# HASHING

*Source: UC Davis, ECS 36C course, Spring 2020*

# Introduction

- Symbol table
  - Association of a value with a key
    - `table[key] = value`
  - Typical operations
    - `Insert(), Remove(), Contains()/Get()`
  - Other possible operations
    - `Min(), Max(), Rank(), SortedList(), etc.`

- Example #1: frequency of letters
  - Compute the frequency of all the (alphabetical) characters in a text

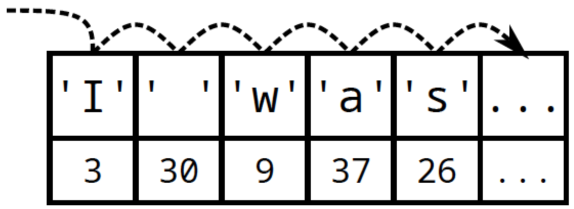
> *I was born in Tuckahoe, near Hillsborough, and about twelve miles from Easton, in Talbot county, Maryland. I have no accurate knowledge of my age, never having seen any authentic record containing it. By far the larger part of the slaves know as little of their ages as horses know of theirs, and it is the wish of most masters within my knowledge to keep their slaves thus ignorant. I do not remember to have ever met a slave who could tell of his birthday. They seldom come nearer to it than planting-time, harvest-time, cherry-time, spring-time, or fall-time. [...]*

| Key | Value |
|---|---|
| 'a' | 37 |
| 'b' | 6 |
| 'c' | 10 |
| 'd' | 10 |
| 'e' | 54 |
| 'g' | 11 |
| ... | ... |

# Naïve Implementations

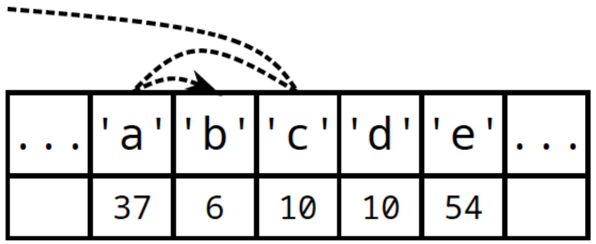
- Unordered array
  - Linear search: Sequentially check each item until match is found

| Search | Insert | Delete |
| --- | --- | --- |
| $O(N)$ | $O(1)$ | (search time +) $O(1)$ |




- Ordered/sorted array
  - Binary search: Can perform binary search to find item

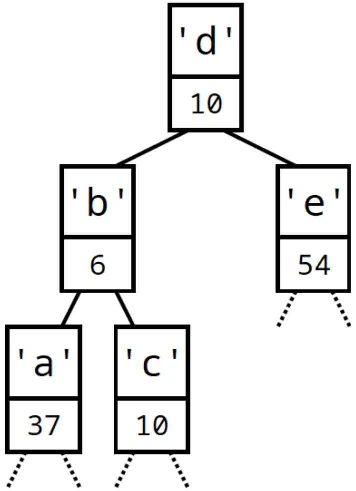
| Search | Insert | Delete |
| --- | --- | --- |
| $O(\log N)$ | $O(N)$ | $O(N)$ |



# Advanced Implementations

- Binary search tree: Organization of data items in BST, following certain order
  - Any node's key is greater than all keys in its left subtree
  - Any node's key is smaller than all keys in its right subtree
- At most, explore height of tree

| Search avg/worst | Insert avg/worst | Delete avg/worst |
|---|---|---|
| $O(\log N)/O(N)$ | $O(\log N)/O(N)$ | $O(\log N)/O(N)$ |




- AVL tree
  - Balanced BST
  - Height of tree is kept to $O(\log N)$

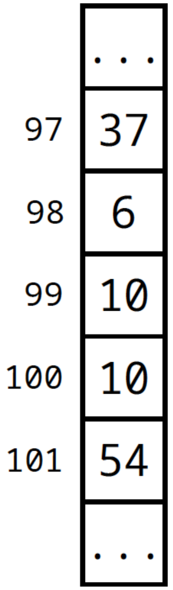
| Search | Insert | Delete |
|---|---|---|
| $O(\log N)$ | $O(\log N)$ | $O(\log N)$ |

# Ideal Data Structure

- Can we achieve constant time on all three main operations?

| Search | Insert | Delete |
|---|---|---|
| $O(1)$ | $O(1)$ | $O(1)$ |

- Using data as index
  - Keys are small integers
  - ASCII is 128 characters
  - Can directly be used as index
    - `table[key] = value`




- Limitations
  - Doesn't (efficiently) support other operations – `min()`, `max()` etc.
  - Wasted memory for unused keys – e.g., ASCII defines about 30 control characters
  - What if the key is not a small integer?

# Example #2: frequency of words

- Compute the frequency of all the words in a text

> *I was born in Tuckahoe, near Hillsborough, and about twelve miles from Easton, in Talbot county, Maryland. I have no accurate knowledge of my age, never having seen any authentic record containing it. By far the larger part of the slaves know as little of their ages as horses know of theirs, and it is the wish of most masters within my knowledge to keep their slaves thus ignorant. I do not remember to have ever met a slave who could tell of his birthday. They seldom come nearer to it than planting-time, harvest-time, cherry-time, spring-time, or fall-time. [...]*

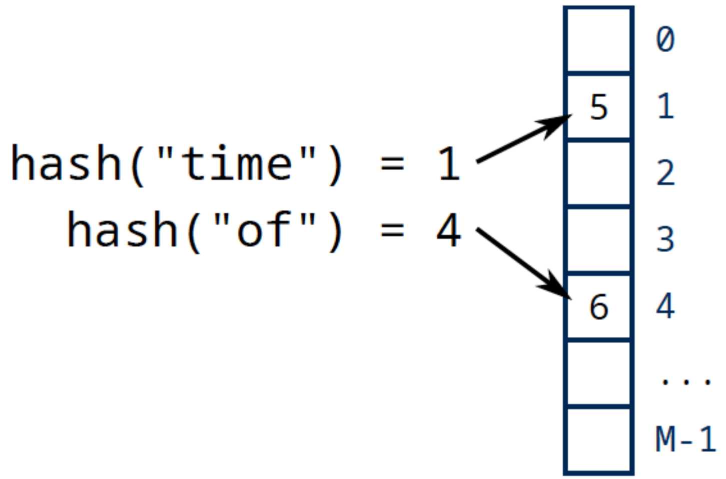
- Issues
  - Very large number of words in English
    - Would need a huge table to index by word
    - `int table[171476];`
  - Cannot index an array with a string
    - table["time"] is not a proper C construct
    - Can only index arrays with an integer

| Key | Value |
|---|---|
| 'of' | 6 |
| 'time' | 5 |
| 'to' | 3 |
| 'the' | 3 |
| 'it' | 3 |
| 'I' | 3 |
| ... | ... |

# Hashing

- Compute an array index from **any** key
- With direct addressing, a key $k$ is stored in slot $k$
- With hashing, this element is stored in slot $h(k)$

- We use a hash function $h$ to compute the slot from the key $k$
- $h$ maps the universe $U$ of keys into the slots of a hash table $T[0, \ldots, M-1]$
- $h: U \rightarrow \{0,1,\ldots,M-1\}$
- The size $M$ of the hash table is typically much less than $|U|$
- We say that an element with key $k$ **hashes to** slot $h(k)$
- And $h(k)$ is the hash value of key $k$

