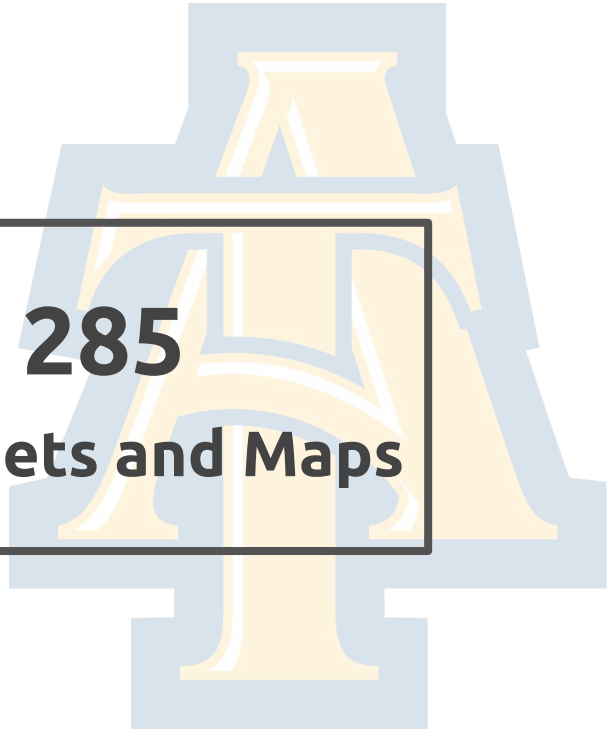
COMP - 285

Advanced Analysis of Algorithms

# Welcome to COMP 285

## Lecture 5: Stacks, Queues, Sets and Maps



**Chris Lucas (cflucas@ncat.edu)**

# HW1 was due!

**Today @ 1:59pm!**

# HW 2 released by EoD!

# Quiz!

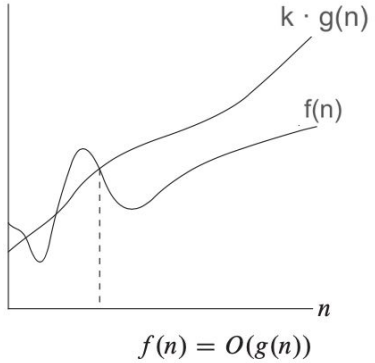
*But first ...*

# Recall where we ended last lecture...

# Recap

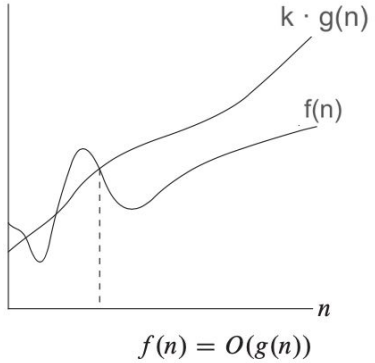
# Recap

**Big-Oh: Upper-bound | f = O(g) is similar to f ≤ g**
**“f grows no faster than g”**

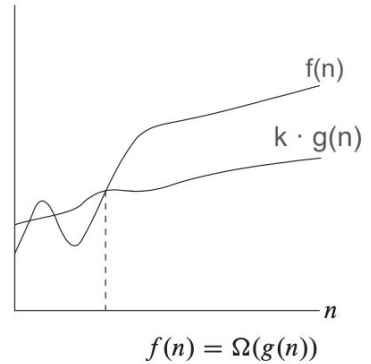


# Recap

**Big-Oh: Upper-bound | f = O(g) is similar to f ≤ g**
**“f grows no faster than g”**




**Big-Omega: Lower-bound | f = Ω(g) is similar to f ≥ g**
**“f grows no slower than g”**

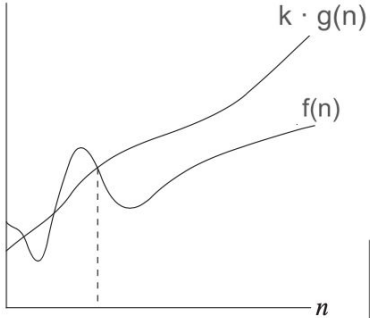


# Recap

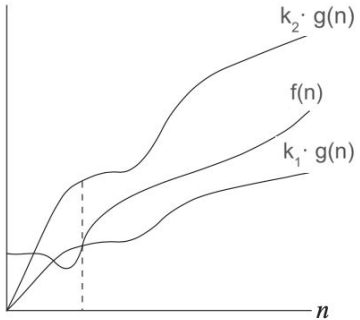
**Big-Oh: Upper-bound | f = O(g) is similar to f ≤ g**
**“f grows no faster than g”**

**Big-Theta: Tight-bound | f = Θ(g) is similar to f = g**
**“f grows as fast as g”**

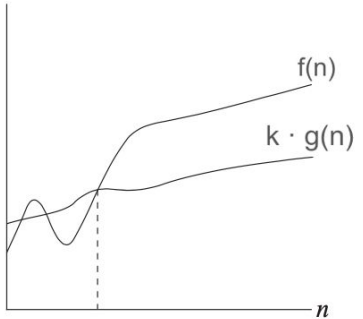
**Big-Omega: Lower-bound | f = Ω(g) is similar to f ≥ g**
**“f grows no slower than g”**




$f(n) = O(g(n))$




$f(n) = \Theta(g(n))$




$f(n) = \Omega(g(n))$

# Quiz!

www.comp285-fall22.ml

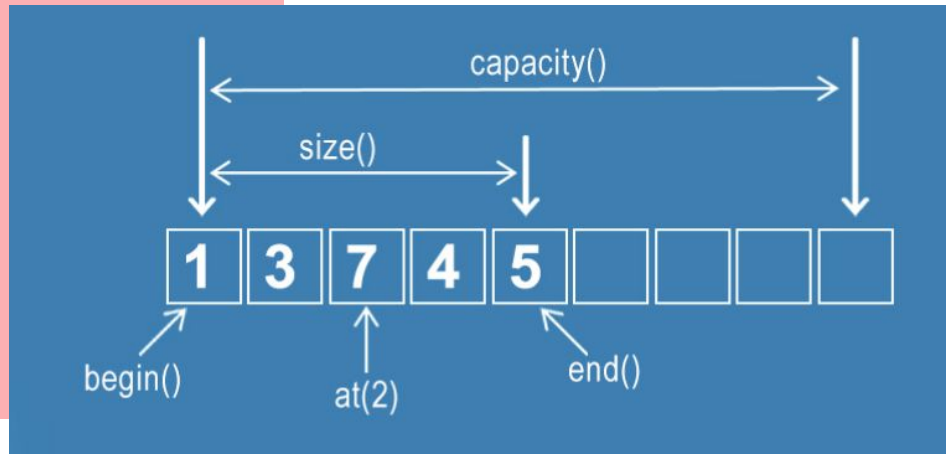
## Big Questions!

- What are conventional data structures again? How fast are they?
- Which data structures use hashing? How fast are they?
- What is hashing?

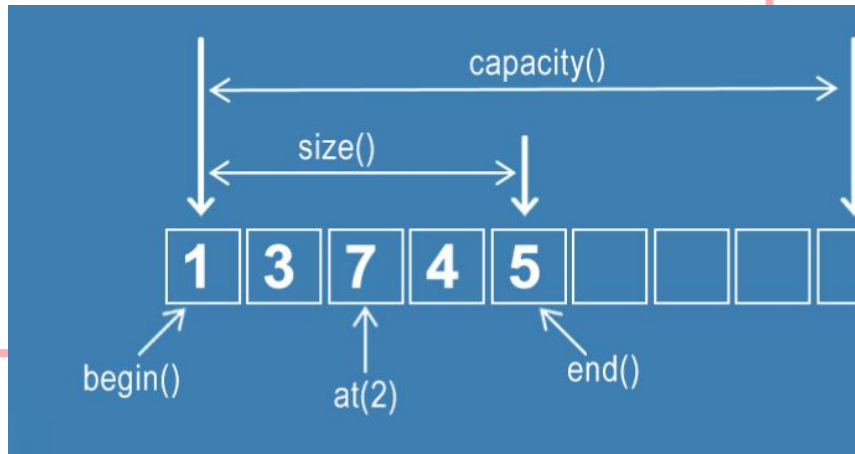
## Big Questions!

- What are conventional data structures again? How fast are they?
- Which data structures use hashing? How fast are they?
- What is hashing?

