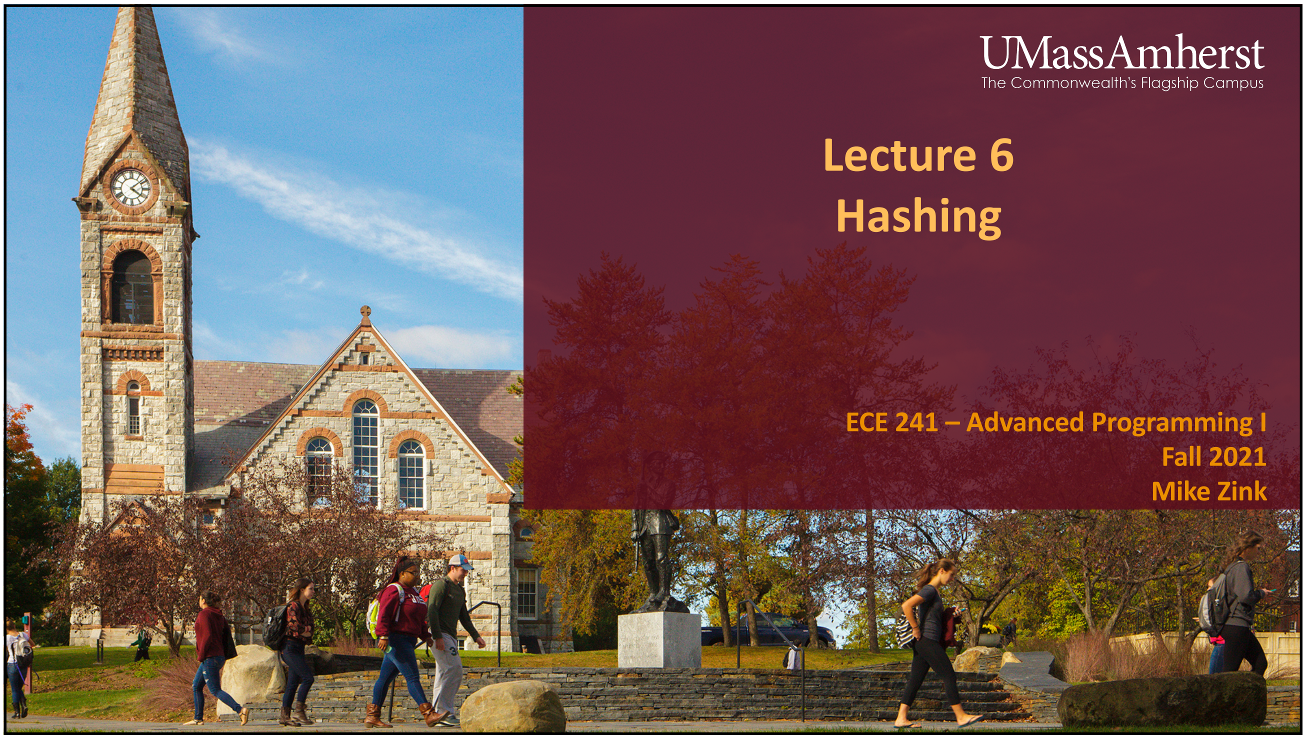


UMassAmherst
The Commonwealth's Flagship Campus

Lecture 6
Hashing

ECE 241 – Advanced Programming I
Fall 2021
Mike Zink

0

UMassAmherst

Overview

- Hash table
- Hash functions
- Collision resolution
- Map data type
- Analysis of hashing

ECE 241 – Data Structures Fall 2021 © 2021 Mike Zink 1

1

UMassAmherst

Objective

- Understand the principles of hash tables and hash functions
- Learn how to resolve collisions in hash functions
- Be able to implement hash tables and hash functions

© 2021 Mike Zink

UMassAmherst

Hashing

- Data structure that can be searched in O(1) time
- Need to know more about where items are when searched for in collection
- Single comparison if item is where it should be