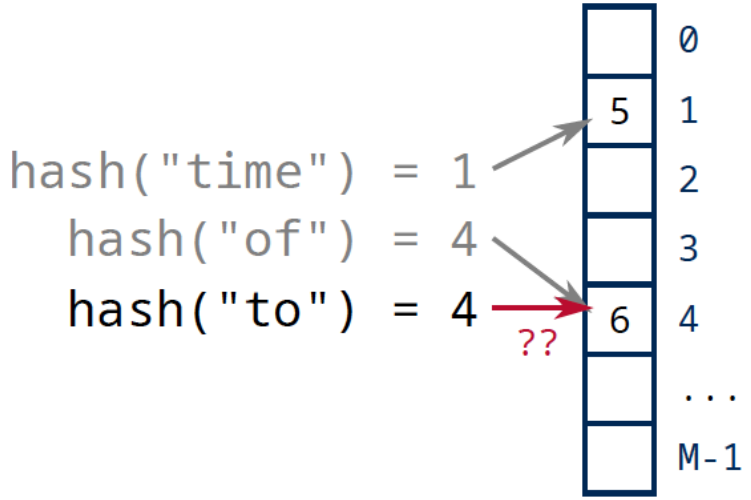


# Hashing

- We will see how to provide hashing for any type of key, e.g., Boolean, integers, floating-point numbers, strings, user-defined objects, etc.

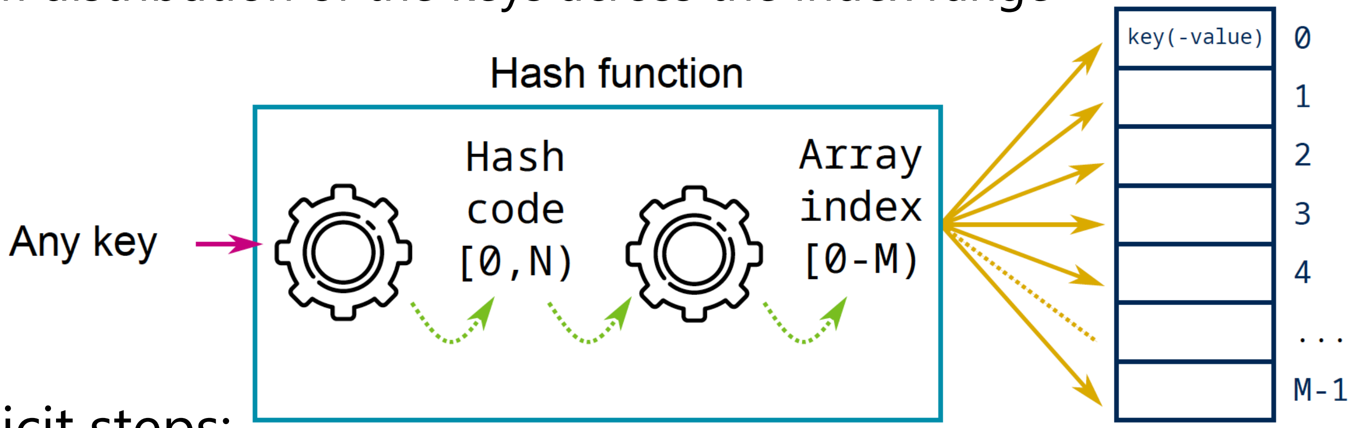
- Two keys may hash to the same slot. Such a situation is called a **collision**

- How to resolve potential collisions properly?



# Hashing

- Before going to collisions, lets discuss some hashing techniques
- Required properties:
  - Simple to compute and fast
  - Hash of a key has to be deterministic
    - Equal keys must produce the same hash value
    - `assert(a == b && hash(a) == hash(b));`
  - Uniform distribution of the keys across the index range




- Two implicit steps:
  - Transform any potentially non-integer key into an integer hash code
  - Reduce this hash code to an index range

# Some Pitfalls

- Worse (yet functional) hash function

```
def GalaxyHashCode(obj: Any) -> int:
  return 42;
```

- Problem
  - Can only insert one key
  - All the following keys will collide, regardless of their value

```
print("Hash of 'time' => {:d}".format(GalaxyHashCode('time')))
print("Hash of 'of' => {:d}".format(GalaxyHashCode('of')))
print("Hash of '99fdr' => {:d}".format(GalaxyHashCode('99fdr')))
print("Hash of '55' => {:d}".format(GalaxyHashCode(55)))
```

```
Hash of 'time' => 42
Hash of 'of' => 42
Hash of '99fdr' => 42
Hash of '55' => 42
```

- Solution
  - Need to distinguish keys by value

# Key Subset Selection

- Possible scenario
  - Keys are phone numbers: e.g., (530) 752-1011
  - Hash table is of size 1000
  - Idea: use first three digits as a direct index?
    - `hash("(530) 752-1011") = 530`

- Problem
  - Phone numbers with same area code will collide
- Solution
  - Select another set of 3 digits that shows better "random" properties (e.g., the last three)
  - Or consider the entire phone number instead

# Hashing Positive Integers

- Key is a simple unsigned integer in the range 0..K-1
- Array index is always an unsigned integer in the range 0..M-1
- Hashing is done via modulo operation

```
def HashUInt(obj: np.uint32, M: int) -> int:
  return obj % M
```

```
# Initialize hash table of size 100
ht = np.full((100), False, dtype=bool)

# Uniform random number generator
distr = np.random.permutation(999)[:25]

for i in distr:
  hash = HashUInt(i, ht.shape[0])
  print("\nHash of {:d} => {:d}".format(i,hash), end="")
  if (ht[hash]):
    print(" (collision!)",  end="")
  ht[hash] = True
```

```
Hash of 562 => 62
Hash of 833 => 33
Hash of 764 => 64
Hash of 981 => 81
Hash of 294 => 94
Hash of 572 => 72
Hash of 605 => 5
Hash of 337 => 37
Hash of 854 => 54
Hash of 939 => 39
Hash of 292 => 92
Hash of 356 => 56
Hash of 364 => 64 (collision!)
Hash of 648 => 48
Hash of 990 => 90
Hash of 753 => 53
Hash of 340 => 40
Hash of 442 => 42
Hash of 20 => 20
Hash of 623 => 23
Hash of 31 => 31
Hash of 453 => 53 (collision!)
Hash of 954 => 54 (collision!)
Hash of 545 => 45
Hash of 847 => 47
```

# Consideration about table size (1/2)

- Does the size of the hash table matter?
- Same scenario as previous slide
  - Keys are random numbers between 0 and 999
  - Distribution of keys is uniform
  - Hashing of 25 keys
- But hash table can either be of size 100 or size 97