# Vectors/Arrays

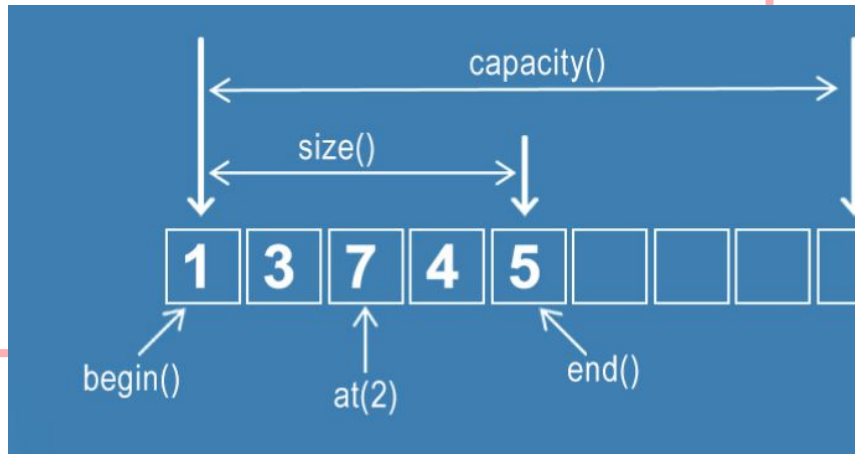


# Array Operations Chart



# Array Operations Chart

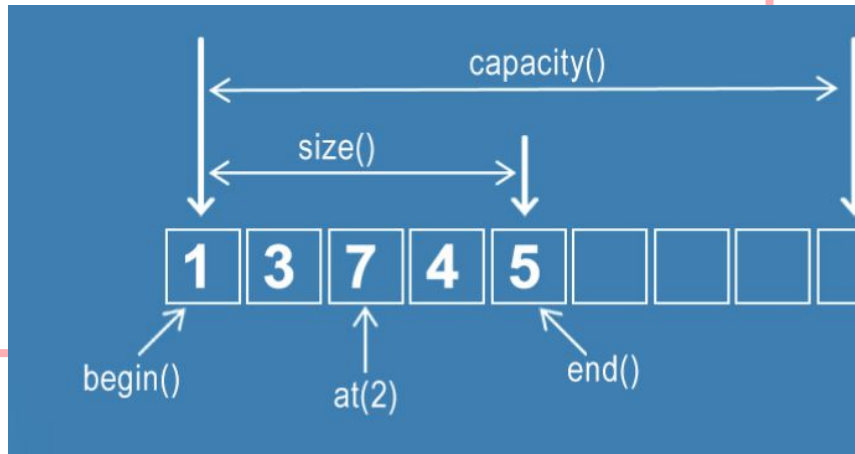
| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Array | | | | |



# Array Operations Chart

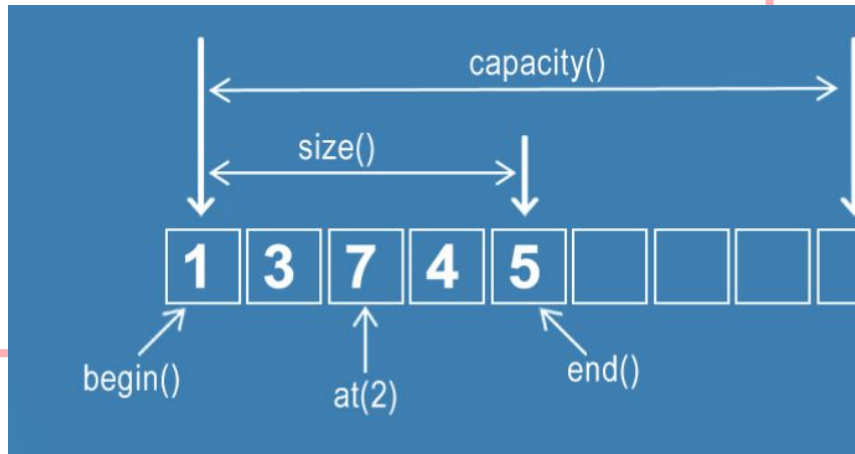
| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Array | | | | |

```
int thirdItem = arr[2];
```



# Array Operations Chart

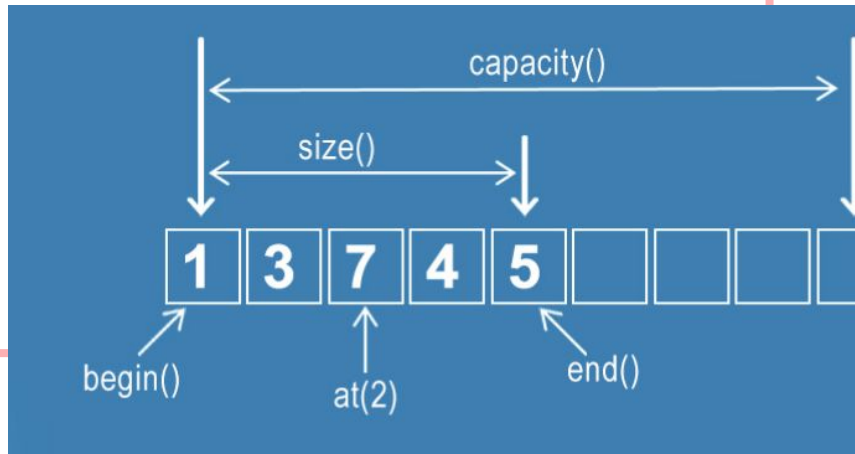
| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Array | O(1) | | | |



# Array Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Array | O(1) | | | |

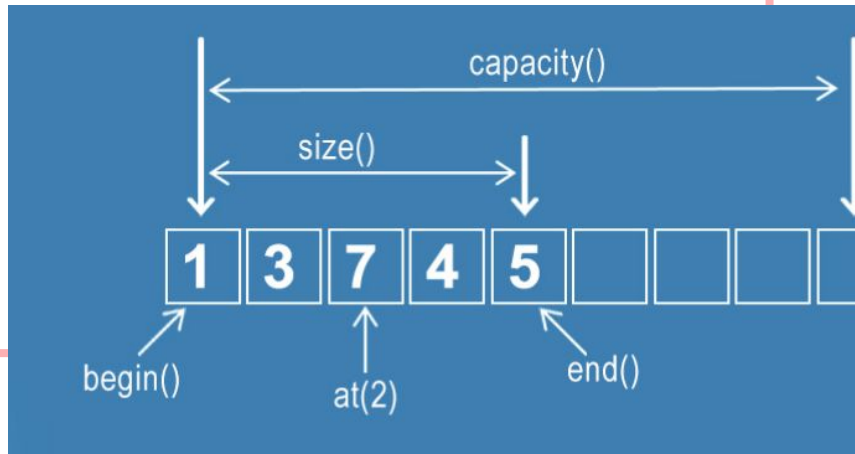
```
for(int i = 0; i < arr.size(); i++) {
    If (arr[i] == target_value) {
        …
    }
}
```



# Array Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Array | O(1) | O(n) | | |

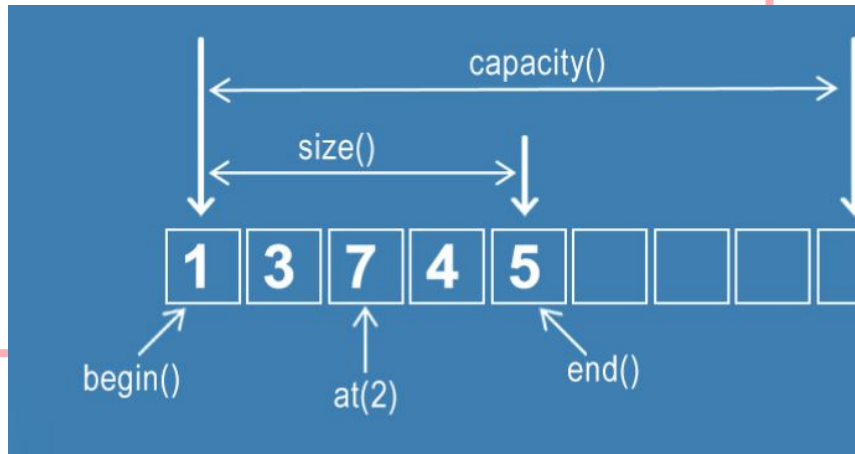
**arr.insert(...),**
**arr.erase(...)**



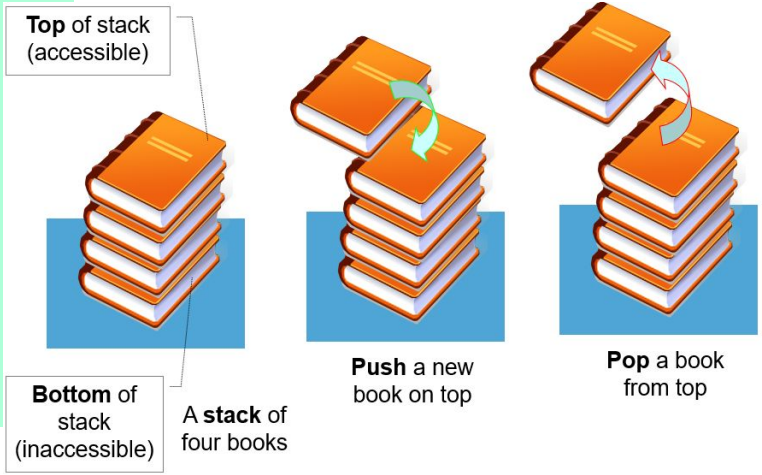
# Array Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Array | O(1) | O(n) | O(n) | O(n) |

**arr.insert(...),**
**arr.erase(...)**

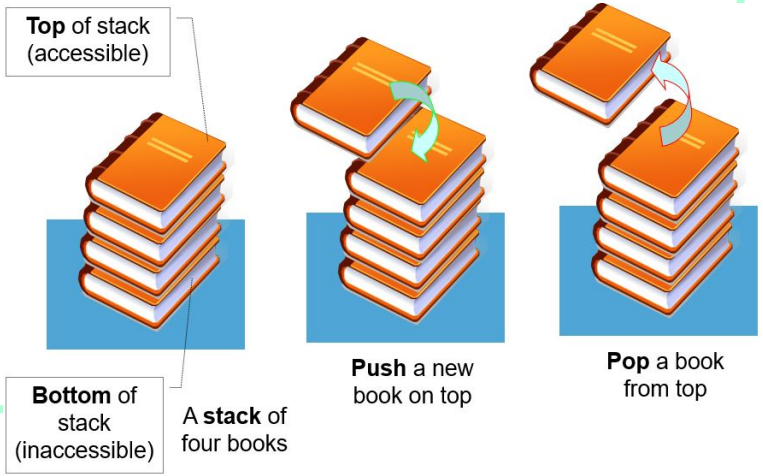


# Stacks


**Top** of stack (accessible)
**Bottom** of stack (inaccessible)
A **stack** of four books
**Push** a new book on top
**Pop** a book from top

