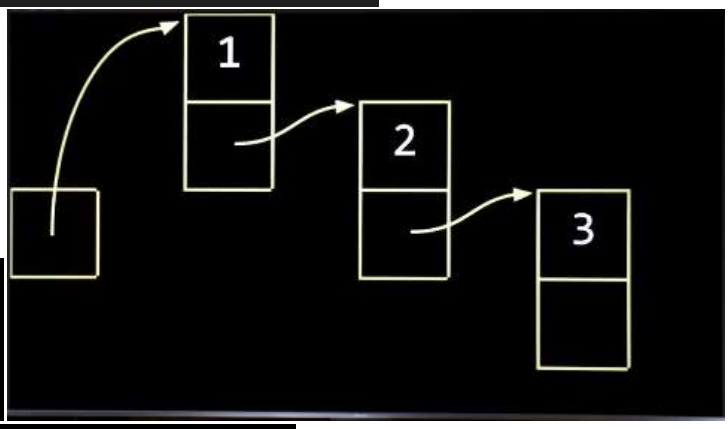
CS50: Week 5 - Data Structures

Data Structures

```
const int CAPACITY = 50;

typedef struct
{
    person people[CAPACITY];
    int size;
} queue;
```

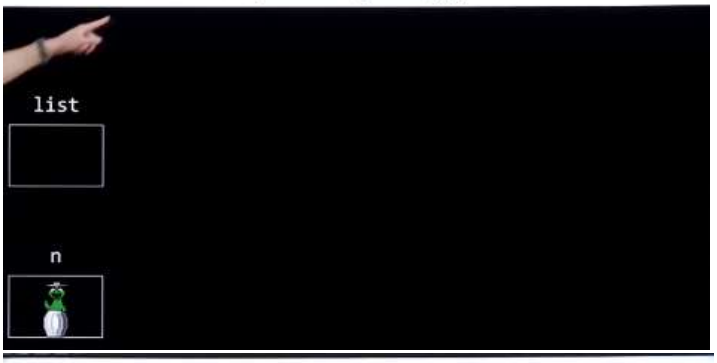

```
#include <cs50.h>
#include <stdio.h>

int main(void)

const int CAPACITY = 50;

typedef struct
{
    person people[CAPACITY];
    int size;
} queue;
```



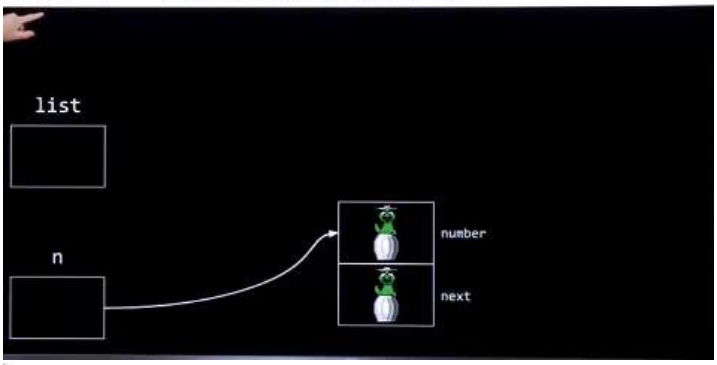

```
struct

->
```

```
typedef struct node
{
    int number;
    struct node *next;
} node;
```

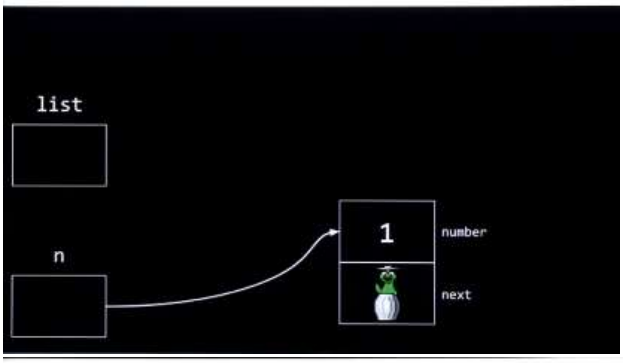
```
node *n = malloc(sizeof(node));
```



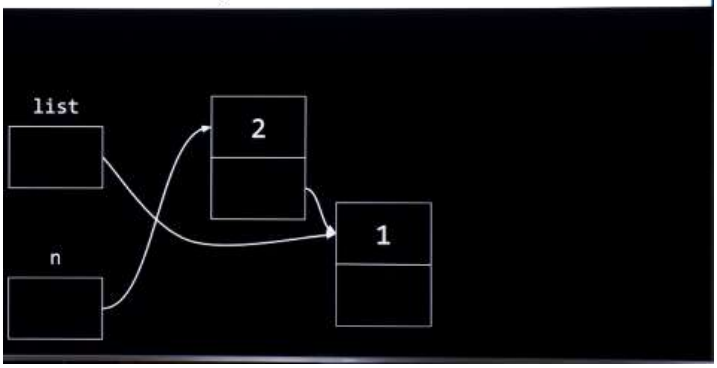

```
node *n = malloc(sizeof(node));
```