- Conclusion
  - Table size is not critical
  - If distribution of keys is uniform

| 100 | 97 |
|---|---|
| 461 => 61 | 461 => 73 |
| 741 => 41 | 741 => 62 |
| 412 => 12 | 412 => 24 |
| 482 => 82 | 482 => 94 |
| 522 => 22 | 522 => 37 |
| 19 => 19 | 19 => 19 |
| 941 => 41 (c!) | 941 => 68 |
| 569 => 69 | 569 => 84 |
| 634 => 34 | 634 => 52 |
| 763 => 63 | 763 => 84 (c!) |
| 476 => 76 | 476 => 88 |
| 191 => 91 | 191 => 94 (c!) |
| 898 => 98 | 898 => 25 |
| 796 => 96 | 796 => 20 |
| 583 => 83 | 583 => 1 |
| 370 => 70 | 370 => 79 |
| 331 => 31 | 331 => 40 |
| 883 => 83 (c!) | 883 => 10 |
| 631 => 31 (c!) | 631 => 49 |
| 352 => 52 | 352 => 61 |
| 718 => 18 | 718 => 39 |
| 959 => 59 | 959 => 86 |
| 702 => 2 | 702 => 23 |
| 469 => 69 (c!) | 469 => 81 |
| 409 => 9 | 409 => 21 |

# Consideration about table size (2/2)

- Does the size of the hash table matter?
- Same scenario as previous slide
  - Keys are random numbers between 0 and 999
  - Distribution of keys is **non-uniform**
  - Hashing of 25 keys
- But hash table can either be of size 100 or size 97

- Conclusion
  - Table size becomes critical
  - If distribution of keys is non-uniform
  - Prime numbers exhibit good properties

| 100 | 97 |
|---|---|
| 0 => 0 | 0 => 0 |
| 25 => 25 | 25 => 25 |
| 50 => 50 | 50 => 50 |
| 75 => 75 | 75 => 75 |
| 100 => 0 (c!) | 100 => 3 |
| 125 => 25 (c!) | 125 => 28 |
| 150 => 50 (c!) | 150 => 53 |
| 175 => 75 (c!) | 175 => 78 |
| 200 => 0 (c!) | 200 => 6 |
| 225 => 25 (c!) | 225 => 31 |
| 250 => 50 (c!) | 250 => 56 |
| 275 => 75 (c!) | 275 => 81 |
| 300 => 0 (c!) | 300 => 9 |
| 325 => 25 (c!) | 325 => 34 |
| 350 => 50 (c!) | 350 => 59 |
| 375 => 75 (c!) | 375 => 84 |
| 400 => 0 (c!) | 400 => 12 |
| 425 => 25 (c!) | 425 => 37 |
| 450 => 50 (c!) | 450 => 62 |
| 475 => 75 (c!) | 475 => 87 |
| 500 => 0 (c!) | 500 => 15 |
| 525 => 25 (c!) | 525 => 40 |
| 550 => 50 (c!) | 550 => 65 |
| 575 => 75 (c!) | 575 => 90 |
| 600 => 0 (c!) | 600 => 18 |

# Hashing Strings

- Naïve approach: Sum ASCII value of each character and then reduce to table size

```python
# Hashing strings
def HashStr(key: str, M: int) -> int:
  h = np.uint32(0)
  for c in key:
    h = h + ord(c)
  return h % M
```

- Observations:
  - Average word length is 4.5. 4.5*127 = 576
  - Longest word is 45 letters long. 45*127 = 5715
- Issues:
  - Assuming a hash table of size 10,000
  - Indices up to 500 will be crowded
  - Indices above 5000 will never be used

# Hashing Strings

- <u>Experiment</u>

```
WordList = [
    # Some of the most used words
    "the", "and", "that", "have",\
    "for", "not", "with", "you", \
    # Longest word in English
    "pneumonoultramicroscopicsilicovolcanoconiosis"]

for wd in WordList:
  print("Hash({:s}) => {:d}".format(wd,HashStr(wd,10000)))
```

```
Hash(the) => 321
Hash(and) => 307
Hash(that) => 433
Hash(have) => 420
Hash(for) => 327
Hash(not) => 337
Hash(with) => 444
Hash(you) => 349
Hash(pneumonoultramicroscopicsilicovolcanoconiosis) => 4880
```

# Hashing Strings

- <u>Better approach</u>: Multiply 31 and add ASCII value of each character and then reduce to table size

```python
# Hashing strings
def HashStr(key: str, M: int) -> int:
  h = np.uint32(0)
  for c in key:
    h = (h*31) + ord(c)
  return h % M
```

```
Hash(the) => 4801
Hash(and) => 6727
Hash(that) => 8823
Hash(have) => 5240
Hash(for) => 1577
Hash(not) => 9267
Hash(with) => 9734
Hash(you) => 9839
Hash(pneumonoultramicroscopicsilicovolcanoconiosis) => 2420
```

- Horner's method
  - Consider string of length L as a polynomial
  - $h = key[0] * 31^{L-1} + key[1] * 31^{L-2} + \ ... + key[L-1] * 31^{0}$
  - Makes the distribution more uniform
  - If unsigned hash value gets too big, overflow is controlled
  - 31 is an interesting prime number (Mersenne prime (like 127 or 8191))
  - Multiplying item by 32 (by shifting <<5) and subtract item
  - Experiments shown good distribution of indices overall

# Collisions

- <u>Uniform Hashing</u>:
  - We assume that the hash function has a uniform distribution
  - Keys are mapped as evenly as possible over the index range
- <u>Collision Probability</u>:
  - Assuming an array index in the range 0..M-1
  - After how many hashed keys will there be the first collision?

- <u>Experiments</u>:
  - Example:- Output range of 97
  - 10 hashes to observe the first collision
  - Now, what is the output range was 223?
  - Need to restart a simulation.
  - Or need for better mathematical tools!

```
461 => 73
741 => 62
412 => 24
482 => 94
522 => 37
19 => 19
941 => 68
569 => 84
634 => 52
763 => 84 (c!)
```

10

# Collisions

- <u>Birthday paradox</u>:
  - In a group of $N$ random people, what is the probability that two people have the same birthdate, assuming a year consists of 365 days?
  - (pigeonhole principle) 100% if group counts 366 people
  - Surprisingly it is 50% if group consists of 23 people only
  - And 99.9% if number of persons is 70
- <u>Collision Probability</u>:
  - Probability of at least two people having same birthdate = 1 – probability of all birthdates are separate
  - Consider $M(=365)$ containers and assume $N$ balls are to be randomly placed in these containers
  - There are $M$ options for the first ball to sit
  - For the second ball, $M-1$ options for 2 distinct containers
  - For the third ball, $M-2$ options for 3 distinct containers
  - Continuing, for $N^{th}$ ball, $M-(N-1)$ options for $N$ distinct containers
  - Without considering separate containers, the number possibilities is $M^N$

# Collisions

- <u>Collision probability</u>:
  - $P(no\ collision) = \frac{M(M-1)(M-2)\ldots(M-N+1)}{M^N} = \prod_{i=0}^{N-1} \frac{M-i}{M}$
  - $P(at\ least\ one\ collision) = 1 - \prod_{i=0}^{N-1} \frac{M-i}{M}$

- Approximation
  - Remember that $\log(1+x) \approx x$, for small $x$

  - $\log \prod_{i=0}^{N-1} \frac{M-i}{M} = \sum_{i=0}^{N-1} \log\left(\frac{M-i}{M}\right) = \sum_{i=0}^{N-1} \log\left(1 - \frac{i}{M}\right) \approx \sum_{i=0}^{N-1} -\frac{i}{M} = -\frac{1}{M}\sum_{i=0}^{N-1} i$