# Stack Operations Chart

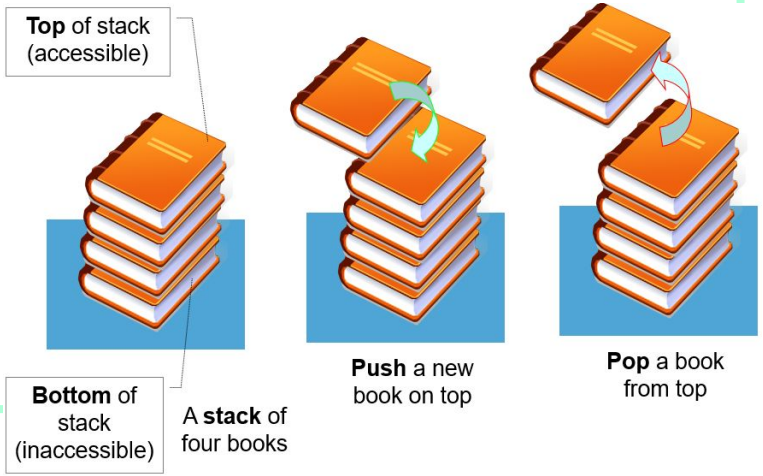
| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Stack | | | | |



# Stack Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Stack | | | | |

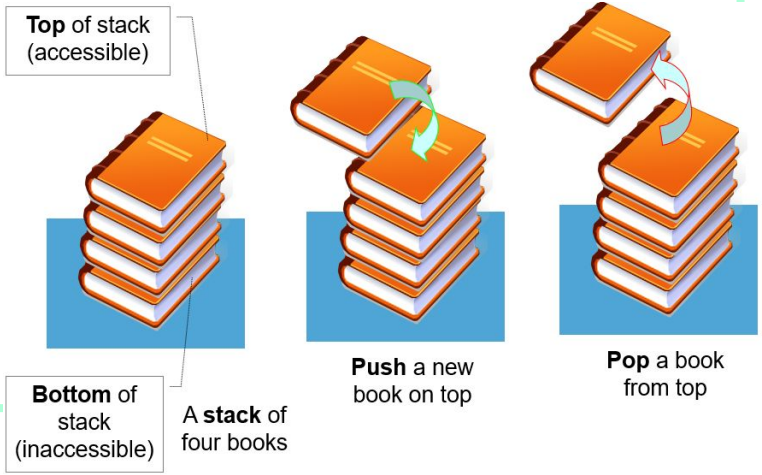
```
while (i < indexToAccess) {
    stack.pop();
    i--;
}
```



# Stack Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Stack | O(n) | | | |

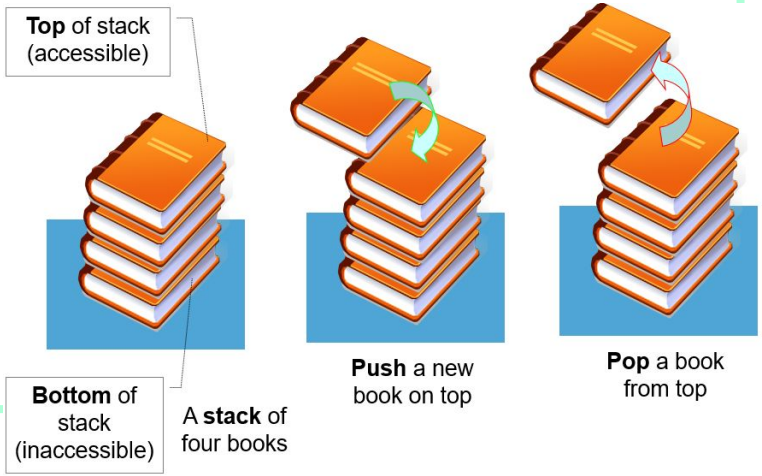
```
while (i < indexToAccess) {
    stack.pop();
    i--;
}
```



# Stack Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Stack | O(n) | | | |

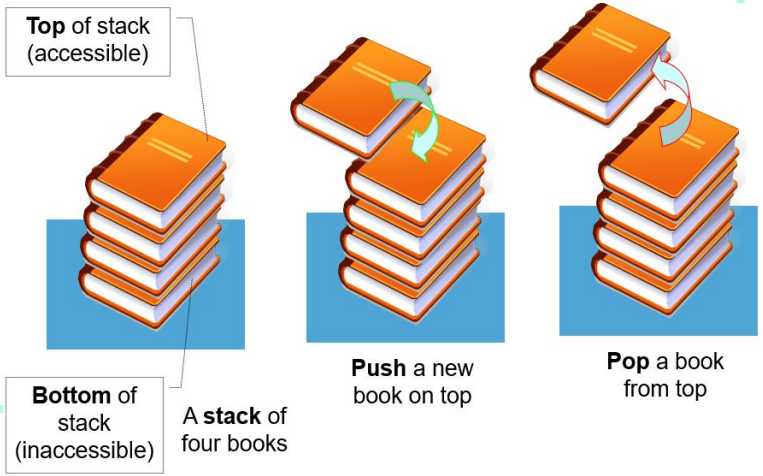
```
while (value != targetValue) {
    value = stack.pop();
}
```



# Stack Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Stack | O(n) | O(n) | | |

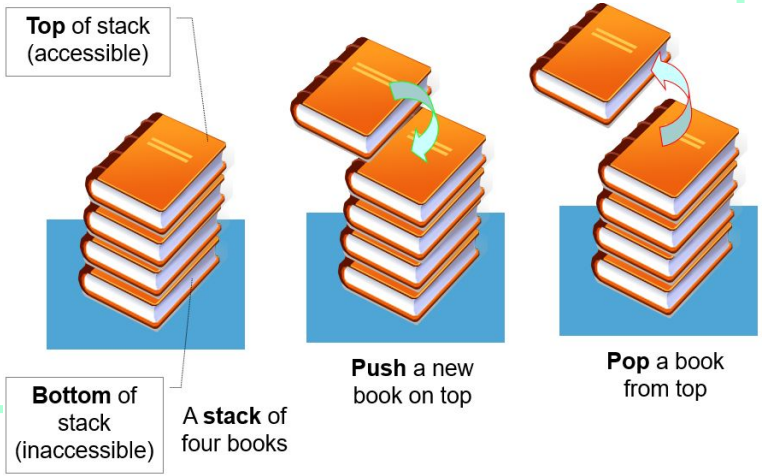
```
while (value != targetValue) {
    value = stack.pop();
}
```



# Stack Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Stack | O(n) | O(n) | | |

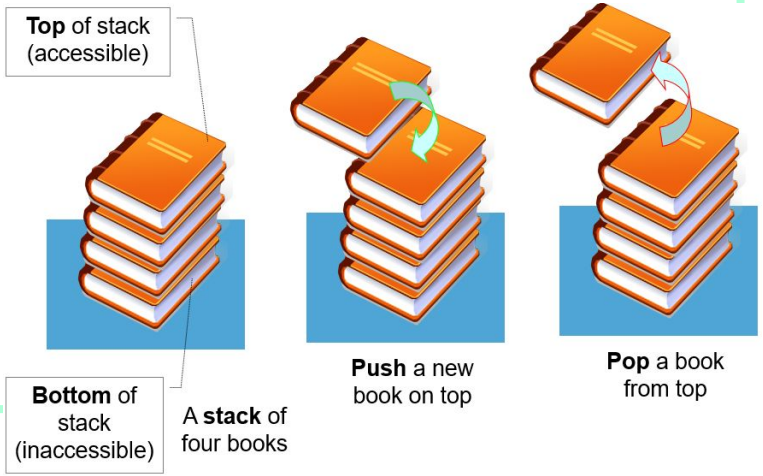
**stack.push(),**
**stack.pop()**



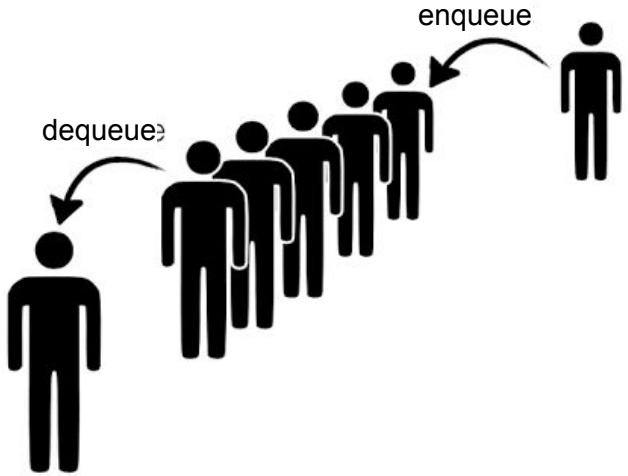
# Stack Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Stack | O(n) | O(n) | O(1) | O(1) |