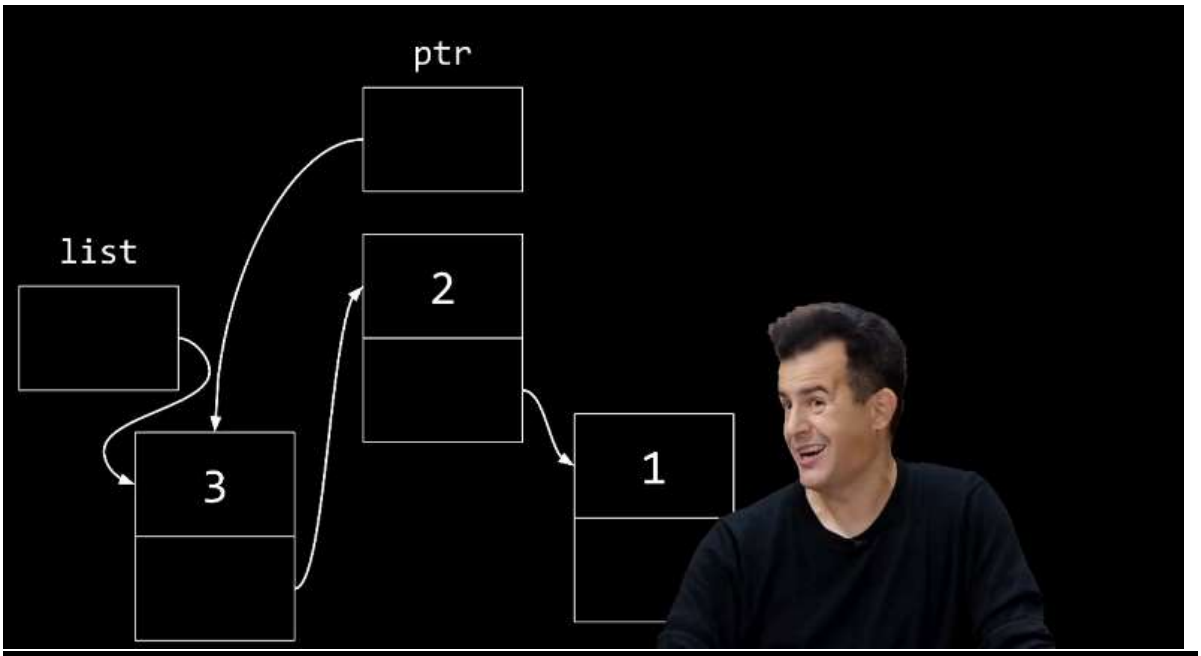


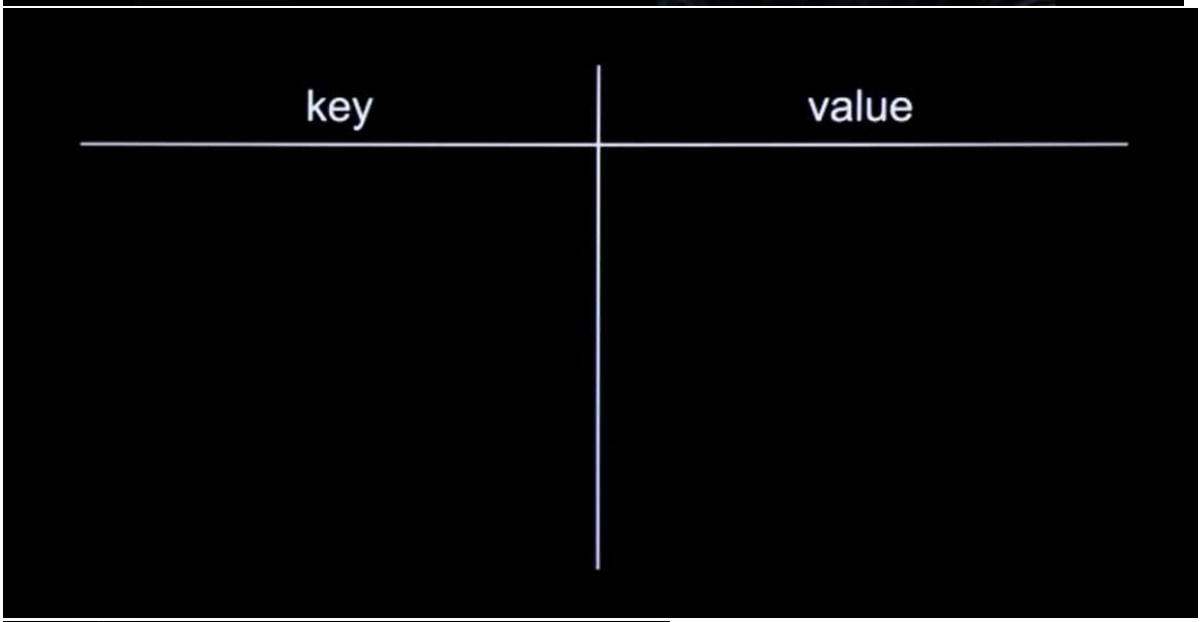

```
(*n).number = 1;
```

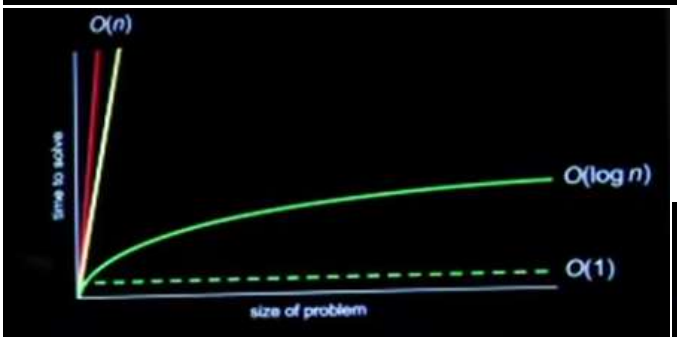



```
n->next = list;
```

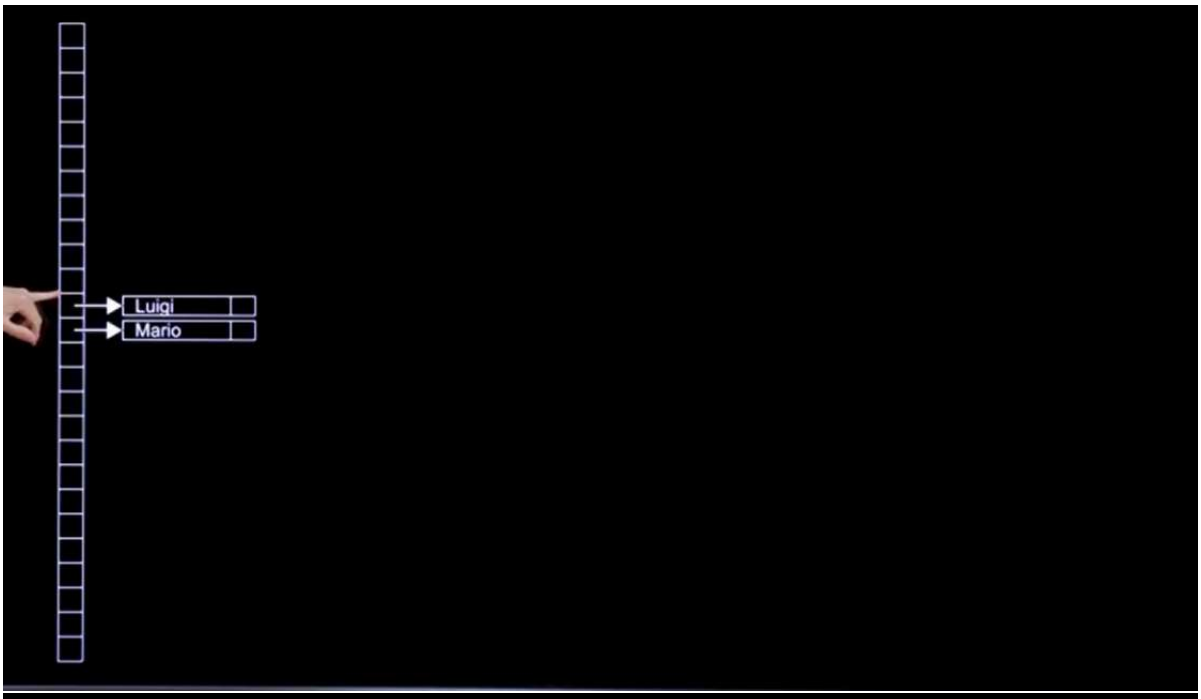


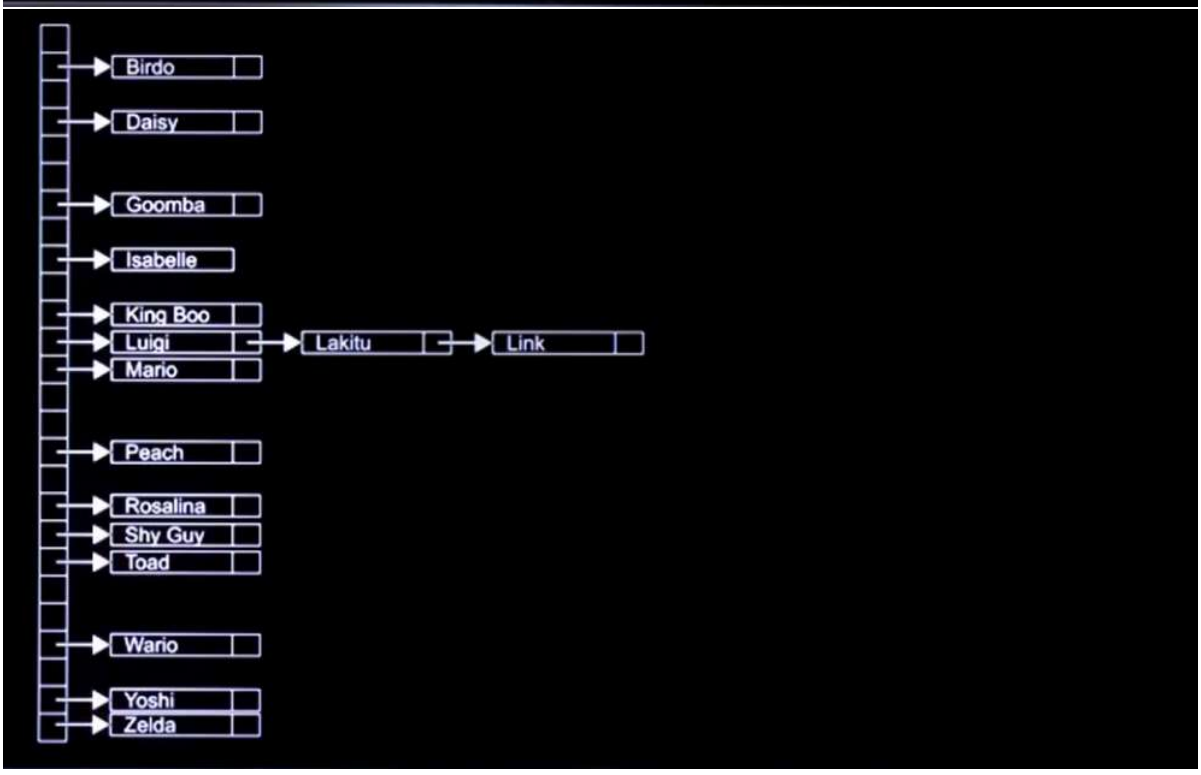