© 2021 Mike Zink

UMassAmherst

Hash Table

- Collection of items stored in a way which makes them easy to find later
- Position in hash table often called **slot**
  - Holds an item
  - Named by integer value
  - Initially, every slot is empty

UMassAmherst

Hash Table

- Implement hash table using list
- Each element initialized to special Python value None
- Hash table of size *m* = 11
  - *m* slots
  - Named 0 through 10

0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

UMassAmherst

Hash Function

- Mapping between item and slot where it belongs in is called **hash function**
- Function take any item in collection and return integer in range of slot names (0, …, $m - 1$)

UMassAmherst

Hash Function: Example

- Set of integer items 54, 26, 93, 17, 77, and 31
- ”remainder method” takes item and dives it by table size => *h(item) = item%11*

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

UMassAmherst

Hash Function: Example

- After hash values computed, insert each item into hash table
- 6 of 11 slots are now occupied => **load factor** λ = *numberofitems/tablesize* (here λ = 6/11)

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

UMassAmherst

Hash Function: Example

- Use hash function to compute slot name and check if item is present
- O(1) since constant amount of time is required
  - to compute hash value
  - index hash table at that location
- => Constant time search algorithm

UMassAmherst

Hash Function: Issue

- Only works if each item maps to unique location in hash table
- If item 44 is next in collection
  - Hash value 44%11 == 0
  - Same index as for value 77
  - **Collision**

UMassAmherst

Perfect Hash Function

- Function that maps each item into a unique slot
- Perfect hash function can be constructed if items never change
- No systematic way to construct perfect hash function given arbitrary collection
- Good news: hash function does not need to be perfect

UMassAmherst

Perfect Hash Function: Approach I

- Increase size of hash table
  - Each value in the item range can be accommodated
  - Unique slot for each item
- Practical for small number of items, not feasible when number is large
- Items: 9-digit SSN => ~one billion slots

 © 2021 Mike Zink

UMassAmherst

Perfect Hash Function: Goal

- Goal:
  - Minimize collisions
  - Easy to compute
  - Evenly distributes items in hash table

 © 2021 Mike Zink

UMassAmherst

Perfect Hash Function: Folding Method

- Divide item into equal size pieces (might not work for last one)
- Add pieces together to calculate hash value
- Example:
  - Phone number: 413-545-0444 (41, 35, 45, 4, 44)
  - 41 + 35 + 45 + 4 + 44 = 169
  - 169 % 11 = 4
  - 4^{th} slot for 413-545-0444

UMassAmherst

Perfect Hash Function: Mid-Square Method

- First square item, then extract some portion of resulting digits
- Example:
  - Item 44 => 44^2 = 1,936
  - Extracting middle two digits => 93
  - 93 % 11 = 5

UMassAmherst

Perfect Hash Function: Comparison

Item	Remainder	Mid-Square
54	10	3
26	4	7
93	5	9
17	6	8
77	0	4
31	9	6

UMassAmherst

Collision Resolution

- How to place two items in hash table if they hash to same slot?
- Since avoiding collisions is impossible, collision resolution is essential

UMassAmherst

Collision Resolution: Open Addressing

- Try to find another open slot to hold item causing collision
- Start at original hash position and sequentially move through slots (loop around to start to cover entire table)
- Systematically probing each slot one at a time => **linear probing**

18

UMassAmherst

Collision Resolution: Open Addressing

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

- Insert 44 / 55 / 20
  - **Slot 0 is already occupied**
  - Slots 9, 10, 0, 1, 2 already occupied
  - **Linear probing => slot 1** also occupied, 55 in slot 2
  - Linear probing => slot3

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

19

UMassAmherst

Collision Resolution: Search

- Look up 93
  - Hash value => 5
  - Slot value => 93
- Look up 20
  - Hash value => 9
  - Slot value => 31
  - Sequential search starting at index 10
  - Find 20 or empty slot

© 2021 Mike Zink

UMassAmherst

Collision Resolution: Clustering

- If many collisions occur for same hash value, number of surrounding slots will be filled
- Negative impact when inserting other items
- Example of inserting 20 (hashing to 0)