**stack.push(),**
**stack.pop()**

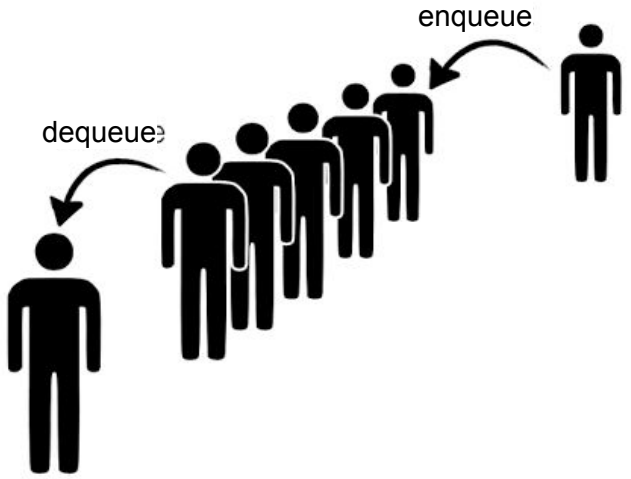


# Queues



# Queue Operations Chart

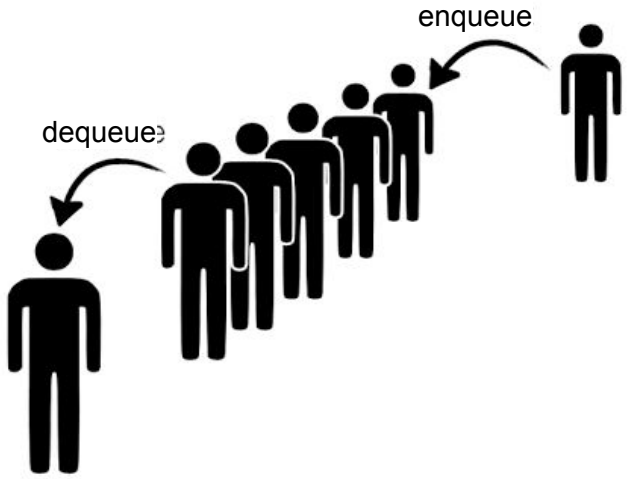
| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Queue | | | | |



# Queue Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Queue | | | | |

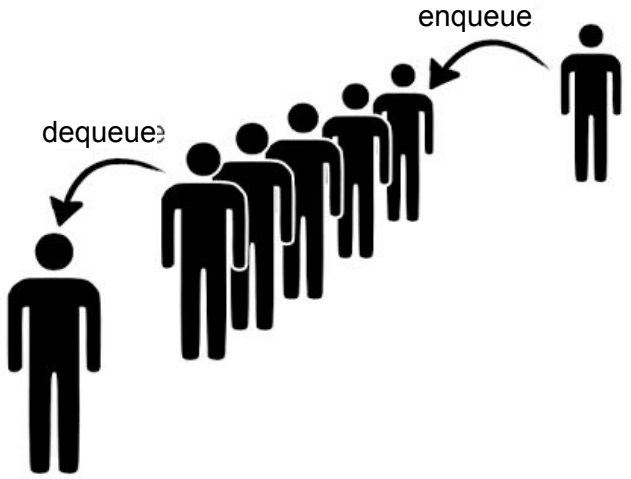
```
while (i < indexToAccess) {
    queue.dequeue();
    i--;
}
```



# Queue Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Queue | O(n) | | | |

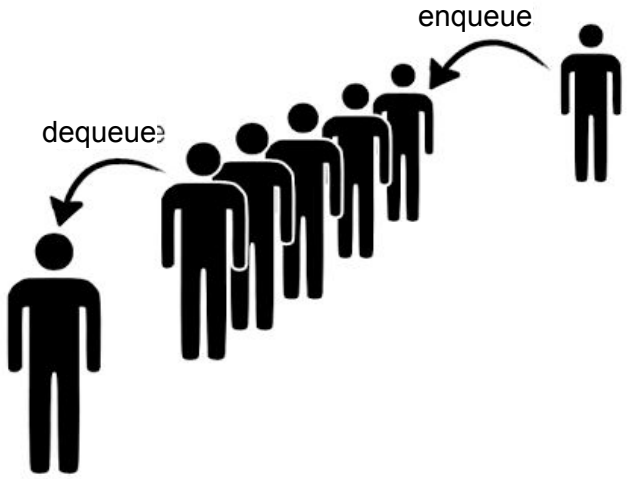
```
while (i < indexToAccess) {
    queue.dequeue();
    i--;
}
```



# Queue Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Queue | O(n) | | | |

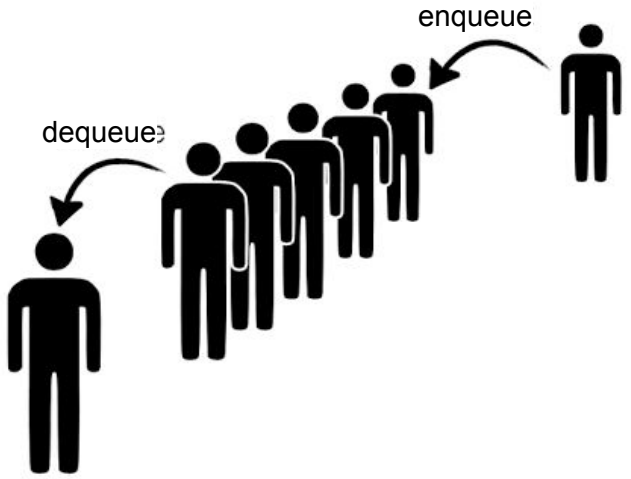
```
while (value != targetValue) {
   value = queue.dequeue();
}
```



# Queue Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Queue | O(n) | O(n) | | |

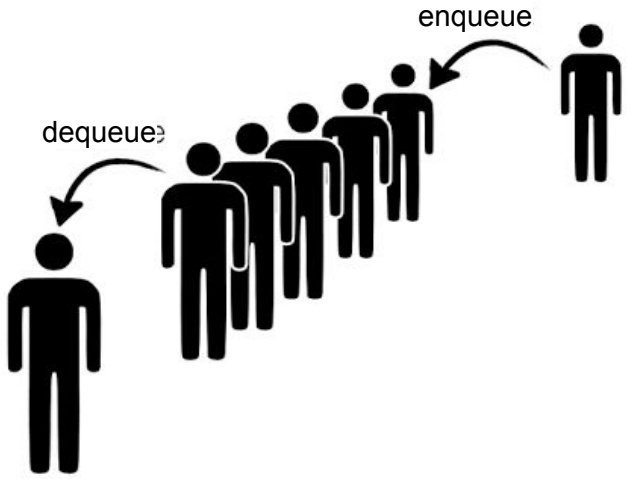
```
while (value != targetValue) {
   value = stack.pop();
}
```



# Queue Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Queue | O(n) | O(n) | | |

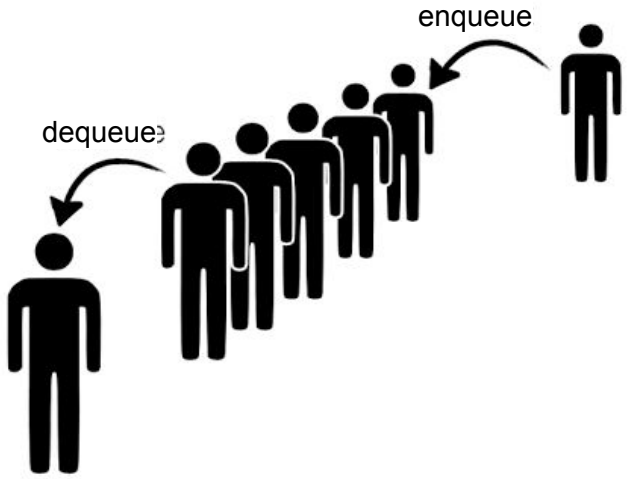
**queue.enqueue(),**
**queue.dequeue()**



# Queue Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Queue | O(n) | O(n) | O(1) | O(1) |

**queue.enqueue(),**
**queue.dequeue()**



# Concrete Examples

› to enhance my programming abilities. In this course I hope to be able to understand and solve more complex algorithms

e I want to better my interviewing and coding skills.

s I want to be able to actually code and like it.

# Concrete Examples

n building efficient algorithms to solve coding interview questions.

enhance my problem solving skills, c

, I want to enhance my coding skills, specifically in C++ to help me earn an internship and a job as a software engineer post grad.

# Balance Parentheses

Given a string containing just char ‘(‘ and char ‘)’, return whether or not the parentheses are valid.

Examples:

() -> true

)( -> false

()(()()) -> true

(() -> false

# Balance Parentheses

Given a string containing just char ‘(‘ and char ‘)’, return whether or not the parentheses are valid.

**Examples:**

() -> true

)( -> false

()(()()) -> true

(() -> false

# Let’s code it!!!

# Kahoot!

www.kahoot.it, Code: 520608

Enter your @aggies.ncat email

# Balance Parentheses V2

Given a string containing chars ‘(‘, ‘{‘ , ‘[‘, ‘)’, ‘}’, ’]’ return whether or not the parentheses are valid.

**Examples:**

([]{}]) -> true

}{( -> false

([{()}])(([])({})) -> true

(({}) -> false

# Let’s code it!!!

## Big Questions!

- What are conventional data structures again? How fast are they?
- Which data structures use hashing? How fast are they?
- What is hashing?

# Hash Sets/Maps

# Hash Sets vs. Maps

# Hash Sets vs. Maps

- A set holds a collection (i.e. unordered) of distinct elements (i.e. no duplicates)
  - Example: {“A”, “B”, “C”}

# Hash Sets vs. Maps

- A set holds a collection (i.e. unordered) of distinct elements (i.e. no duplicates)
  - Example: {“A”, “B”, “C”}
- A map holds a collection (i.e. unordered) of distinct keys (i.e. no duplicates) and their values. Each element is a pair of a key *and its associated value*.
  - Example: {“A”: 2, “B”: 4, “C”: 2}.

# Hash Sets vs. Maps

- A set holds a collection (i.e. unordered) of distinct elements (i.e. no duplicates)
  - Example: {“A”, “B”, “C”}
- A map holds a collection (i.e. unordered) of distinct keys (i.e. no duplicates) and their values. Each element is a pair of a key *and its associated value*.
  - Example: {“A”: 2, “B”: 4, “C”: 2}.
- Implementations of these use hash functions, which helps us quickly decide where to insert/lookup key-value pairs in amortized O(1) time.