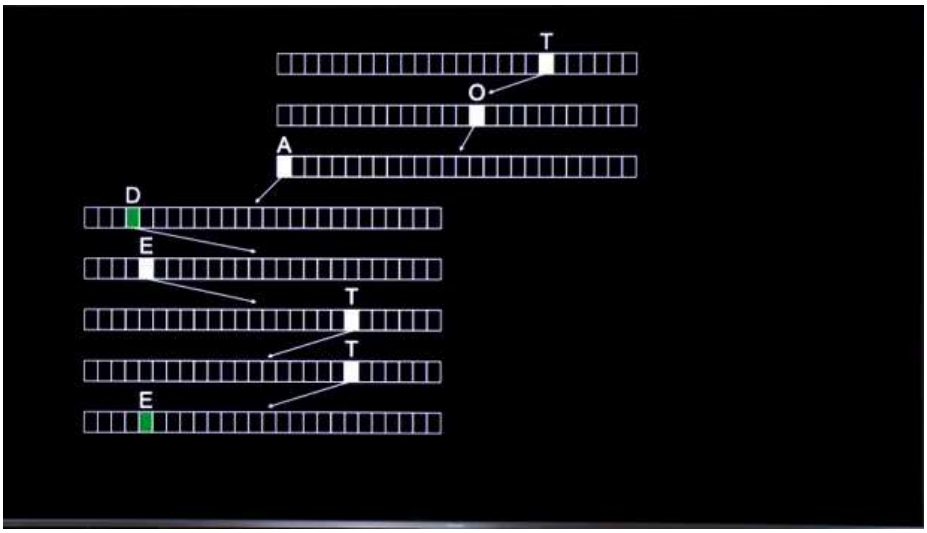




hash function


Luigi
Mario

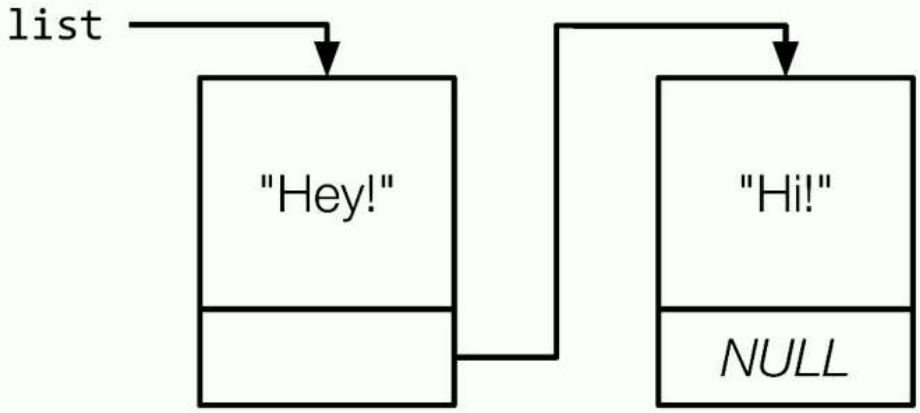

Birdo
Daisy
Goomba
Isabelle
King Boo
Luigi
Lakitu
Link
Mario
Peach
Rosalina
Shy Guy
Toad
Wario
Yoshi
Zelda