  - $\log(P(no\ collision)) = -\frac{1}{M} . \frac{(N-1)N}{2}$

  - $P(no\ collision) = e^{-\frac{1}{M} . \frac{(N-1)N}{2}}$

  - $P(at\ least\ one\ collision) = 1 - e^{-\frac{1}{M} . \frac{(N-1)N}{2}} = 1 - e^{-const.(N^2-N)}$

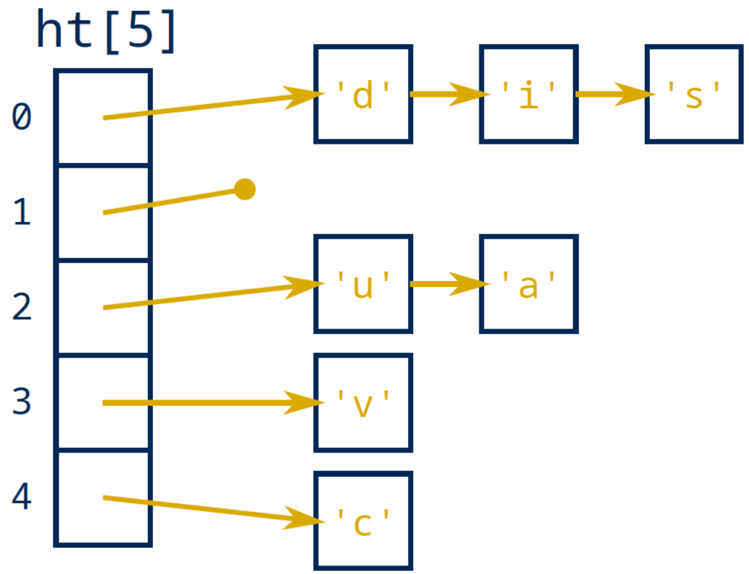
# Collisions

- <u>Collision probability</u>:
  - Number of hashed keys to first collision is $\approx \sqrt{\frac{\pi}{2}M}$
  - For an output range of 97: $\sqrt{\frac{\pi}{2}97} = 12$
  - Our experiments showed first collision at $10^{th}$ record
  - For an output range of 223: $\sqrt{\frac{\pi}{2}223} = 18$
- <u>Observation</u>:
  - Unless table size is quadratically bigger than necessary it is impossible to avoid collisions
- <u>Conclusion</u>:
  - Need to deal with collisions gracefully
  - Two options:
    - **Separate chaining**: couple the hash table with another container
    - **Open addressing**: put the key at another index in case of collision

# Separate Chaining

- <u>Principle</u>:
  - Hash table is an array of linked-lists
  - Keys are still hashed to array index in the range 0..M-1

| key | hash |
|---|---|
| u | 2 |
| c | 4 |
| d | 0 |
| a | 2 |
| v | 3 |
| i | 0 |
| s | 0 |




- **Insertion**: key pushed in the $i^{th}$ chain
- **Search**: only need to search key in the $i^{th}$ chain

# Separate Chaining

```
class HTChaining:
  # init function for our hash table
  def __init__(self, Hash, M):
    # Hash is the hash function that maps a
    # universe U to range(n)
    self.Hash = Hash
    # Create M slots, each of which has a linked list.
    # We're just going to implement as python list
    self.ht = [[] for i in range(M)]
```

- Hash table of chains, as a list of lists in python

- Python's internal hash functionality

```
print('Hash of 42 is:', hash(42))
print('Hash of -42 is:', hash(-42))
print('Hash of 42.23 is:', hash(42.23))
print('Hash of -42.23 is:', hash(-42.23))
print('Hash of IITKharagpur is:', hash('IITKharagpur'))
```

```
Hash of 42 is: 42
Hash of -42 is: -42
Hash of 42.23 is: 530343892119142442
Hash of -42.23 is: -530343892119142442
Hash of IITKharagpur is: -614617000377890825
```

# Separate Chaining

- Defining a custom hash function

```
def HashFn(key: Any, M: int) -> int:
  return ctypes.c_size_t(hash(key)).value % M
```

- Attaching it to the Hash Table object

```
# make a hash table that uses the
# custom defined HashFn
HT = HTChaining(HashFn, 10)
```

- Inserting a key

```
def insert(self, x):
  if self.find(x) is None:
    # Push key into right slot
    self.ht[self.Hash(x,len(self.ht))].append(x)
```

- Finding a key

```
def find(self,x):
  slot = self.ht[self.Hash(x,len(self.ht))]
  # take time O(n) to look for x in the slot
  for i in range(len(slot)):
    if slot[i] == x:
      return slot[i]
  return None
```

# Separate Chaining

- Deleting a key from the hash table

```
# delete an item in the hash table, if it's in there
# returns the deleted item, or None if it wasn't found.
def Remove(self,x):
  slot = self.ht[self.Hash(x,len(self.ht))]
  # take time O(n) to look for x in the bucket.
  for i in range(len(slot)):
    if slot[i] == x:
      return slot.pop(i)
  return None
```

- Helper code to print the hash table

```
def __repr__(self) -> str:
  """
  Get the string representation of this hash tabke.
  """
  # return f"Node({self.data})"
  return str([self.ht[i] for i in range(len(self.ht))])
```

# Separate Chaining

- Driver code (1/2)

```python
# make a hash table that uses the
# custom defined HashFn
HT = HTChaining(HashFn, 10)

x = 1234567
y = 890
# Insert x but not y
HT.insert(x)

# let's make sure that x is there and y isn't.
if HT.find(x) == x:
  print("Successfully found", x, "in the hash table")

if HT.find(y) is None:
  print( y, "is not in the hash table")

# Insert z and remove x
z = 1234
HT.insert(z)
removed_key = HT.Remove(x)
print("Successfully removed", removed_key)
if HT.find(x) is None:
  print( x, "is not in the hash table")
```

```
Successfully found 1234567 in the hash table
890 is not in the hash table
Successfully removed 1234567
1234567 is not in the hash table
```

# Separate Chaining

- Driver code (2/2)

```
# Uniform random number generator
distr = np.random.permutation(999)[:15]

# make a hash table that uses the
# custom defined HashFn
HT = HTChaining(HashFn, 10)
for i in distr:
  HT.insert(i)

print(HT)
```

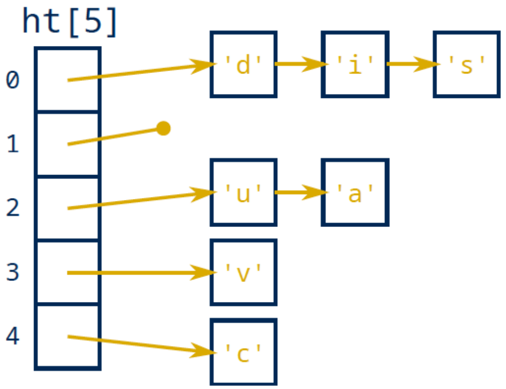
```
[[950], [], [132, 242], [], [854, 864], [], [246, 636, 216, 846], [417, 47, 567], [448, 708, 148], []]
```