# Really O(1) Time?

| Set Operation | Runtime* |
|---|---|
| `insert(x)` | O(1) |
| `remove(x)` | O(1) |
| `contains(x)` | O(1) |
| `empty()` | O(1) |
| `size()` | O(1) |

| Map Operation | Runtime* |
|---|---|
| `put(k, v)` | O(1) |
| `remove(k)` | O(1) |
| `contains(k)` | O(1) |
| `get(k)` | O(1) |
| `empty()` | O(1) |
| `size()` | O(1) |

# Hash Set/Map Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
| --- | --- | --- | --- | --- |
| Hash Set/Map | | | | |

# Hash Set/Map Operations Chart

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Hash Set/Map | N/A | O(1) | O(1) | O(1) |

# Concrete Examples

# Intersecting Arrays

Given two vector<string> which contain lists of names vecA and vecB. Return true whether any name appears in both vecA and vecB.

```
A = {"bob", "sally", "jill"}
B = {"alice", "john", "sally"}
return true

A = {"bob", "sally", "jill"}
B = {"alice", "john"}
return false
```

# Intersecting Arrays

Given two vector<string> which contain lists of names vecA and vecB. Return true whether any name appears in both vecA and vecB.

```
A = {"bob", "sally", "jill"}
B = {"alice", "john", "sally"}
return true

A = {"bob", "sally", "jill"}
B = {"alice", "john"}
return false
```

# Let’s code it!!!

# Kahoot!

www.kahoot.it, Code: 520608

Enter your @aggies.ncat email

# Finding the Mode

Given a vector, identify the mode. The mode is defined as the value that occurs the most number of times. If there is a tie, select any of the valid mode answers.

```
A = {1, 2, 2, -13, 100, 3}
return 2

B = {-1, -1, -1, 0, 10, 0, 0}
return (either -1 or 0)
```

# Finding the Mode

Given a vector, identify the mode. The mode is defined as the value that occurs the most number of times. If there is a tie, select any of the valid mode answers.

```
A = {1, 2, 2, -13, 100, 3}
return 2

B = {-1, -1, -1, 0, 10, 0, 0}
return (either -1 or 0)
```

# Let’s code it!!!

# Kahoot!

www.kahoot.it, Code: 520608

Enter your @aggies.ncat email

# Runtime Comparison

| Data Structure | Access | Search | Insertion | Deletion |
|---|---|---|---|---|
| Array | O(1) | O(n) | O(n) | O(n) |
| Stack | O(n) | O(n) | O(1) | O(1) |
| Queue | O(n) | O(n) | O(1) | O(1) |
| Hash Set/Map | N/A | O(1) | O(1) | O(1) |

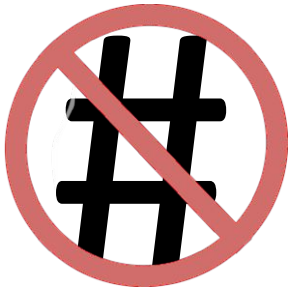
# How do hash sets/maps work?

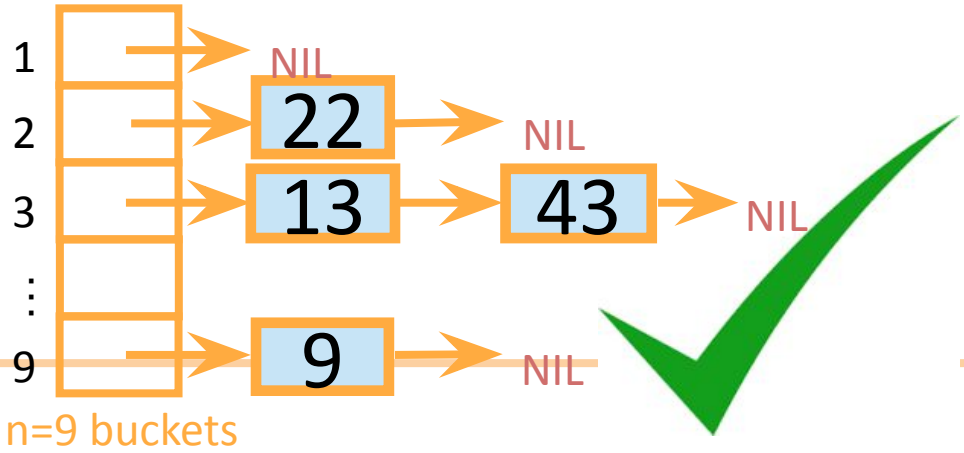
# How do hash sets/maps work?

## Big Questions!

- What are conventional data structures again? How fast are they?
- Which data structures use hashing? How fast are they?
- What is hashing?

# What is Hashing?

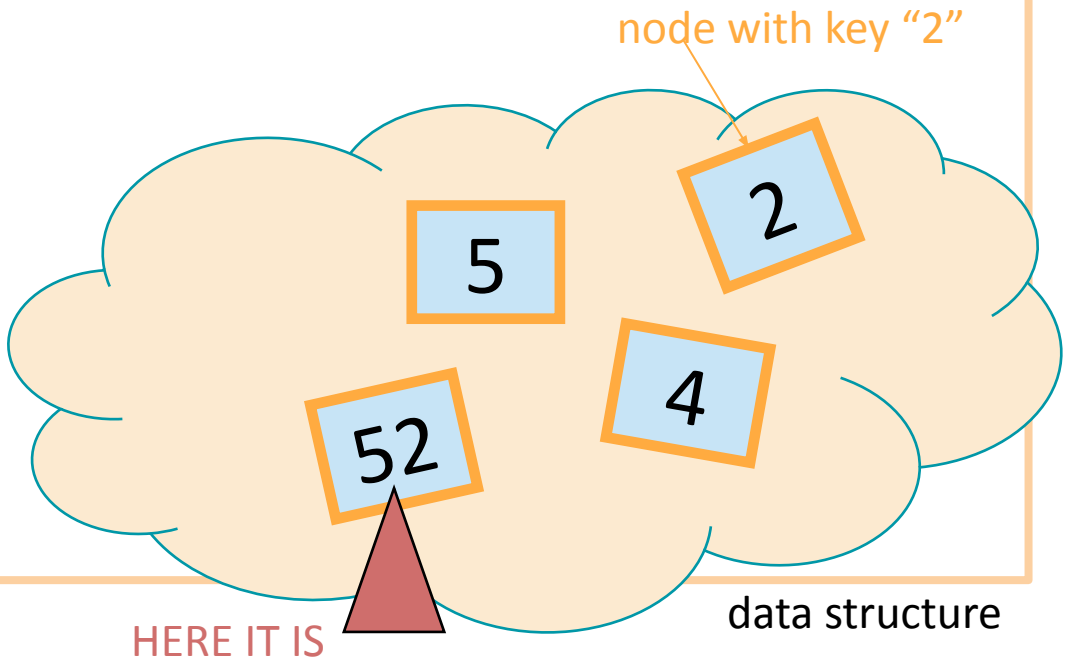




# Goal of Hashing

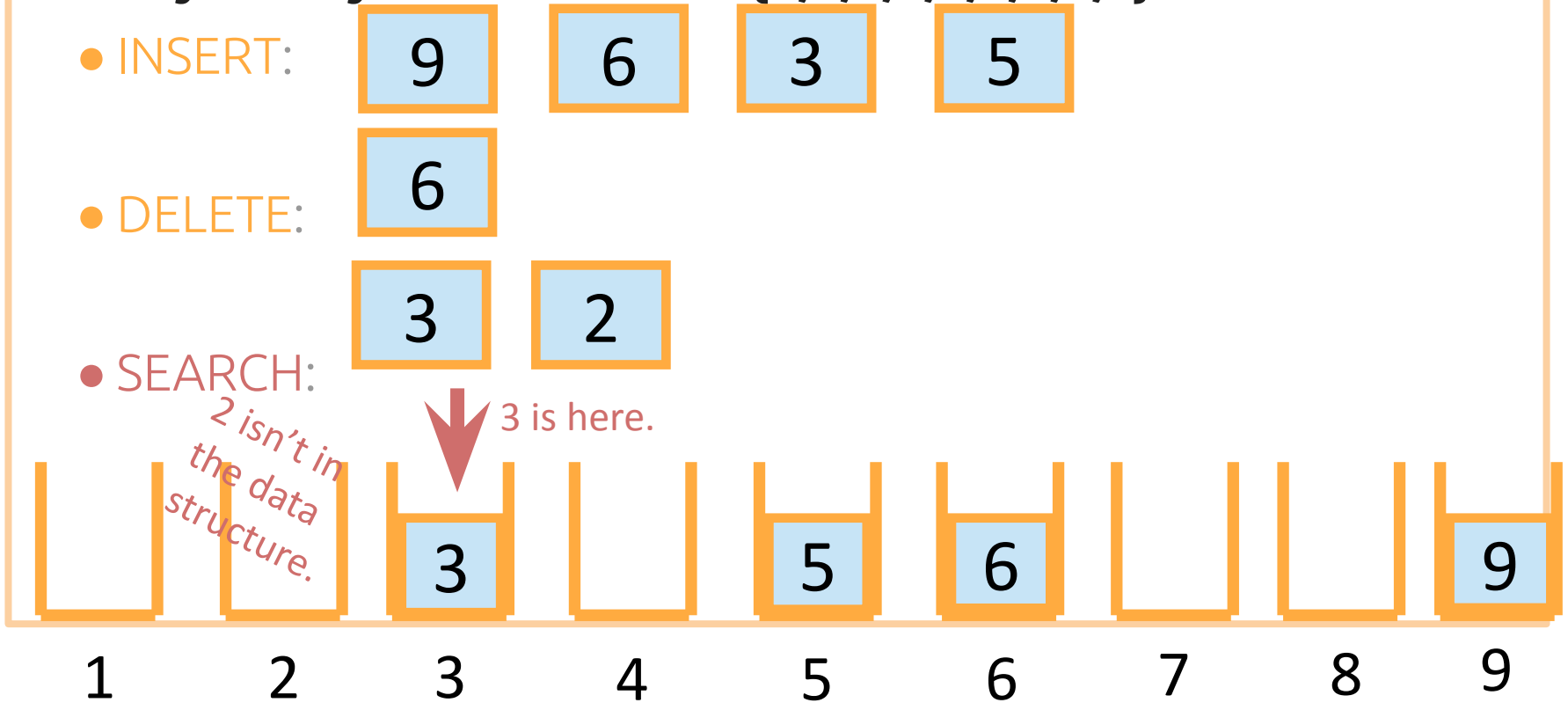
- We want to store nodes with keys in a data structure that supports fast INSERT/DELETE/SEARCH.