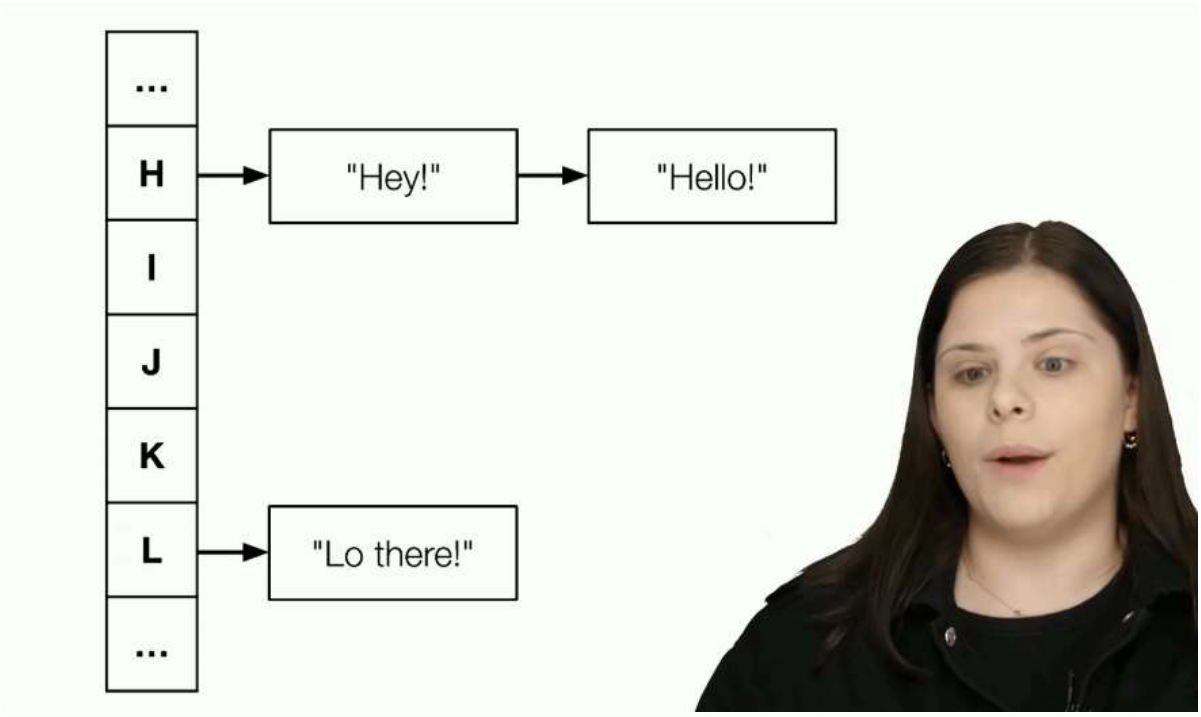


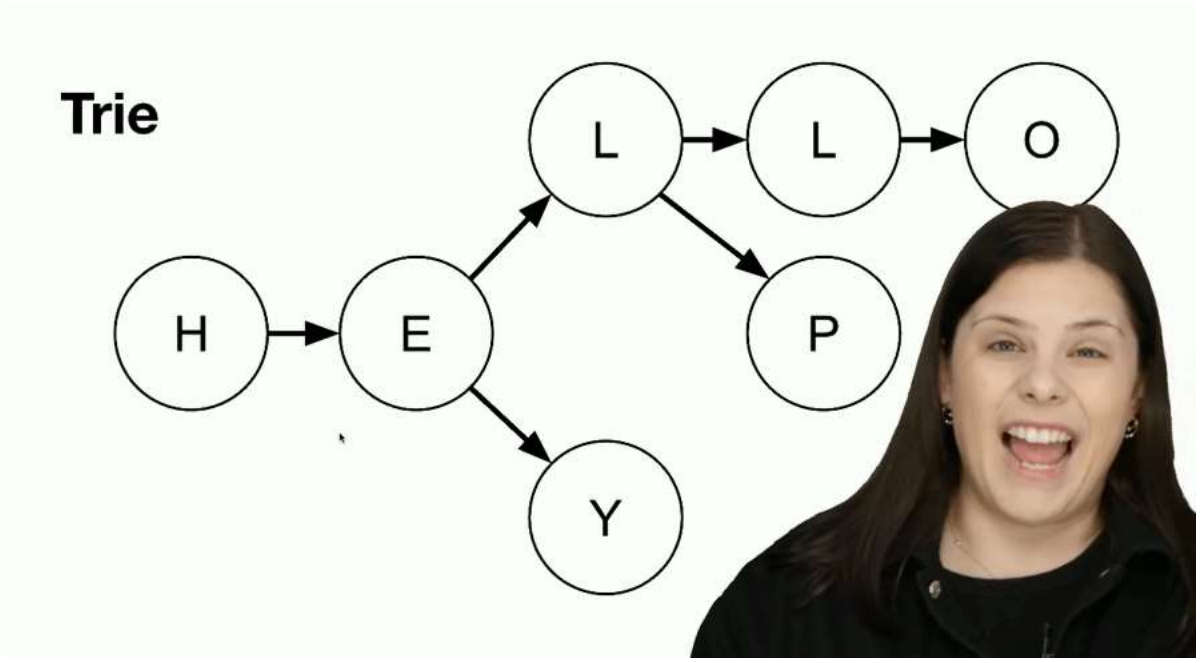
Section-5: Data Structures by Yuleia

1. Search
2. Insertion
3. Deletion

Linked List

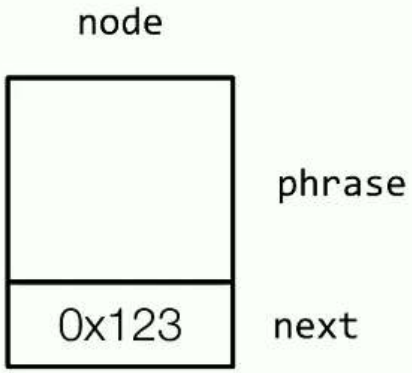






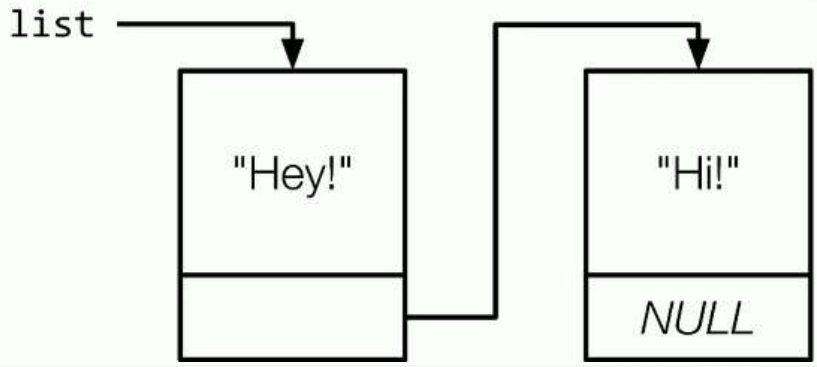
🔗 Trie Code Block

```
typedef struct node
{
    string phrase;
    struct node *next;
}
node;
```

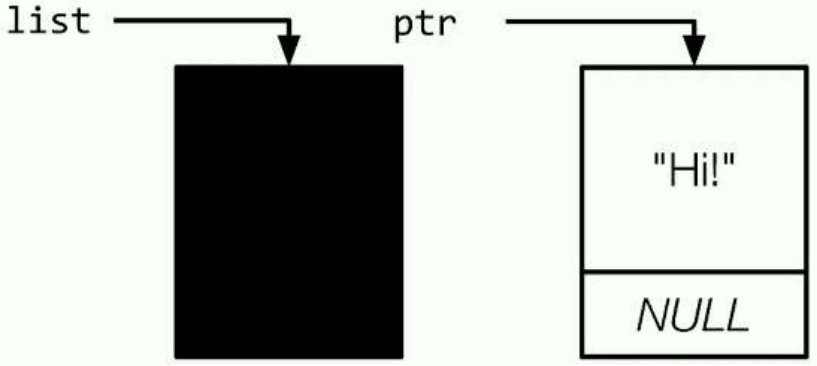


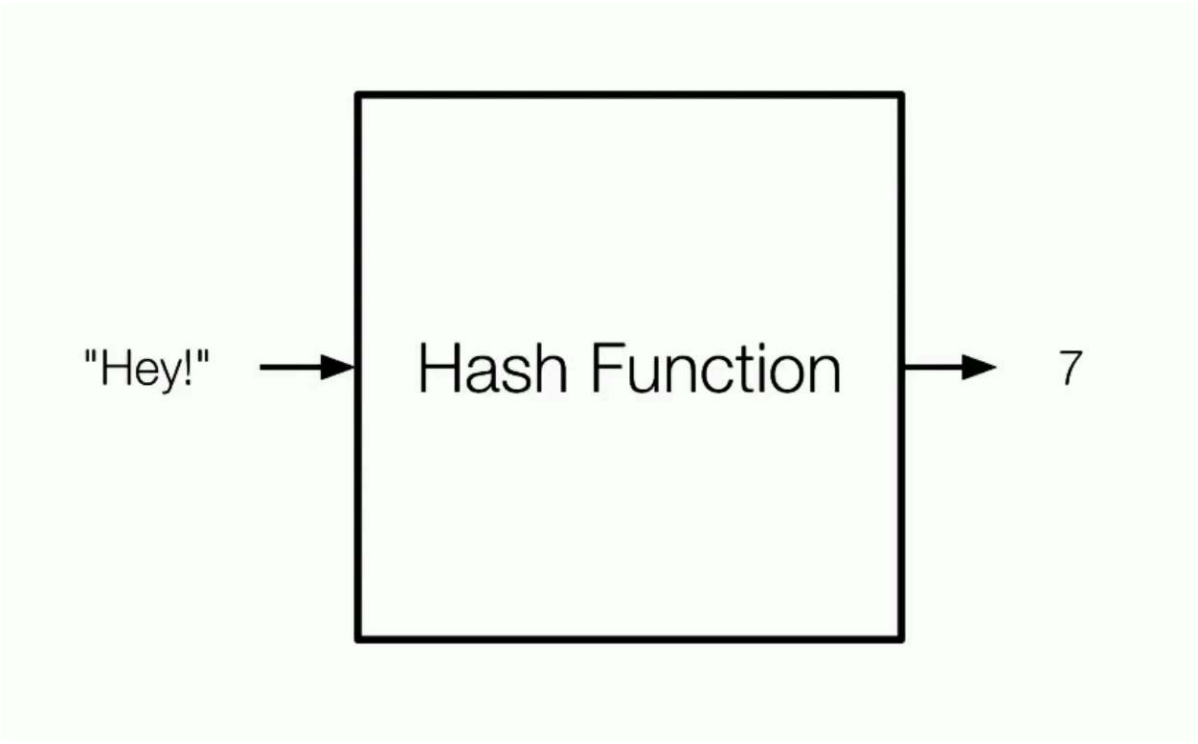

```
node *n = malloc(sizeof(node));
```

```
list = n;
```




```
list = ptr;
```