# Separate Chaining - Performance analysis

- Hashing key to array index: $O(1)$
- Searching key in chain: $O(length\ of\ chain)$
- If using poor hash function, such as GalaxyHashCode()
  - One chain can end up containing all the items
  - Runtime complexity of operations: $O(N)$!
- If using hash function providing uniform distribution
  - On average, chains will be of length $^N/_M = L$
  - $L$ is also know as the load factor
  - Runtime complexity of operations: $O(L)$
  - If load factor is maintained to be constant, then complexity also becomes constant
  - Number of items $N$ in hash table cannot be controlled
  - But number of chains $M$ can be …

# Separate Chaining - Resizing

Scenario

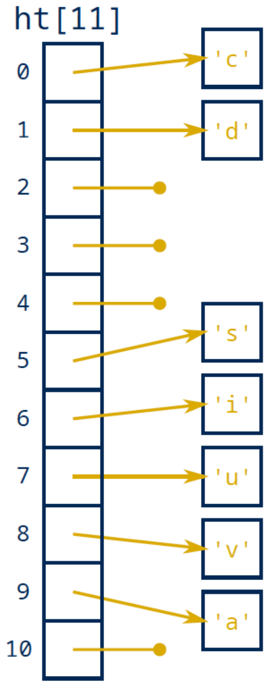



- Load factor = 7/5=1.4

Resizing - Rehashing

- Increase the size of hash table
- **Rehash** every item into new table

Result




- Load factor = 7/11=0.63

# Separate Chaining - Resizing

- Implementation

```
class HTChainingWResize:
  # init function for our hash table
  def __init__(self, Hash, M):
    # Hash is the hash function that maps a
    # universe U to range(n)
    self.Hash = Hash
    # Create M slots, each of which has a linked list.
    # We're just going to implement as python list
    self.ht = [[] for i in range(M)]
    # Number of current keys
    self.cur_size = 0

  def insert(self, x):
    if self.find(x) is None:
      # Push key into right slot
      self.ht[self.Hash(x,len(self.ht))].append(x)
      # Update current size
      self.cur_size = self.cur_size + 1
      # Resize hash table if load factor is 1
      if self.cur_size > len(self.ht):
        self.Resize(len(self.ht) * 2)
```

# Separate Chaining - Resizing

- Implementation

```
def Resize(self, capacity):
  # Back up existing hash table
  old_ht = copy.deepcopy(self.ht)
  # Create an empty table with new capacity
  self.ht = [[] for i in range(capacity)]
  self.cur_size = 0
  # Reinsert all previous items into new table
  for chain in old_ht:
    for i in chain:
      self.insert(i)
```

```
# delete an item in the hash table, if it's in there
# returns the deleted item, or None if it wasn't found.
def Remove(self,x):
  slot = self.ht[self.Hash(x,len(self.ht))]
  # take time O(n) to look for x in the bucket.
  for i in range(len(slot)):
    if slot[i] == x:
      return slot.pop(i)
      # Update current size
      self.cur_size = self.cur_size - 1
  return None
```

Complexity

- Resizing happens infrequently
- Amortized time complexity: O(1)
- Ideally, resize to another prime number

# Separate Chaining - Conclusion

Complexity

- Proportional to load factor. Typically load factor is maintained at 1.0

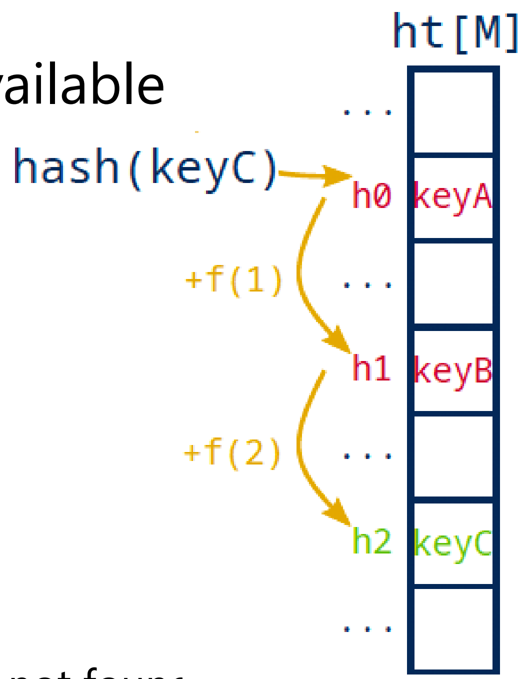
| Search | Insert | Delete |
|---|---|---|
| $O(1)$ | $O(1)$ *amortized* | $O(1)$ |

- Pros
  - Very effective if keys are uniformly distributed
- Cons
  - Rely on another data structure, such as lists, to implement the chains
    - Note that the chains could also be implemented with balanced trees!
  - Cost associated to this additional data structure
    - Memory management, traversal, etc.

# Open Addressing

Principle

- We are not going to chain collisions
- Maximum number of keys that can fit is same as the hash table size
- Collided keys are inserted at the same flat hash table, but at the next available index
- <u>Formally</u>: Key $x$ is inserted at the first index $h_i$ available
  - $h_i(x) = (hash(x) + f(i)) \bmod M$
  - where $f(0) = 0$
- Function $f$ is the collision resolution strategy
  - Linear probing, quadratic probing etc.
- Operations
  - Insert: Keep probing until an empty index is found
  - Search: Keep probing until key is found
    - Or until encountering an empty available index (i.e., key is not found)



# Linear Probing

Principle

- Function $f$ is a linear function of $i$
  - Typically $f(i) = i$
- In case of collision, try next indices sequentially

Example

- Hash table of size M = 17
- Insert sequence of characters

| Key | Hash | Index |
|---|---|---|
| I | 5 | 5 |
| < | 9 | 9 |
| 3 | 0 | 0 |
| L | 8 | 8 |
| U | 0 | 1 *(= 0 + 1)* |
| V | 1 | 2 *(= 1 + 1)* |

Observations

- Keys can form clusters of contiguous blocks
- Caused by multiple collisions
- Keys hashed into a cluster might require multiple attempts to be inserted

# Linear Probing - Implementation

```python
class HTLinear:
  # init function for our hash table
  def __init__(self, Hash, M):
    # Hash is the hash function that maps a
    # universe U to range(M)
    self.Hash = Hash

    @dataclass
    class Node:
      key: int
      taken: bool
    # Create M slots as an array of Nodes. We're
    # going to implement the array as python list
    self.ht = [Node(0, False) for i in range(M)]
    # Number of current keys
    self.cur_size = 0
```

Same hash function as for separate chaining implementation

```python
def HashFn(key: Any, M: int) -> int:
  return ctypes.c_size_t(hash(key)).value % M
```

