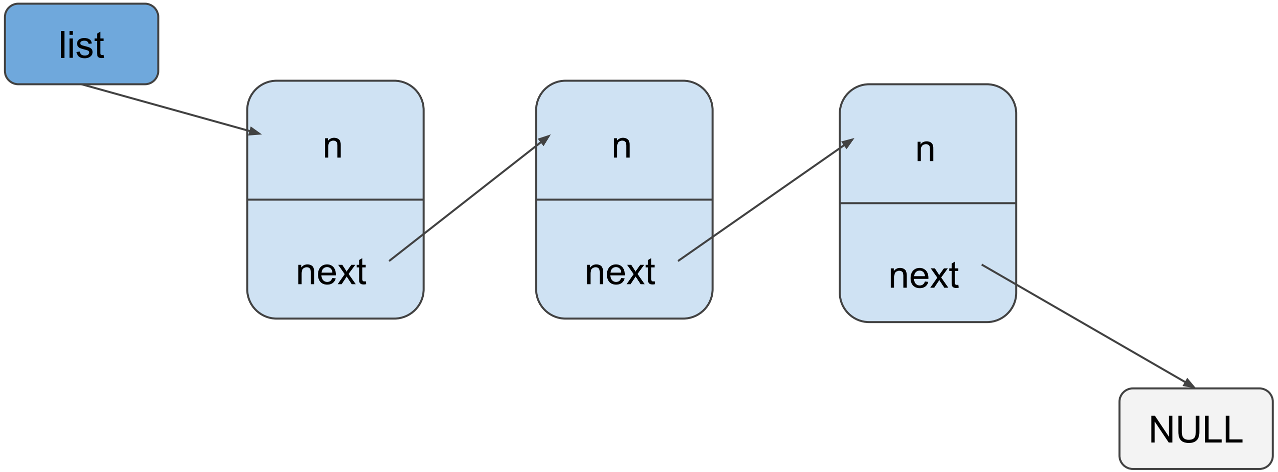



```
void insert(int n)
{
    //dynamically allocate space for a new node
    node *new = malloc(sizeof(node));

    //check to make sure we didn't run out of memory
    if (new == NULL)
    {
        exit(1);
    }

    // initialize the n field
    new -> n = n;

    // initialize the next field
    new -> next = NULL;

    //check if the list exists
    if (head == NULL)
    {
        //create a pointer to the new list
        head = new;
    }
    else
    {
        // insert new node at head
        new->next = head;
        head = new;
    }
}
```

Let’s Look at an Example

```
ex3_insert_linked.c
```

Linked List: Empty List



Linked List: Empty List

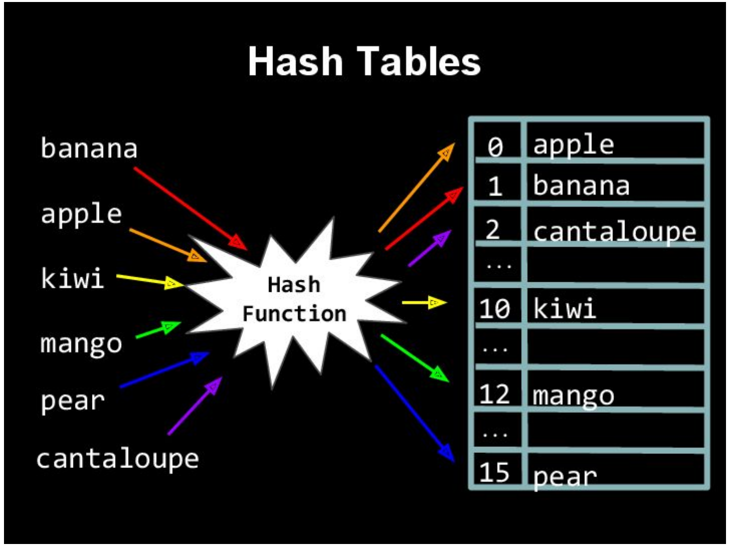
```
void empty_list(node *list)
{
    // if the list is empty, nothing else to do
    if (list == NULL)
    {
        return;
    }

    // empty the rest of the list
    empty_list(list->next);

    //free the current node
    free(list);
}
```

Linked List: Pros & Cons

- Pro: ability to grow and shrink
- Con: slower searches, insertions, deletions

Hash Table

- An "associative array" where the position of each element is decided by (associated with) the result of passing data into a *hash function*.
- Combines the random access of an array with the dynamism of a linked list.
- This means insertion deletion, and lookup can all tend toward $\Theta(1)$! We're gaining the advantages of both, and mitigating the disadvantages.
- Cons? Not good at ordering or sorting data



Hash Table: Hash Function

A hash function describes where to insert a word and, when necessary, where to look up a word.