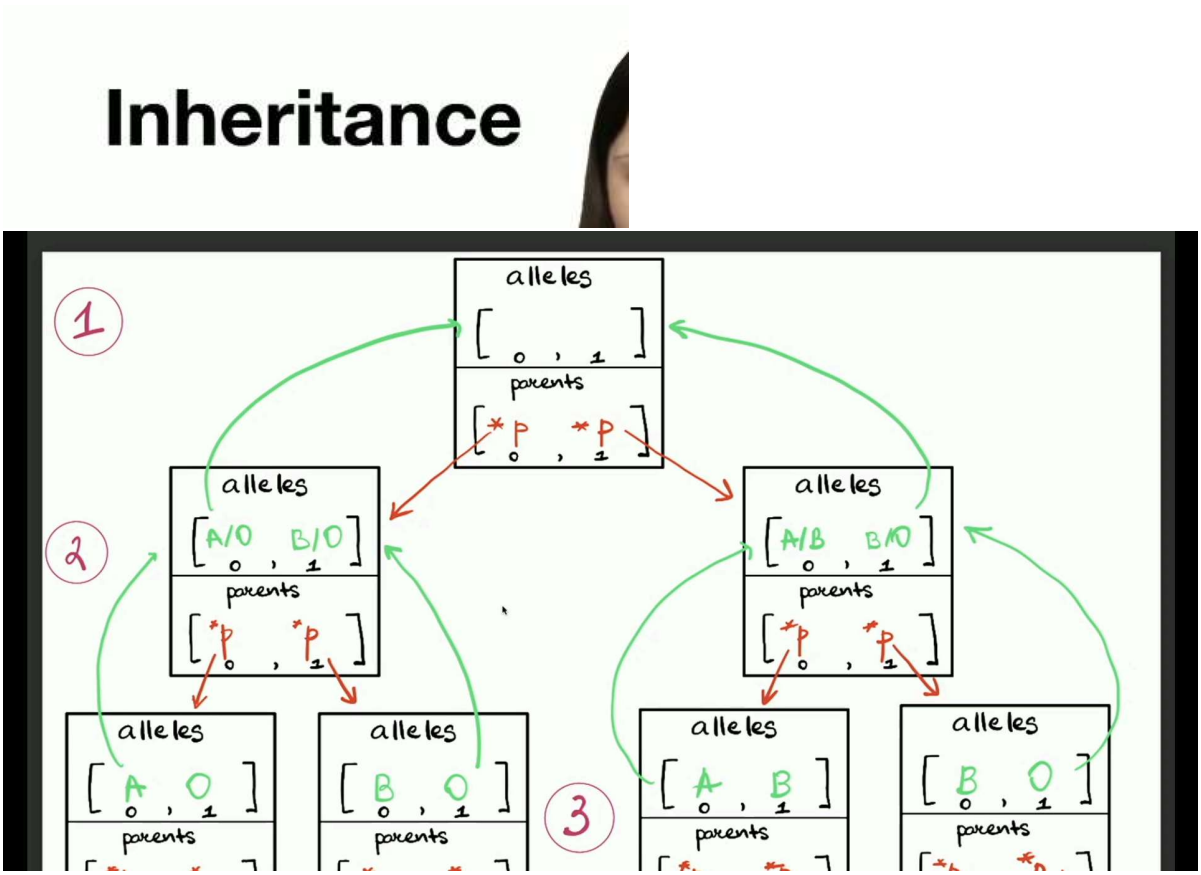


A good hash function…

Always gives you the same value for the same input

Produces an even distribution across buckets

Uses all buckets



Shorts: 05 - Data Structures

Structures

Structures

```
struct car
{
    int year;
    char model[10];
    char plate[7];
    int odometer;
    double engine_size;
};
```

Structures

- Once we have defined a structure, which we typically do in separate .h files or atop our programs outside of any functions, we have effectively created a new type.
- That means we can create variables of that type using the familiar syntax.
- We can also access the various **fields** (also known as **members**) of the structure using the dot operator (.)

Structures

```
// variable declaration
struct car mycar;

// field accessing
mycar.year = 2011;
strcpy(mycar.plate, "CS50");
mycar.odometer = 50505;
```

Structures

```
// variable declaration
struct car mycar;

// field accessing
mycar.year = 2011;
strcpy(mycar.plate, “CS50”);
mycar.odometer = 50505;
```

Structures

```
// variable declaration
struct car mycar;

// field accessing
mycar.year = 2011;
strcpy(mycar.plate, “CS50”);
mycar.odometer = 50505;
```

Structures

- Structures, like variables of all other data types, do not need to be created on the stack. We can dynamically allocate structures at run time if our program requires it.
- In order to access the fields of our structures in that situation, we first need to dereference the pointer to the structure, and then we can access its fields.

Structures

```
// variable declaration
struct car *mycar = malloc(sizeof(struct car));
```

Dynamic Allocation of the Memory

Structures

- This is a little annoying. And so as you might expect, there's a shorter way!

- The arrow operator (`->`) makes this process easier. It's an operator that does two things back-to-back:
  - First, it **dereferences** the pointer on the left side of the operator.
  - Second, it **accesses** the field on the right side of the operator.

Structures

```
// variable declaration
struct car *mycar = malloc(sizeof(struct car));

// field accessing
mycar->year = 2011;
strcpy(mycar->plate, "CS50");
mycar->odometer = 50505;
```

Singly-Linked Lists

Singly-Linked Lists

- Arrays also suffer from a great inflexibility – what happens if we need a larger array than we thought?

- Through clever use of pointers, dynamic memory allocation, and `structs`, we can put the pieces together to develop a new kind of data structure that gives us the ability to grow and shrink a collection of like values to fit our needs.

Singly-Linked Lists

- We call this combination of elements, when used in this way, a **linked list**.

- A linked list **node** is a special kind of struct with two members:
  - Data of some data type (`int`, `char`, `float`...)
  - A pointer to another node of the same type

- In this way, a set of nodes together can be thought of as forming a chain of elements that we can follow from beginning to end.

Singly-Linked Lists

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

Singly-Linked Lists

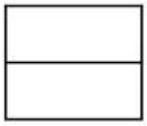
- In order to work with linked lists effectively, there are a number of operations that we need to understand:

1. Create a linked list when it doesn't already exist.
2. Search through a linked list to find an element.
3. Insert a new node into the linked list.
4. Delete a single element from a linked list.
5. Delete an entire linked list.

Singly-Linked Lists

```
sllnode* new = create(6);
```

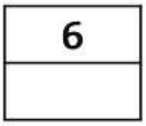
a. Dynamically allocate space for a new `sllnode`.
b. Check to make sure we didn't run out of memory.
c. Initialize the node's `val` field.
d. Initialize the node's `next` field.
e. Return a pointer to the newly created `sllnode`.



Singly-Linked Lists

```
sllnode* new = create(6);
```

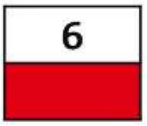
a. Dynamically allocate space for a new `sllnode`.
b. Check to make sure we didn't run out of memory.
c. Initialize the node's `val` field.
d. Initialize the node's `next` field.
e. Return a pointer to the newly created `sllnode`.



Singly-Linked Lists

```
sllnode* new = create(6);
```

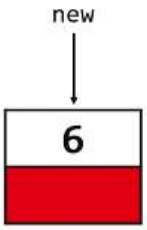
a. Dynamically allocate space for a new `sllnode`.
b. Check to make sure we didn't run out of memory.
c. Initialize the node's `val` field.
d. Initialize the node's `next` field.
e. Return a pointer to the newly created `sllnode`.



Singly-Linked Lists

```
sllnode* new = create(6);
```

a. Dynamically allocate space for a new `sllnode`.
b. Check to make sure we didn't run out of memory.
c. Initialize the node's `val` field.
d. Initialize the node's `next` field.
e. Return a pointer to the newly created `sllnode`.