# Linear Probing - Implementation

```python
def find(self,x):
  idx = self.Hash(x,len(self.ht))
  while (self.ht[idx].taken):
    if (self.ht[idx].key == x):
      return self.ht[idx].key
    idx = (idx + 1) % len(self.ht)
  return None
```

```python
def Resize(self, capacity):
  # Back up existing hash table
  old_ht = copy.deepcopy(self.ht)
  # Create an empty table with new capacity
  self.ht = [Node(0, False) for i in range(capacity)]
  self.cur_size = 0
  # Reinsert all previous items into new table
  for nd in old_ht:
    if nd.taken:
      self.insert(nd.key)
```

```python
def insert(self, x):
  if self.find(x) is None:
    idx = self.Hash(x,len(self.ht))
    # Find empty spot
    while (self.ht[idx].taken):
      idx = (idx + 1) % len(self.ht)
    # Push key into right spot
    self.ht[idx].key = x
    self.ht[idx].taken = True
    # Update current size
    self.cur_size = self.cur_size + 1
    # Resize hash table if half full
    if self.cur_size > len(self.ht)/2:
      self.Resize(len(self.ht) * 2)
```

Try to maintain load factor of maximum 0.5

# Deletion Principle

Scenario

- Assuming following hash table
  - Cluster of three keys
- How to remove key U?
  - Key V should still be reachable

Problems

- If free index at U, V is not reachable anymore
  - Hash(V) points to available entry
- Same issue for key U if remove key 3

Solution

- Free key's index upon removal
- But rehash every key of cluster located directly after key
  - V would be reinserted at index 1, and stay reachable

| Key | Hash | Index |
|---|---|---|
| I | 5 | 5 |
| < | 9 | 9 |
| 3 | 0 | 0 |
| L | 8 | 8 |
| U | 0 | 1 |
| V | 1 | 2 |

# Deletion Implementation

```
# delete an item in the hash table, if it's in there
# returns the deleted item, or None if it wasn't found.
def Remove(self,x):
  if self.find(x) is None:
    return None

  # Find key position and remove it
  idx = self.Hash(x,len(self.ht))
  while (self.ht[idx].key != x):
    idx = (idx + 1) % len(self.ht)
  self.ht[idx].taken = False
  ret_val = self.ht[idx].key

  # Rehash the next keys of the same cluster
  idx = (idx + 1) % len(self.ht)
  while (self.ht[idx].taken):
    # Temporarily remove key from its spot, and reinsert it right away
    temp_key = copy.deepcopy(self.ht[idx].key)
    self.ht[idx].taken = False
    # Decrease current size, as it will increased in subsequent insert
    self.cur_size = self.cur_size - 1
    self.insert(nd.key)
    idx = (idx + 1) % len(self.ht)

  # Update current size and resize hash table if 12.5% full
  self.cur_size = self.cur_size - 1
  if self.cur_size > 0 and self.cur_size < len(self.ht)/8:
    self.Resize(len(self.ht)//2)
  # Return
  return ret_val
```

# Linear Probing - Conclusion

Complexity

- Difficult complexity analysis
  - Often simplified to $O(1)$

| Search hit | Search miss/Insert |
| --- | --- |
| $O(\sim \frac{1}{2}(1+\frac{1}{1-\alpha}))$ | $O(\sim \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right))$ |

Pros

- No overhead for memory management
- Locality of reference, especially if multiple consecutive items can be in the same cache line

Cons

- Requires low load factor (0.5) compared to separate chaining
  - Bigger hash table
- Causes primary clustering

# Open Addressing Strategies

Linear probing

- $f(i) = i$
- If index at $i$ is taken, try $i + 1$ , then $i + 2$ , … , $i + k$

Quadratic probing

- $f(i) = i^2$
- If index at $i$ is taken, try $i + 1^2$ , then $i + 2^2$ , … , $i + k^2$

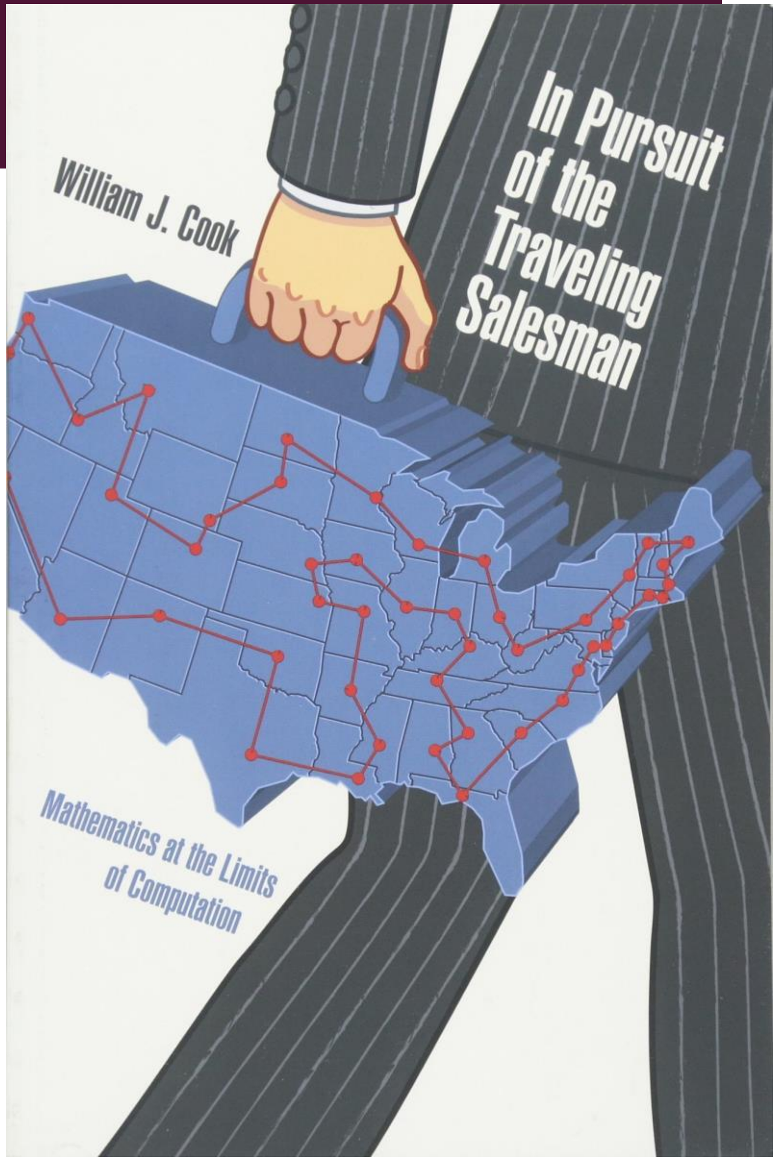
Double hashing

- Use second hash functions to compute intervals between probes
- $f(i) = i * h_2(k)$