- INSERT 5
- DELETE 4
- SEARCH 52



# One way to get O(1) time

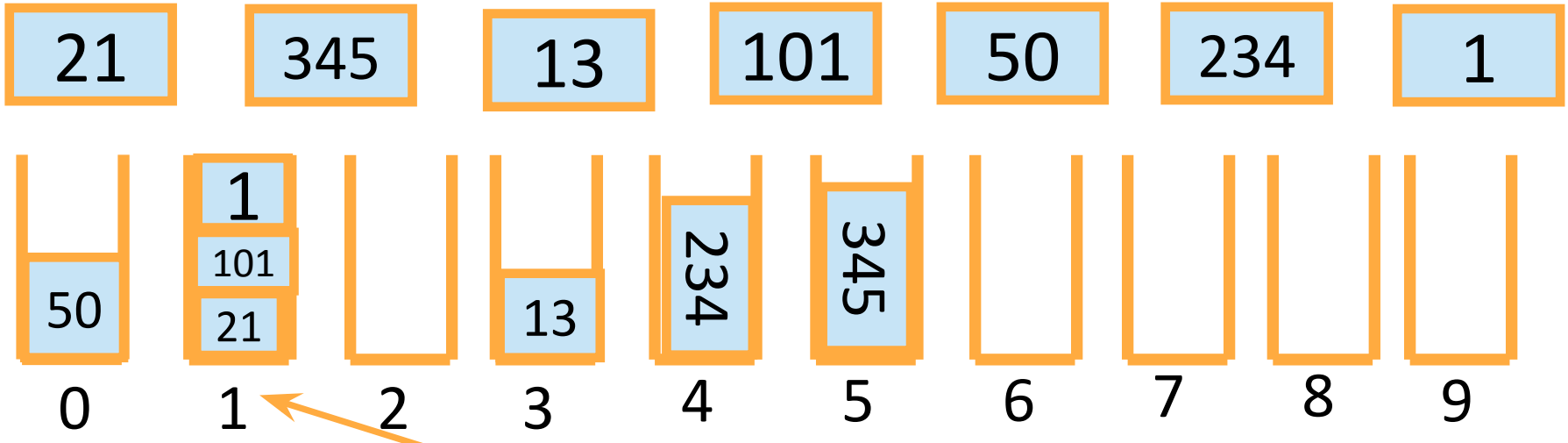
- **Say all keys are in the set {1,2,3,4,5,6,7,8,9}.**
- INSERT:
- DELETE:
- SEARCH:



Put things in buckets based on one digit

# Solution?

INSERT:

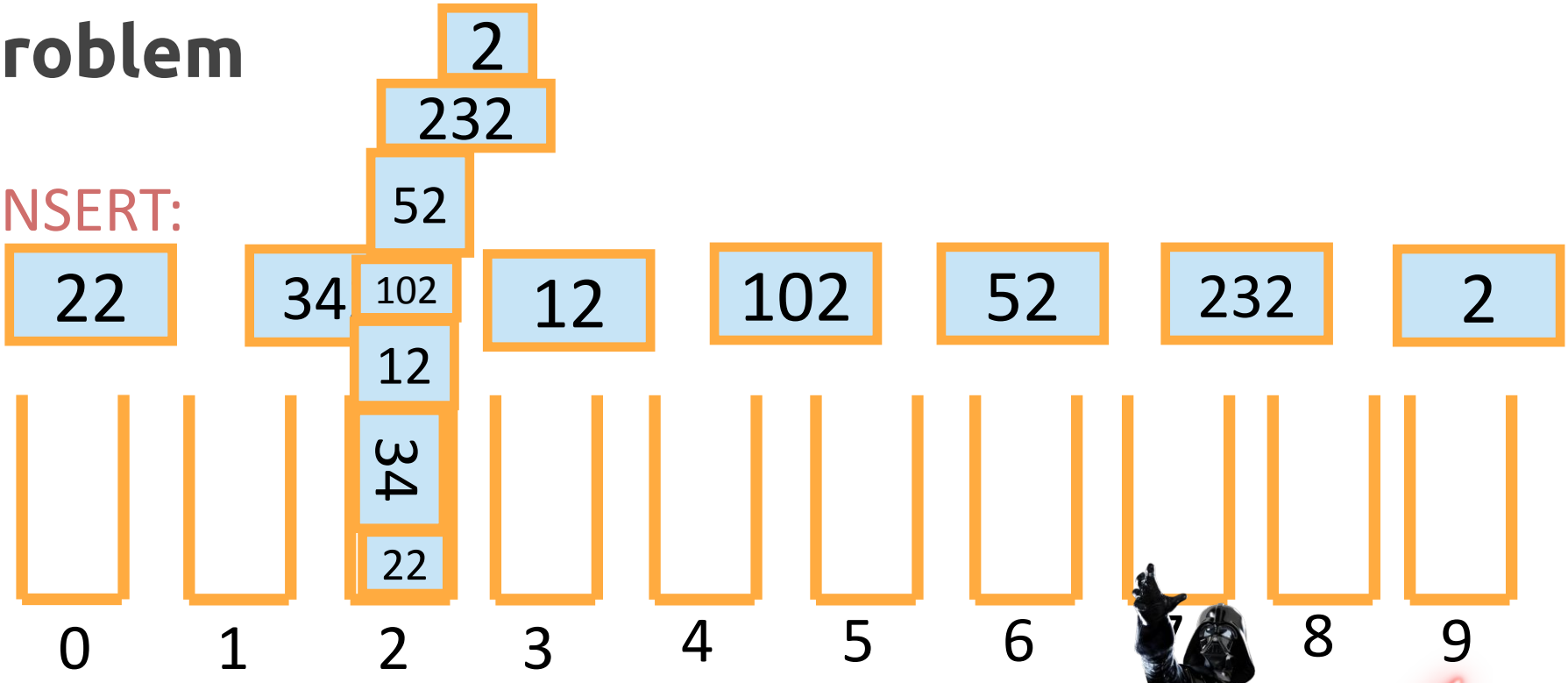



It's in this bucket somewhere…
go through until we find it.

Now SEARCH 21

# Problem

INSERT:



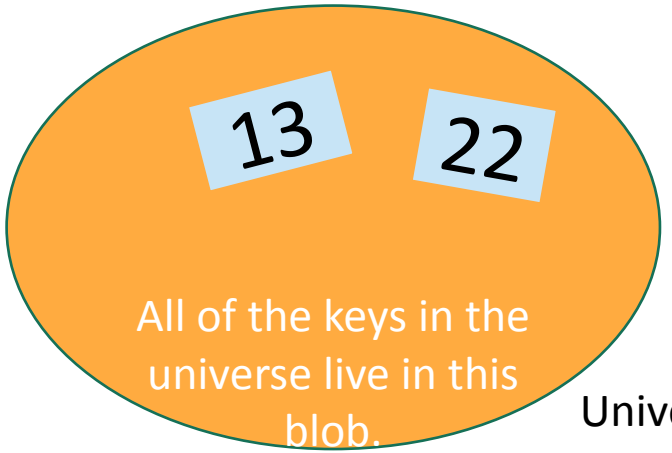

Now SEARCH 22

….this hasn't made our lives easier…

# But first! Terminology.

- U is a *universe* of size M.
  - M is really big.
- But only a few (at most n) elements of U are ever going to show up.
  - M is waaaayyyyyyy bigger than n.
- But we don't know which ones will show up in advance.

**Only n keys will ever show up.**




Universe U

Example: U is the set of all strings of at most 280 ascii characters. ($128^{280}$ of them).