Singly-Linked Lists

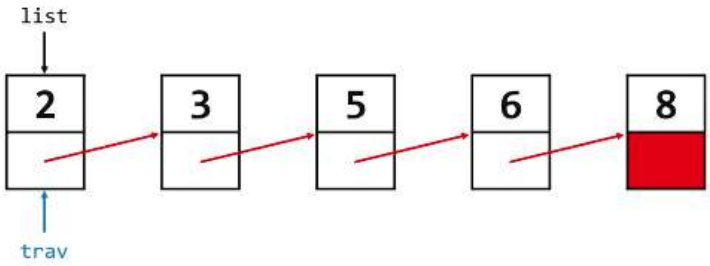
- Search through a linked list to find an element.

```
bool find(sllnode* head, VALUE val);
```

- Steps involved:
  a. Create a traversal pointer pointing to the list's head.
  b. If the current node's `val` field is what we're looking for, report success.
  c. If not, set the traversal pointer to the next pointer in the list and go back to step b.
  d. If you've reached the end of the list, report failure.

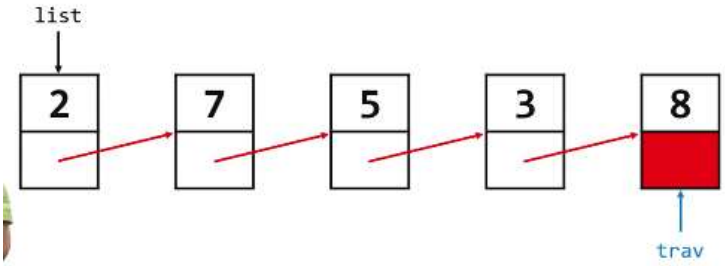
Singly-Linked Lists

```
bool exists = find(list, 6);
```



Singly-Linked Lists

```
bool exists = find(list, 6);
```



Singly-Linked Lists

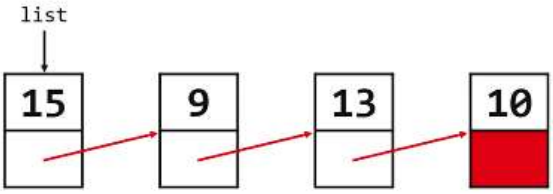
- Insert a new node into the linked list.

```
sllnode* insert(sllnode* head, VALUE val);
```

- Steps involved:
  a. Dynamically allocate space for a new `sllnode`.
  b. Check to make sure we didn't run out of memory.
  c. Populate and insert the node at the beginning of the linked list.
  d. Return a pointer to the new head of the linked list.

Singly-Linked Lists

```
list = insert(list, 12);
```



Singly-Linked Lists

- Decision time!
- Which pointer should we move first? Should the "12" node be the new head of the linked list, since it now exists, or should we connect it to the list first?
- This is one of the trickiest things with linked lists. Order matters!

Singly-Linked Lists

- Delete an entire linked list.

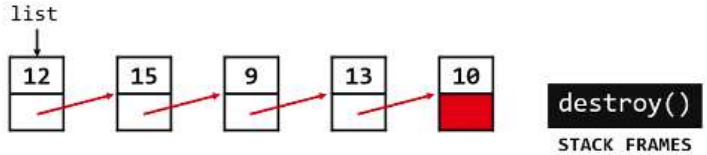
```
void destroy(sllnode* head);
```

- Steps involved:
  a. If you've reached a null pointer, stop.
  b. Delete the rest of the list.
  c. Free the current node.

Singly-Linked Lists

```
destroy(list);
```

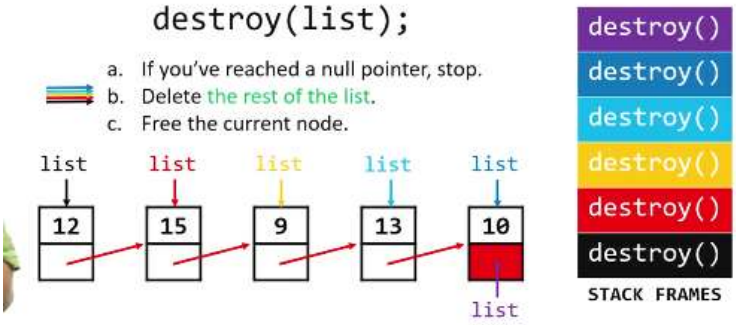
a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Singly-Linked Lists

```
destroy(list);
```

a. If you've reached a null pointer, stop.
b. Delete the rest of the list.
c. Free the current node.



Doubly-Linked Lists

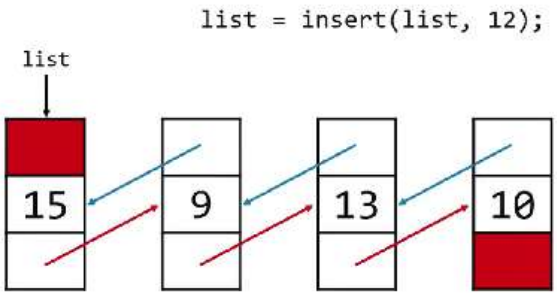
Doubly-Linked Lists

- Singly-linked lists really extend our ability to collect and organize data, but they suffer from a crucial limitation.
  - We can only ever move in one direction through the list.
- Consider the implication that would have for trying to delete a node.
- A doubly-linked list, by contrast, allows us to move forward and backward through the list, all by simply adding one extra pointer to our `struct` definition.

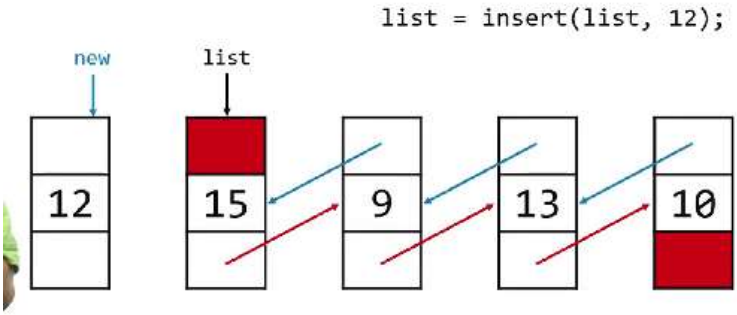
Doubly-Linked Lists

```
typedef struct dllist
{
    VALUE val;
    struct dllist* prev;
    struct dllist* next;
}
dllnode;
```

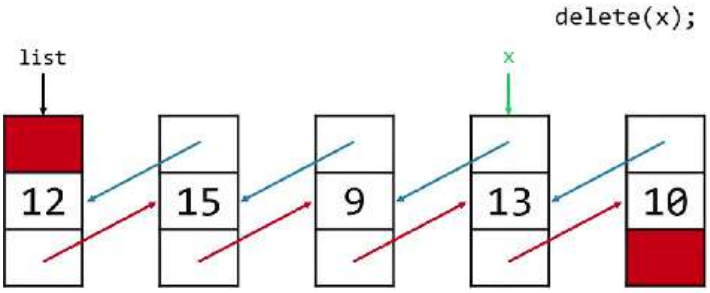
Doubly-Linked Lists



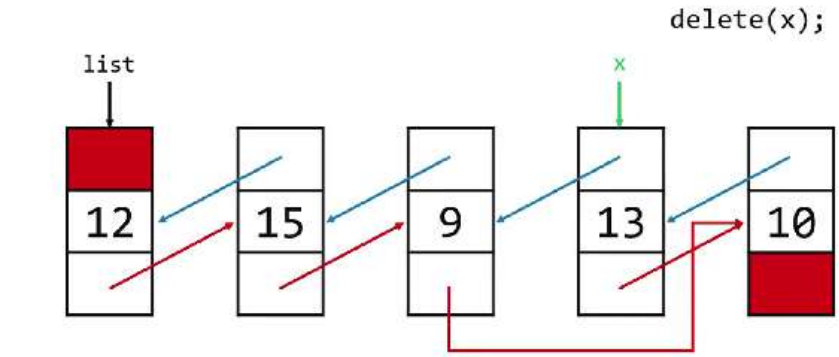
Doubly-Linked Lists



Doubly-Linked Lists



Doubly-Linked Lists



Stacks

Stacks (Data Structure)

- There are only two operations that may legally be performed on a stack.
  - ***Push***: Add a new element to the top of the stack.
  - ***Pop***: Remove the most recently-added element from the top of the stack.