A good hash function should:

- Use only the data being hashed
- Use all of the data being hashed
- Be deterministic (return same value for same data)
- Uniformly distribute data
- Generate very different hash codes for very similar data

```
#define HASH_MAX 10
string hashtable [10];

unsigned int hash(char *str)
{
      Int sum = 0;
      for(int j = 0; str [j] != '\0'; j++)
      {
            sum += str [j];
      }

      return sum % HASH_MAX;
}
```

Problem?! What happens if there are more than one word resulting in the same hash code? [redacted]

Hash Table

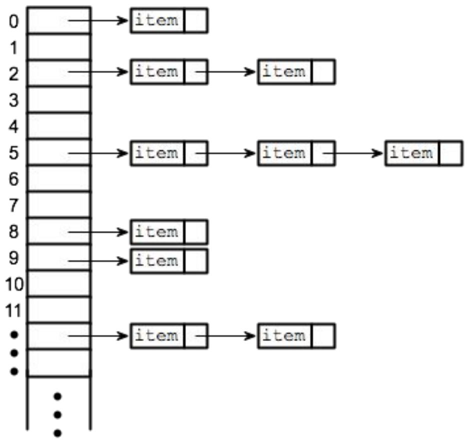
A solution to a collision?

- Linear probing: if a key hashes to the same index as a previously stored key, it is assigned the next available slot in the table
- Upperbound insertion, deletion, and lookup times have devolved to O(n), where n is the size of the table
- Linear probing may lead to clustering

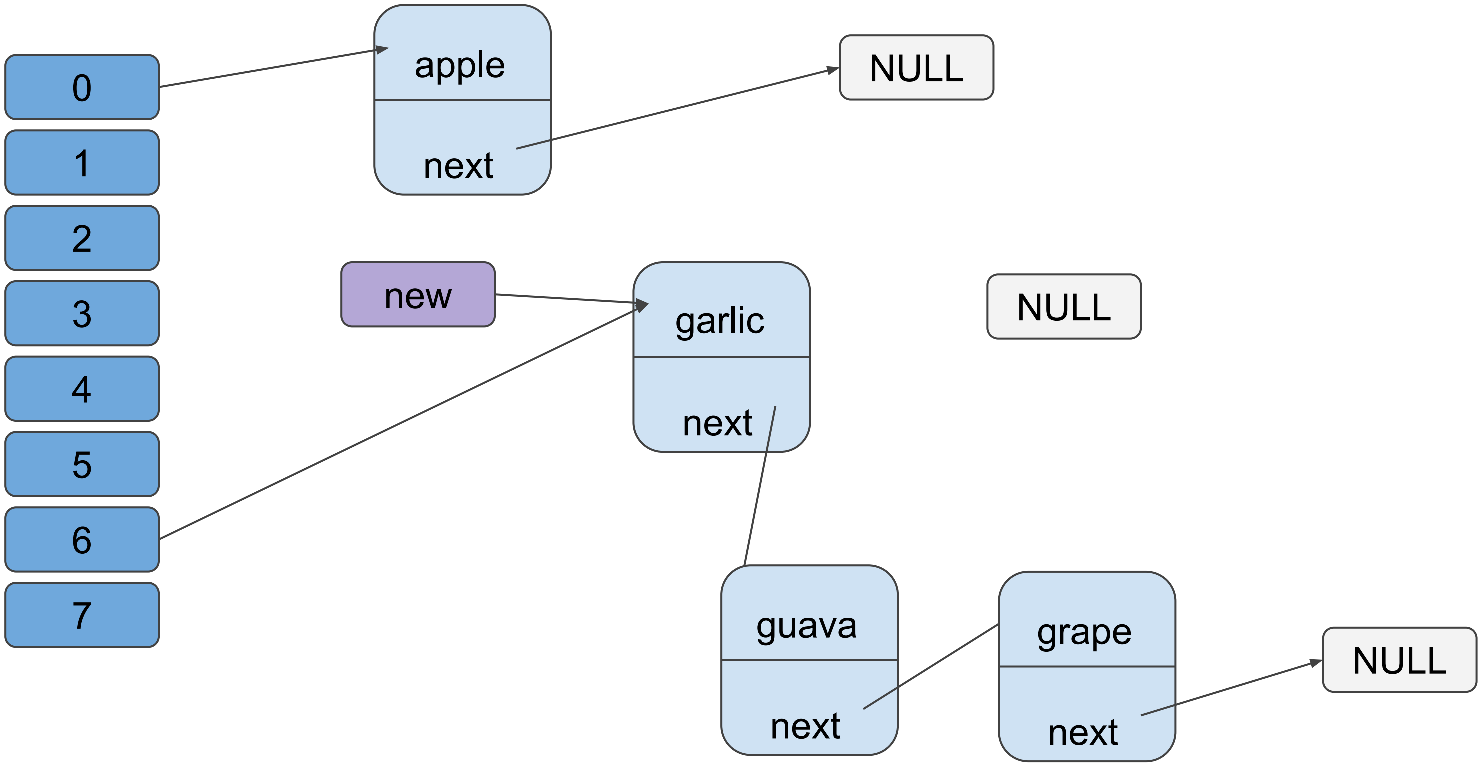
Another solution?

Hash Table: Chaining

- Chaining: each element of the array is a pointer to the head of a linked list, so multiple pieces of data can yield the same hash code and we’ll be able to store it all!



Hash Table: Insert a Node



Examples

`ex3_insert_linked.c`

`ex4_hashtable.c`

`ex5_enqueue.c`

`ex6_dequeue.c`

GOOD LUCK!!!