The only ones which I care about are those which appear as trending hashtags on twitter. #hashinghashtags

*There are way fewer than $128^{280}$ of these.*

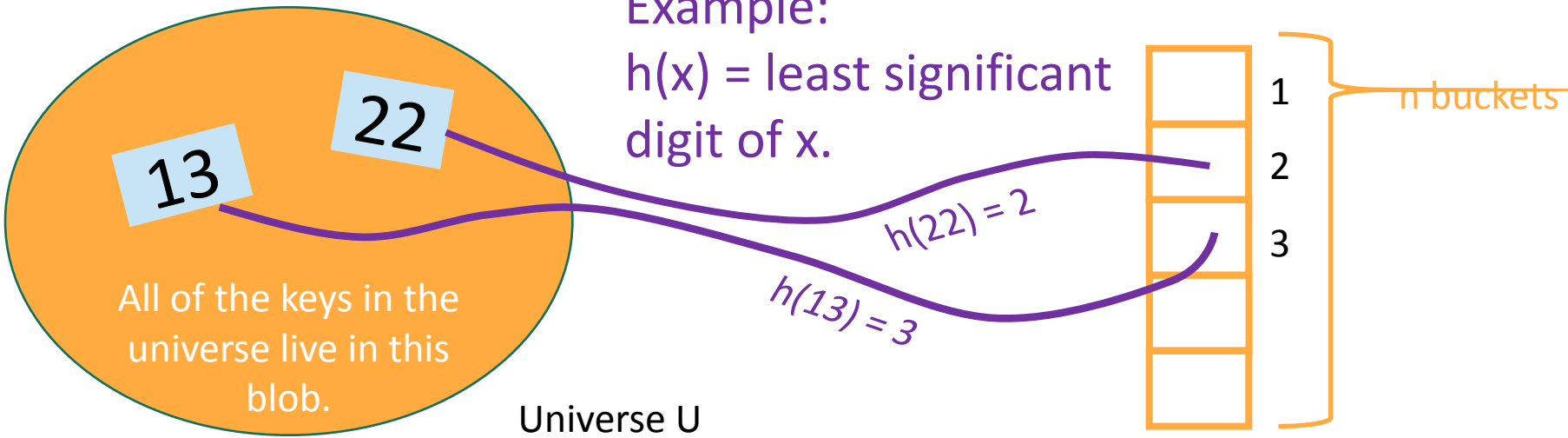
# Hash Functions

- A *hash function* $h: U \to \{1, \ldots, n\}$ is a function that maps elements of U to buckets 1, …, n.

For this lecture, we are assuming that the number of things that show up is the same as the number of buckets, both are n.

This doesn't have to be the case, although we do want:

**#buckets = O( #things which show up )**



# Hash Tables (with chaining)

- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- A hash function $h: U \to \{1, \ldots, n\}$.
  - For example, h(x) = least significant digit of x.

**For demonstration purposes only!**
This is a terrible hash function!

INSERT:

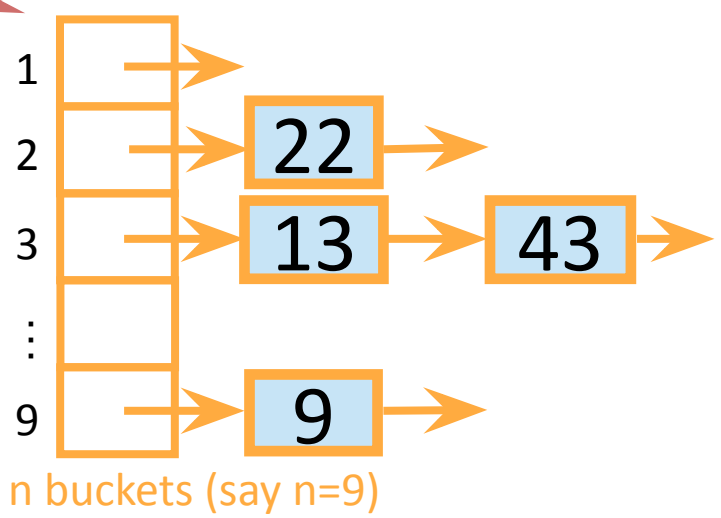
13 22 43 9

SEARCH 43:

Scan through all the elements in bucket h(43) = 3.

DELETE 43:

Search for 43 and remove it.



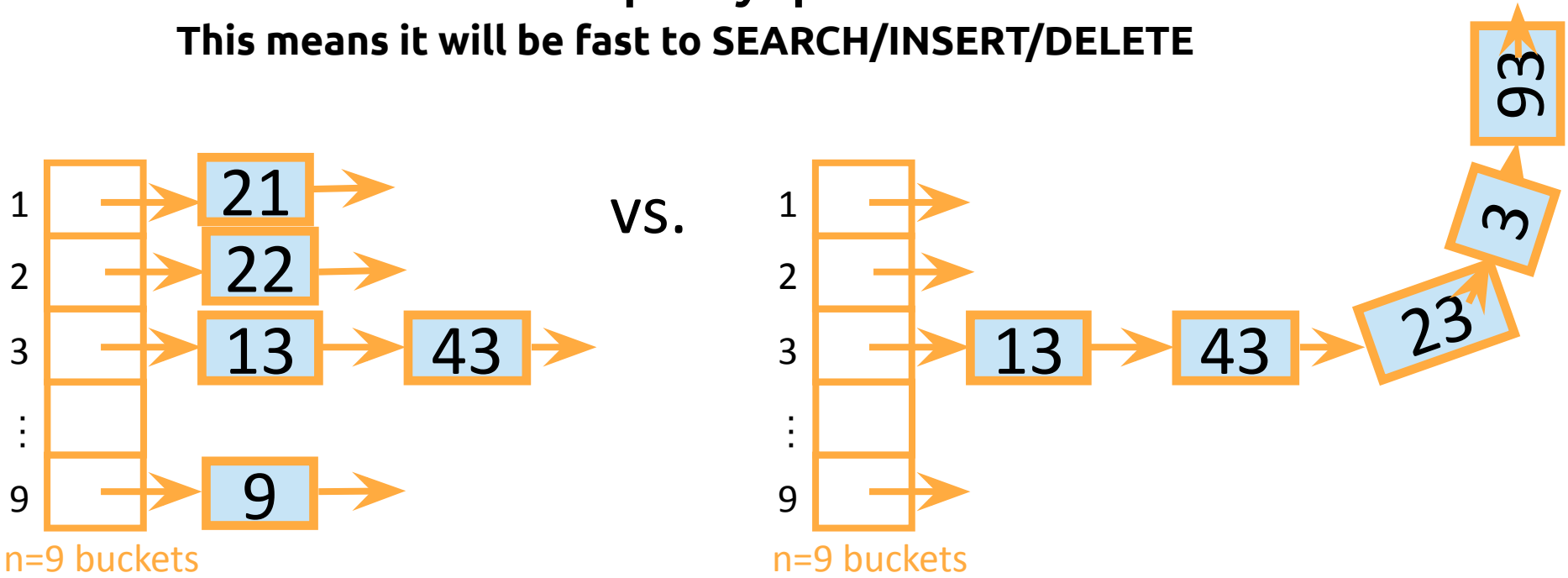
# Ideal Hash Tables

**We want there to be not many buckets (say, n).**

**This means we don't use too much space**

**We want the items to be pretty spread-out in the buckets.**

**This means it will be fast to SEARCH/INSERT/DELETE**



# Hashing for Cybersecurity

- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

# Hashing for Cybersecurity

- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

# Hashing for Cybersecurity

- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

**PorridgeQueen123**!

# Hashing for Cybersecurity

- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

**PorridgeQueen123**!

h(x)

# Hashing for Cybersecurity

- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

**PorridgeQueen123**!

h(x)

**ad8fd102jnfpa021j…**

# Hashing for Cybersecurity

- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

# Hashing for Cybersecurity

- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

# Hashing for Cybersecurity

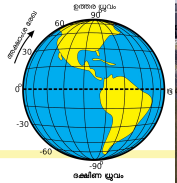
- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

# Hashing for Cybersecurity

- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

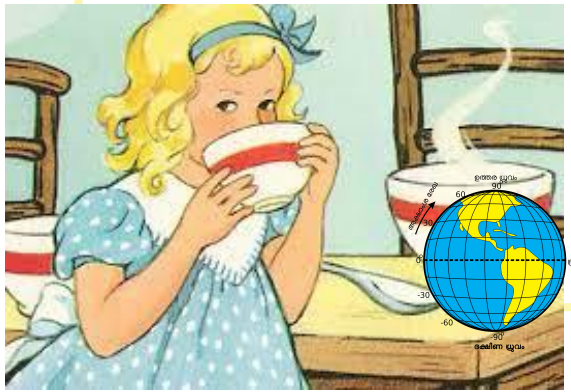
# Hashing for Cybersecurity

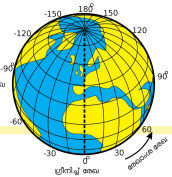
- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.



# Hashing for Cybersecurity

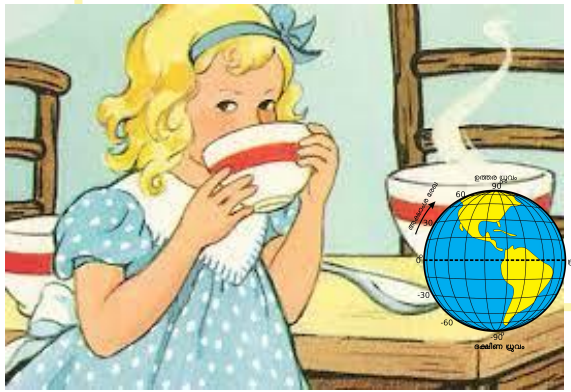
- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

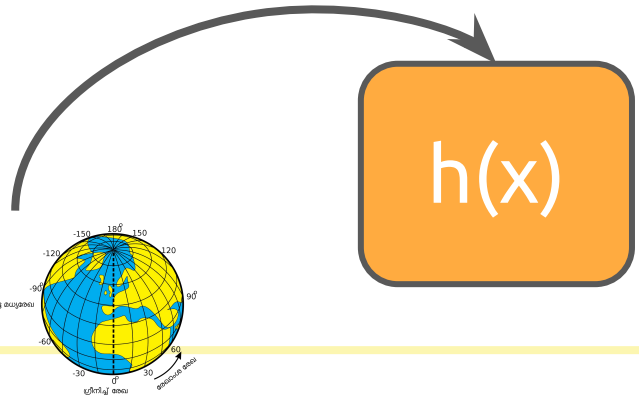




# Hashing for Cybersecurity

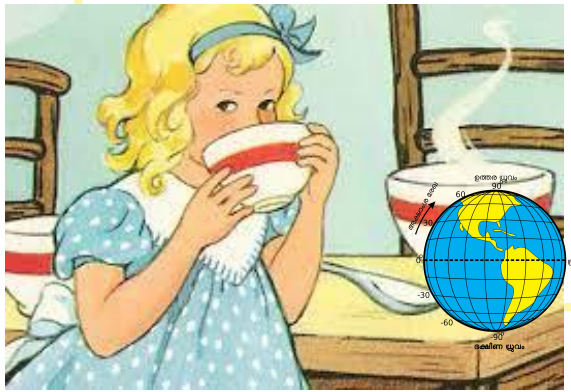
- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