Stacks (Data Structure)

- Array-based implementation

```
typedef struct _stack
{
    VALUE array[CAPACITY];
    int top;
}
stack;
```

Stacks (Data Structure)

- Array-based implementation

```
stack s;
```

Stacks (Data Structure)

- Array-based implementation

```
stack s;
```

s

Stacks (Data Structure)

- Array-based implementation
  - ***Push***: Add a new element to the top of the stack.

In the general case, `push()` needs to:

- Accept a pointer to the stack.
- Accept data of type VALUE to be added to the stack.
- Add that data to the stack at the top of the stack.
- Change the location of the top of the stack.

Stacks (Data Structure)

- Array-based implementation
  - ***Pop***: Remove the most recent element from the top of the stack.

In the general case, pop( ) needs to:

- Accept a pointer to the stack.
- Change the location of the top of the stack.
- Return the value that was removed from the stack.

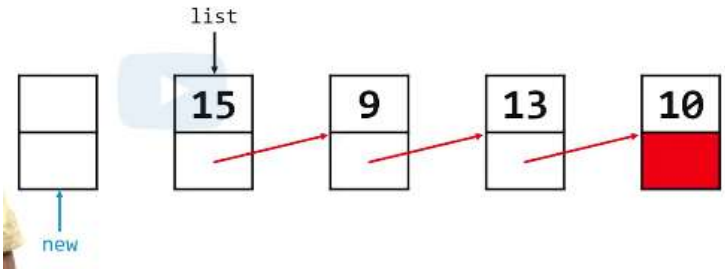
Stacks (Data Structure)

- Linked list-based implementation

```
typedef struct _stack
{
    VALUE val;
    struct _stack *next;
}
stack;
```

Stacks (Data Structure)

```
list = push(list, 12);
```



Queue
1
2
3

Stack
1
2
3

Queues

Queues

- A queue is a special type of structure that can be used to maintain data in an organized way.
- This data structure is commonly implemented in one of two ways: as an **array** or as a **linked list**.
- In either case, the important rule is that when data is added to the queue, it is tacked onto the end, and so if an element needs to be removed, the element at the front is the only element that can legally be removed.
  - *First in, first out (FIFO)*

Queues

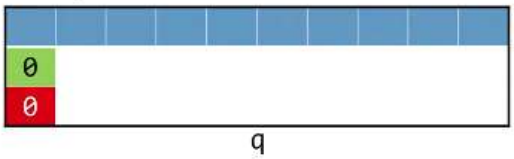
- Array-based implementation

```
queue q;
```

Queues

- Array-based implementation

```
queue q;
q.front = 0;
q.size = 0;
```



Queues

- Array-based implementation
  - ***Enqueue***: Add a new element to the end of the queue.

In the general case, `enqueue()` needs to:

- Accept a pointer to the queue.
- Accept data of type VALUE to be added to the queue.
- Add that data to the queue at the end of the queue.
- Change the size of the queue.

Queues

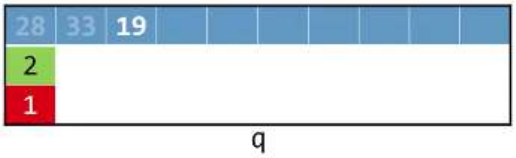
- Array-based implementation

```
VALUE dequeue(queue* q);
```

Queues

- Array-based implementation

```
int x = dequeue(&q);
```

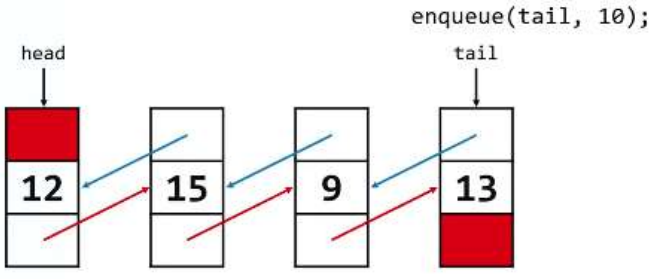


Queues

- Just make sure to always maintain pointers to the head ***and*** tail of the linked list! (probably global)

- To **enqueue**:
  - Dynamically allocate a new node;
  - Set its next pointer to NULL, set its prev pointer to the tail;
  - Set the tail's next pointer to the new node;
  - Move the tail pointer to the newly-created node.

Queues



Hash Tables

Hash Tables

- Hash tables combine the random access ability of an array with the dynamism of a linked list.

- This means (assuming we define our hash table well):
  - Insertion can start to tend toward $\theta(1)$
  - Deletion can start to tend toward $\theta(1)$
  - Lookup can start to tend toward $\theta(1)$

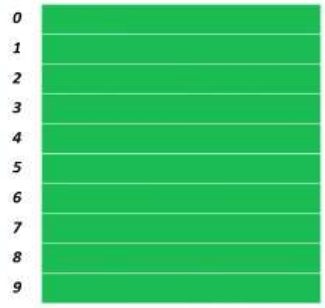
- We're gaining the advantages of both types of data structure, while mitigating the disadvantages.

Hash Tables

- A hash table amounts to a combination of two things with which we're quite familiar.
  - First, a **hash function**, which returns an nonnegative integer value called a *hash code*.
  - Second, an **array** capable of storing data of the type we wish to place into the data structure.

- The idea is that we run our data through the hash function, and then store the data in the element of the array represented by the returned hash code.

Hash Tables




```
int x = hash("John");
```

Hash Tables


```
int x = hash("John");

// x is now 4

hashtable[x] = "John";
```

Hash Tables

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

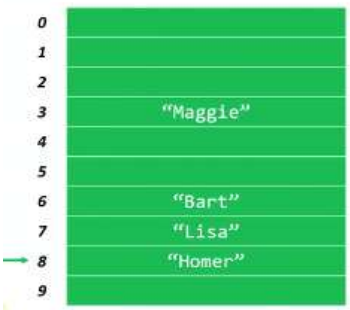
Hash Tables


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


Hash Tables

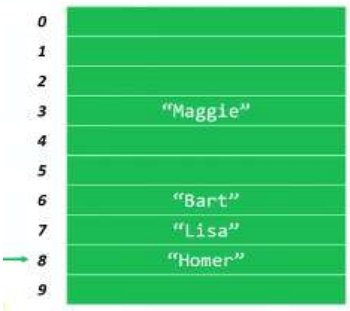
- Resolving collisions: *Linear probing*
- In this method, if we have a collision, we try to place the data in the next consecutive element in the array (wrapping around to the beginning if necessary) until we find a vacancy.
- That way, if we don't find what we're looking for in the first location, at least hopefully the element is somewhere nearby.