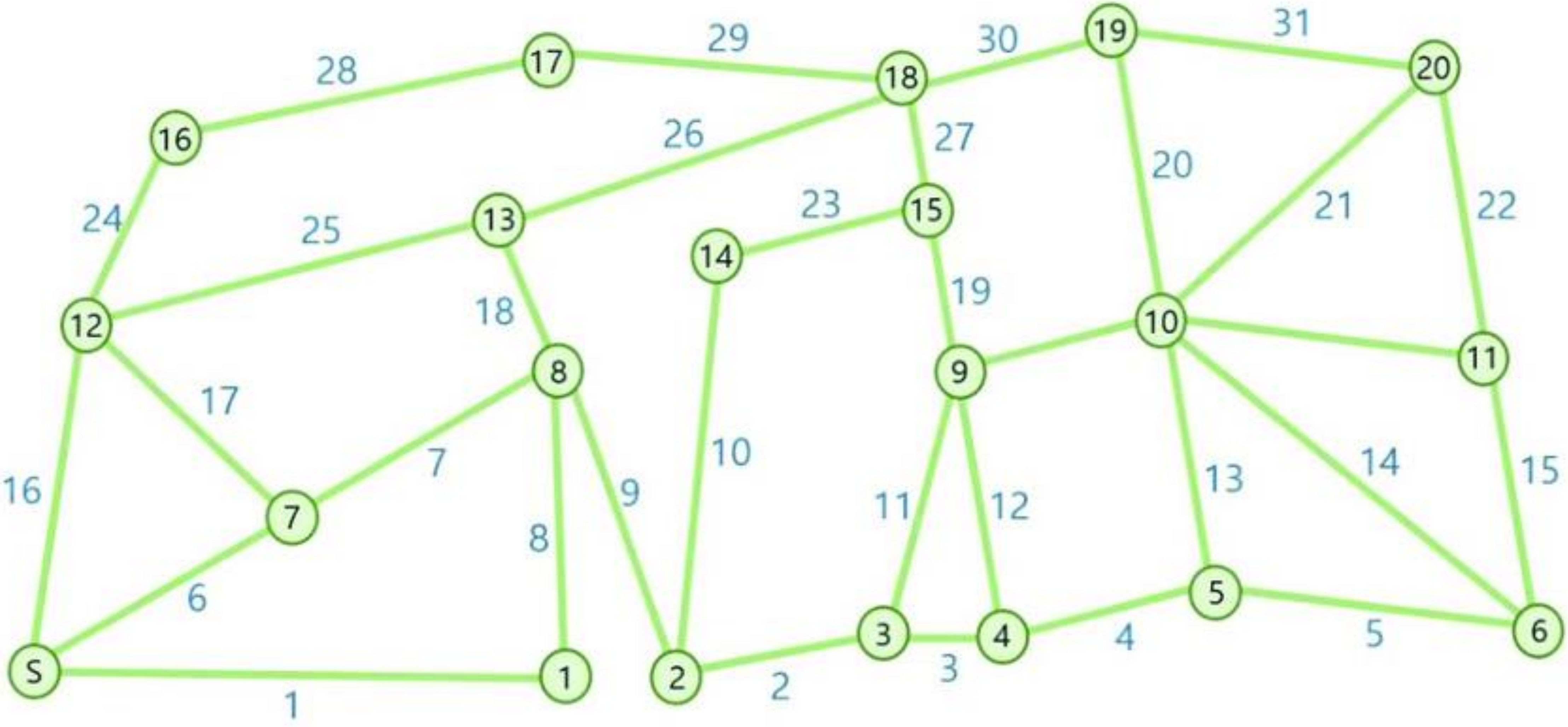
# COMPUTATIONALLY HARD PROBLEMS

# TRAVELING SALESPERSON PROBLEM

- Delivery person at the USPS! Your first job is to deliver your letters across the country.
- **Goal**: **In what order should you visit all of the cities to minimize the total distance of your trip?**
  - You'll hear this also called the "Traveling Salesman Problem" or TSP for short.
- *Problem*: There are a LOT of possible paths



# TRY IT OUT


S
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31

# TSP MORE FORMALLY

**Input:** A undirected, weighted graph G.

- Commonly assume G is complete (pair-wise connections)

**Output:** A path of minimum total cost from some start vertex that visits every node in the graph exactly once.

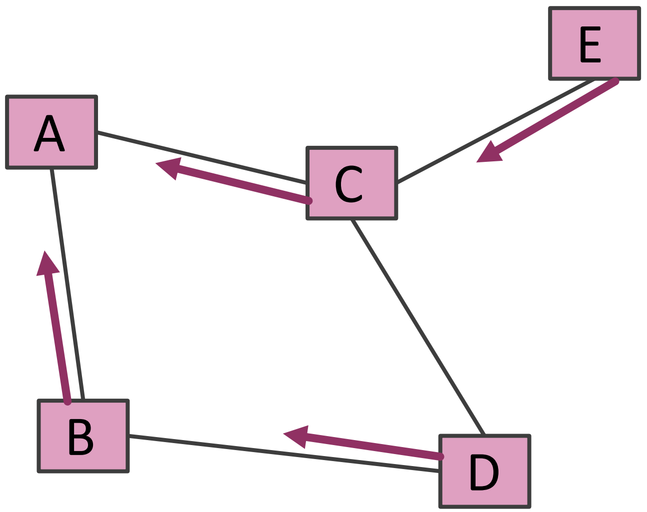
- Start vertex will be visited twice, once at beginning, and once at end
- Also called a "Hamiltonian cycle of minimum weight"

# *IDEA I* SOMETHING SOMETHING DIJKSTRA'S

Finding shortest paths in a graph? I know! Let's use Dijkstra's!

**Algorithm**:

- Pick a start vertex *v*
- Run Dijkstra's from *v* to get shortest path tree of graph
- ???
- ???




Not clear how to turn this shortest path tree into a solution! It gives an idea of where to start, but once you explore one path, you won't know how to finish the path that visits the rest of the graph.

# AN EXACT SOLUTION

The simplest algorithm is to try all possible paths

- Takes $\mathcal{O}(|V|!)$ time

So far, the only algorithms we know of to solve TSP exactly involve some kind of "choose-explore-unchoose" pattern of recursive backtracking.

- Some of the best algorithms take something like $\mathcal{O}\left(2^{|V|}\right)$

Most people think that the best we can hope for is exponential.

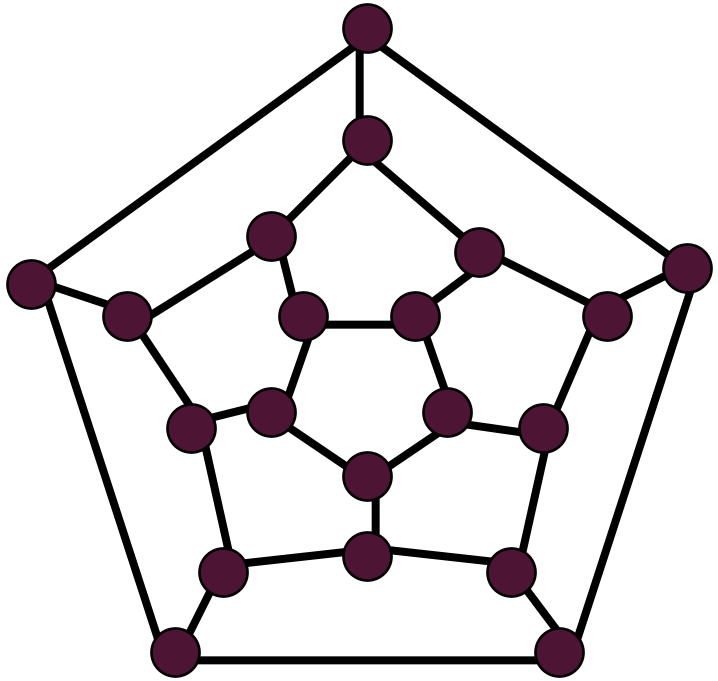
- This means this problem isn't tractable for even moderately sized inputs. With *only* 85k cities, would take something like 130 years to compute an exact solution on a fast computer (Applegate et al. (2006)).
- It has to do with this problem called "P vs. NP."