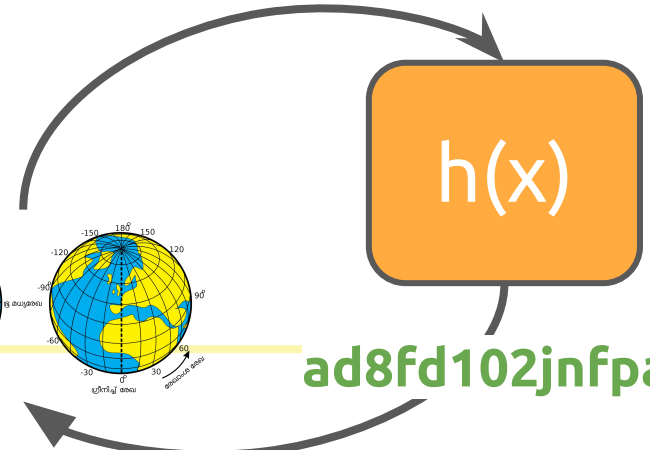





# Hashing for Cybersecurity

- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.






# Hashing for Cybersecurity

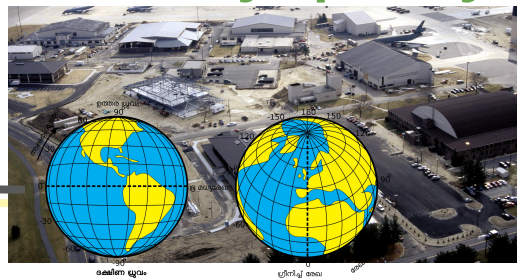
- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

# Hashing for Cybersecurity

- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

h(x)

ad8fd102jnfpa021j...



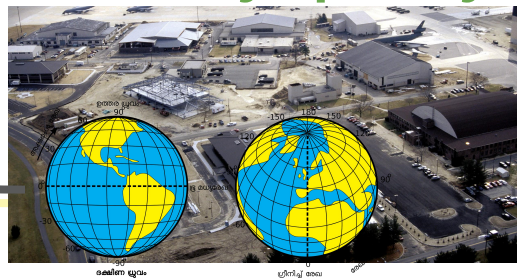
# Hashing for Cybersecurity

- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

ad8fd102jnfpa021j...

h(x)

ad8fd102jnfpa021j...



# Hashing for Cybersecurity

- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

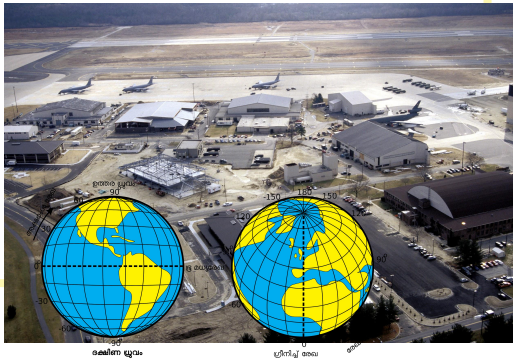
ad8fd102jnfpa021j...

ad8fd102jnfpa021j...

# Hashing for Cybersecurity

- Cryptographic hash function: algorithm that takes an arbitrary amount of data input (credentials, top secret documents, passwords, etc.) and produces a fixed-size output of enciphered text called a hash value, or just “hash.” That enciphered text can then be stored instead of the sensitive data itself, and later used for verification.

h(x)

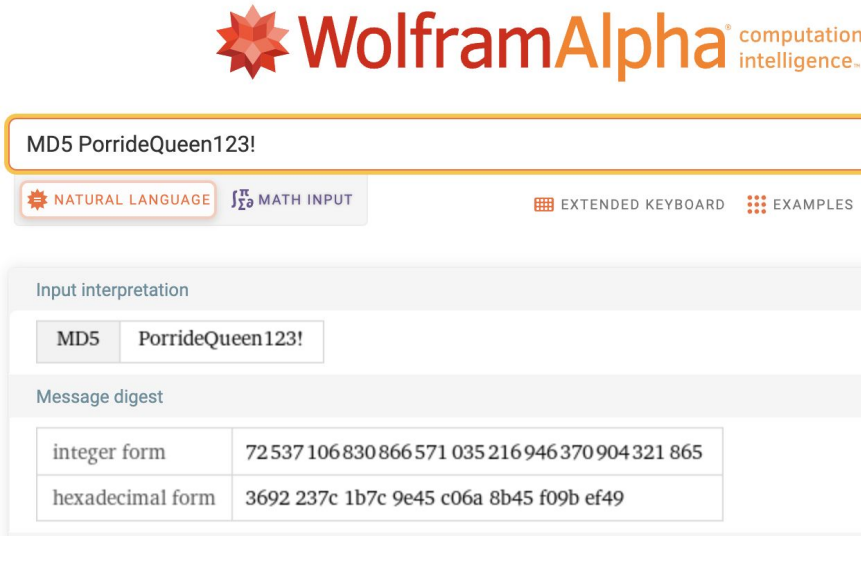


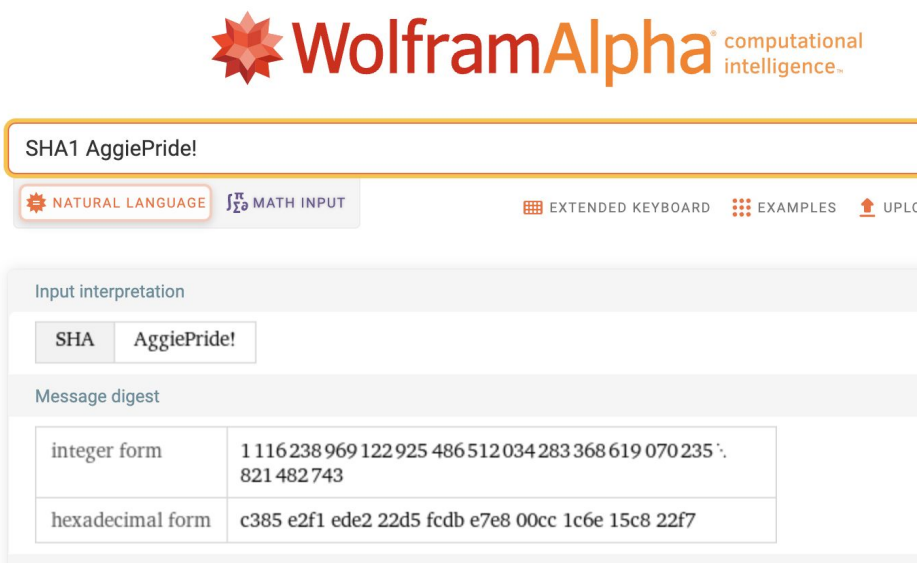
# Hashing for Cybersecurity

- **Non-reversibility:** or one-way function. A good hash should make it very hard to reconstruct the original password from the output or hash.
- **Diffusion, or avalanche effect:** A change in just one bit of the original password should result in change to half the bits of its hash. In other words, when a password is changed slightly, the output of enciphered text should change significantly and unpredictably.
- **Determinism**: A given password must always generate the same hash value or enciphered text.
- **Collision resistance**: It should be hard to find two different passwords that hash to the same enciphered text.
- **Non-predictable**: The hash value should not be predictable from the password.

# Examples of Hash Functions

- Message Digest 5 (MD5)
- Secure Hashing Algorithm 1 and 2



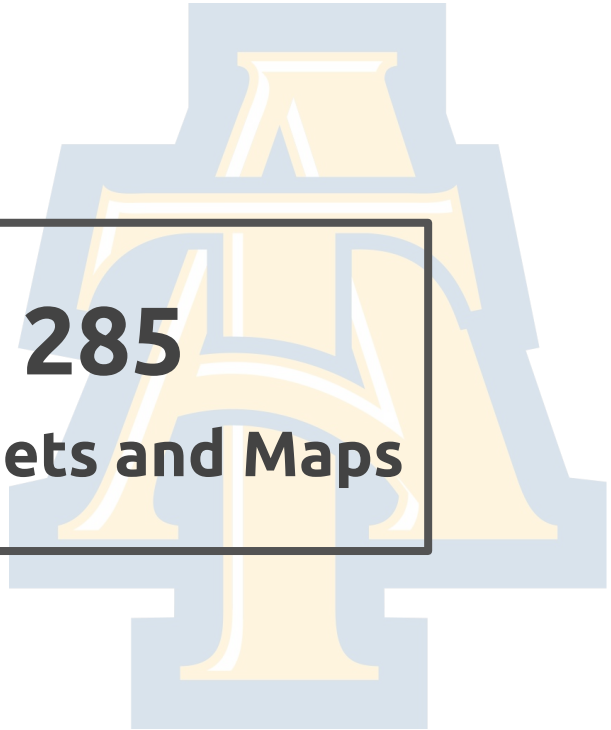



COMP - 285

Advanced Analysis of Algorithms

# Welcome to COMP 285

## Lecture 5: Stacks, Queues, Sets and Maps



**Chris Lucas (cflucas@ncat.edu)**