Hash Tables



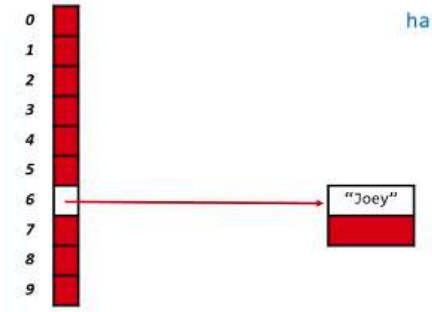
hash("Bart"); returns **6**
hash("Lisa"); returns **6**
hash("Homer"); returns **7**
hash("Maggie"); returns **3**
hash("Marge"); returns **6**

Hash Tables



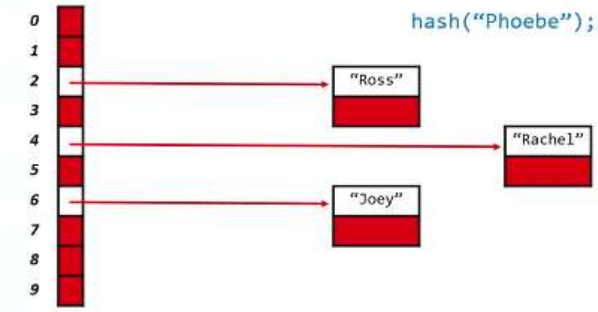
hash("Bart"); returns **6**
hash("Lisa"); returns **6**
hash("Homer"); returns **7**
hash("Maggie"); returns **3**
hash("Marge"); returns **6**

Hash Tables



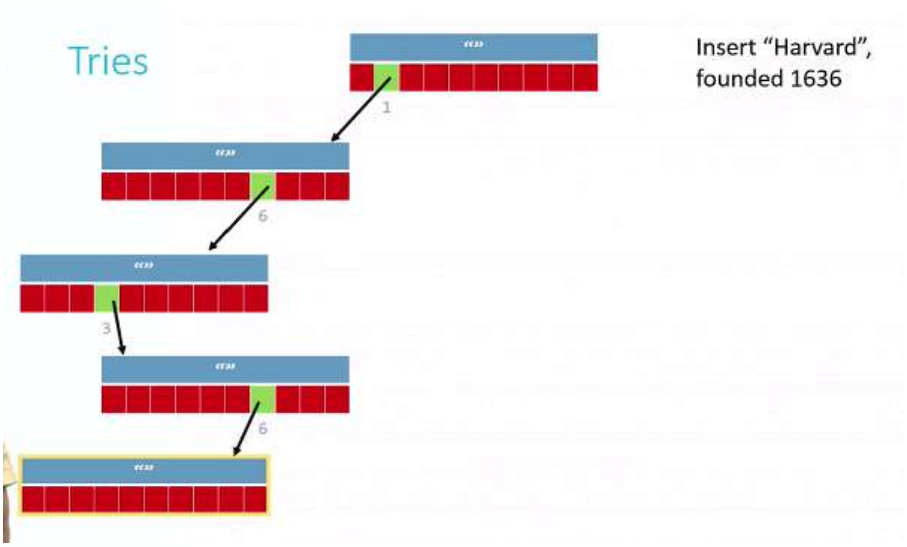
hash("Joey"); returns **6**

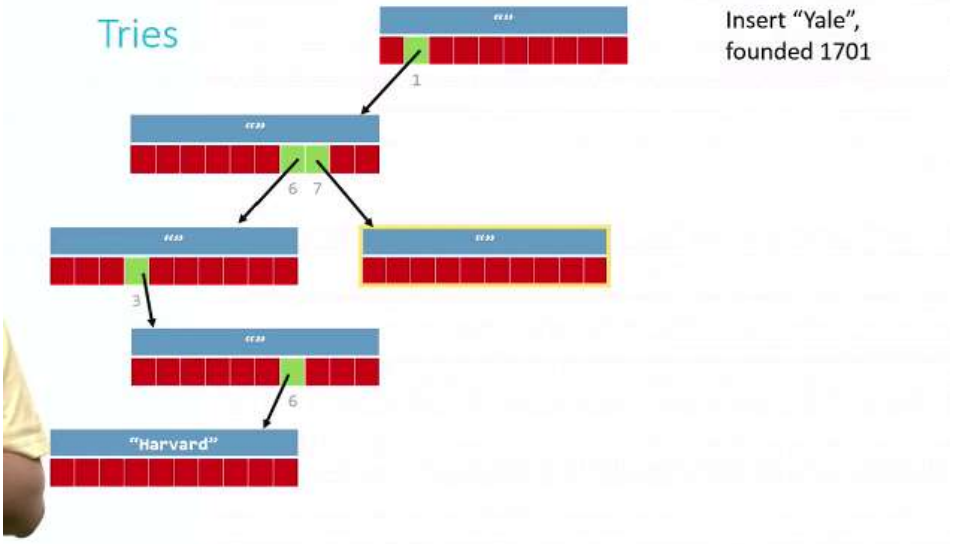
Hash Tables

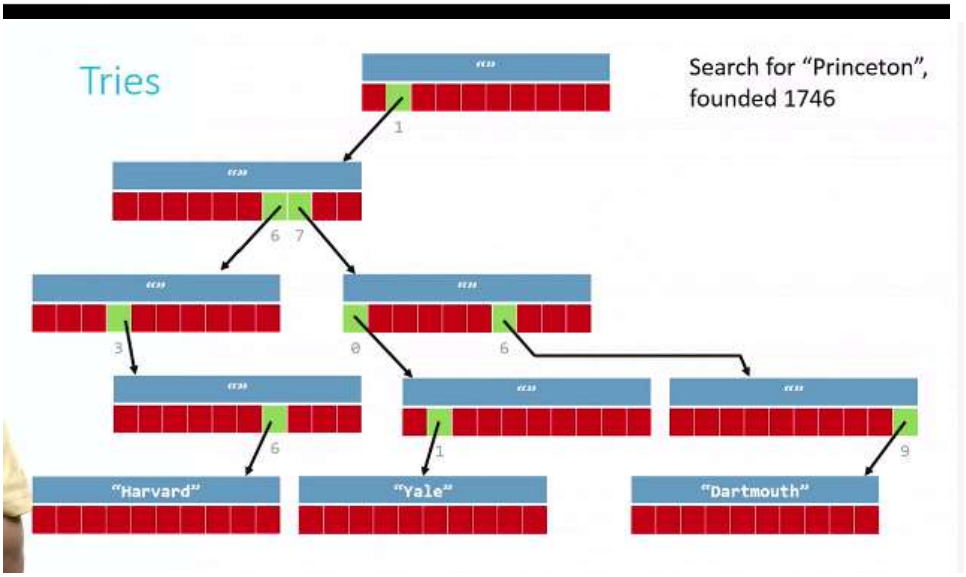


hash("Phoebe"); returns

Tries







Data Structures

Data Structures Summary

- Arrays
  - Insertion is bad – lots of shifting to fit an element in the middle
  - Deletion is bad – lots of shifting after removing an element
  - Lookup is great – random access, constant time
  - Relatively easy to sort
  - Relatively small size-wise
  - Stuck with a fixed size, no flexibility

Data Structures Summary

- Linked lists
  - Insertion is easy – just tack onto the front
  - Deletion is easy – once you find the element
  - Lookup is bad – have to rely on linear search
  - Relatively difficult to sort – unless you're willing to compromise on super-fast insertion and instead sort as you construct
  - Relatively small size-wise (not as small as arrays)

Data Structures Summary

- Hash tables
  - Insertion is a two-step process – hash, then add
  - Deletion is easy – once you find the element
  - Lookup is on average better than with linked lists because you have the benefit of a real-world constant factor
  - Not an ideal data structure if sorting is the goal – just use an array
  - Can run the gamut of size

Data Structures Summary

- Tries
  - Insertion is complex – a lot of dynamic memory allocation, but gets easier as you go
  - Deletion is easy – just free a node
  - Lookup is fast – not quite as fast as an array, but almost
  - Already sorted – sorts as you build in almost all situations
  - Rapidly becomes huge, even with very little data present, not great if space is at a premium