# HAMILTONIAN CYCLES

- A *Hamiltonian cycle* is a cycle that passes through each node of the graph exactly once.

# HAMILTON'S AROUND THE WORLD GAME

**In 1857, Irish mathematician William Rowan Hamilton invented a puzzle that he hoped would be very popular.**

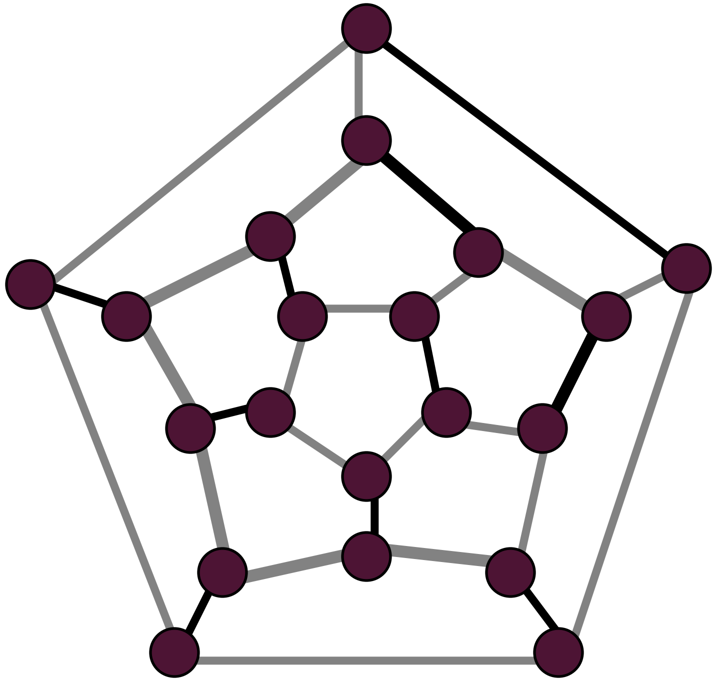


**The objective was to make what we just called a hamiltonian cycle.**

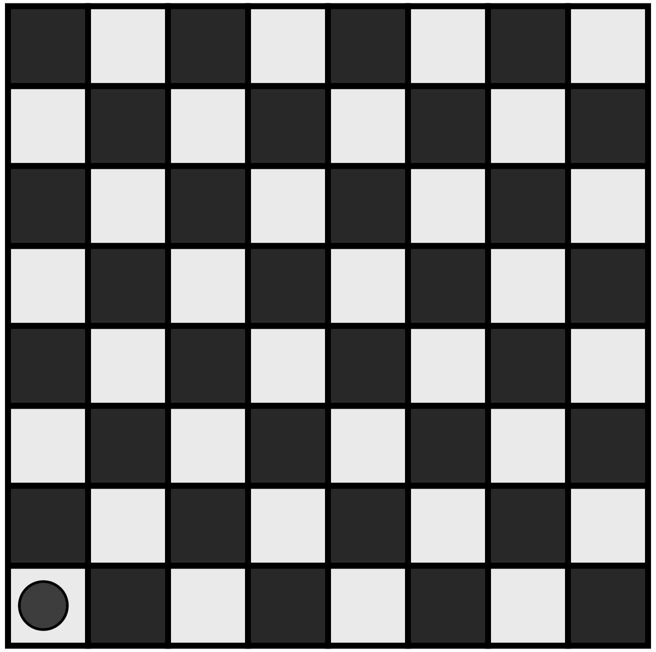
**The game was not a commercial success.**

**But the mathematics of hamiltonian cycles is very popular today.**

# HAMILTON'S AROUND THE WORLD GAME



# THE KNIGHT'S TOUR PROBLEM



**Can a knight visit all squares of a chessboard exactly once, starting at some square, and by making 63 legitimate moves?**

**The knight's tour problem is a special case of the hamiltonian tour problem.**

**The answer is yes!**

# BOOLEAN EXPRESSIONS

- Boolean, or propositional-logic expressions are built from variables and constants using the operators AND, OR, and NOT.
  - Constants are true and false, represented by 1 and 0, respectively.
  - We'll use concatenation (juxtaposition) for AND, + for OR, - for NOT, unlike the text.

# EXAMPLE: BOOLEAN EXPRESSION

- (x+y)(-x + -y) is true only when variables x and y have opposite truth values.
- Note: parentheses can be used at will, and are needed to modify the precedence order NOT (highest), AND, OR.

# THE SATISFIABILITY PROBLEM (*SAT* )

- Study of boolean functions generally is concerned with the set of *truth assignments* (assignments of 0 or 1 to each of the variables) that make the function true.
- NP-completeness needs only a simpler question (SAT): does there exist a truth assignment making the function true?

## EXAMPLE: SAT

- $(x+y)(-x + -y)$ is satisfiable.
- There are, in fact, two satisfying truth assignments:
  1. $x=0; y=1$.
  2. $x=1; y=0$.
- $x(-x)$ is not satisfiable.

# SAT AS A LANGUAGE/PROBLEM

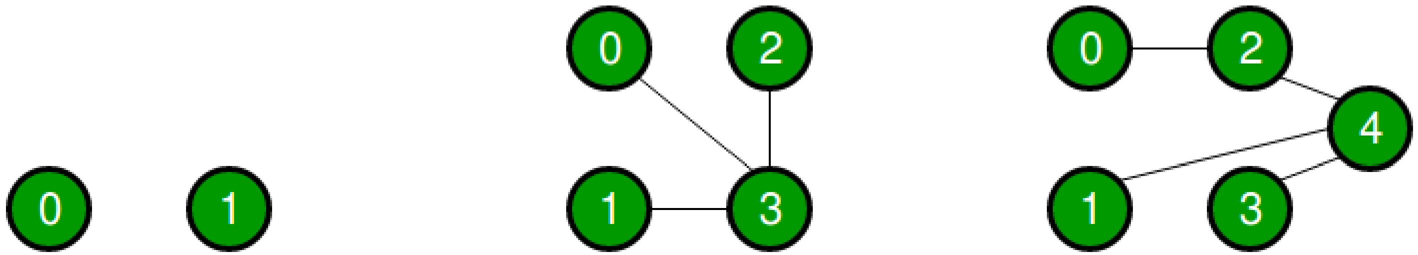
- An instance of SAT is a boolean function.
- Must be coded in a finite alphabet.
- Use special symbols (, ), +, - as themselves.
- Represent the i-th variable by symbol x followed by integer i in binary.

# FEW OTHER PROBLEMS

- **Graph Coloring**: Given a graph, assign colors to each vertex so that no two adjacent vertices have the same color, using as few colors as possible.

# FEW OTHER PROBLEMS

- **Vertex Cover**: Given a graph and a number k, the problem is to find out whether there's a set of k vertices that touch every edge in the graph.




Minimum vertex cover is empty{} | Minimum vertex cover is {3} | Minimum vertex cover is {4, 2} or {4, 0}

# FEW OTHER PROBLEMS

- **Subset Sum Problem:** Given a set of integers, is there a non-empty subset whose sum is zero or some other specific value?