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

© 2021 Mike Zink

UMassAmherst

Collision Resolution: Slot Skipping

- Skip slots
  - More evenly distribute items that have caused collision
  - Reduce clustering
- Example: plus 3 probing

0	1	2	3	4	5	6	7	8	9	10
77	55	None	44	26	93	17	20	None	31	54

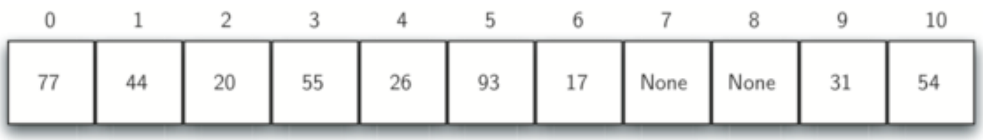
UMassAmherst

Collision Resolution: Rehashing

- Linear probing: *rehash(pos) = (pos + 1) % sizeoftable*
- Rehash "plus 3": *rehash(pos) = (pos + 3) % sizeoftable*
- General: *rehash(pos) = (pos + skip) % sizeoftable*
- Note: *skip* such that all slots in table will be used
- Often prime number is used (11 in case of example)

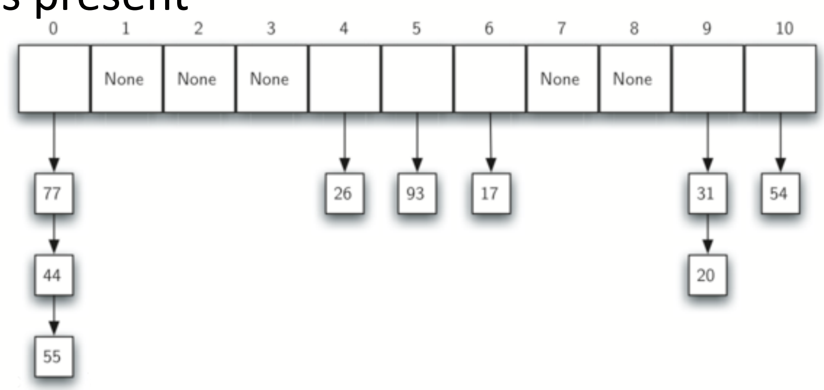
Collision Resolution: Quadratic Probing

- Rehash function that increments have value by 1, 3, 5, 7, 9
- *H, h + 1, h + 4, h + 9, h + 16*
- Quadratic probing uses skip of successive squares



Chaining

- Many items at same location
- Search: use hash function then search to decide wether item is present



UMassAmherst

Implementing Hash Table

- Dictionary => data type to store key:value pairs
- Key is used to look up associated data value
- Often referred to as **map**

 © 2021 Mike Zink

UMassAmherst

Map: Abstract Data Type

- `Map()` creates a new, empty map; returns an empty map collection.
- `put(key,val)` adds new key-value pair; if key already in map, replace old with new value
- `get(key)` returns value stored in map or `none` otherwise
- `del` delete key-value pair using statement `del map[key]`
- `len()` returns number of key-value pairs stored in map
- `in` returns `True` for statement `key in map`, `False` otherwise

 © 2021 Mike Zink

UMassAmherst

Map

- Benefit: given key look up associated data quickly
- Implementation that supports efficient search
- Hash table potentially O(1) performance

UMassAmherst

Hash Table Implementation

- Class HashTable uses two lists
  - slots holds keys
  - data holds value
  - Initial size 11 in example

```
class HashTable:
    def __init__(self):
        self.size = 11
         self.slots = [None] * self.size
         self.data = [None] * self.size
```

UMassAmherst

Hash Table Implementation

```
def put(self,key,data):
  hashvalue = self.hashfunction(key,len(self.slots))

  if self.slots[hashvalue] == None:
    self.slots[hashvalue] = key
    self.data[hashvalue] = data
  else:
    if self.slots[hashvalue] == key:
      self.data[hashvalue] = data  #replace
   else:
      nextslot = self.rehash(hashvalue,len(self.slots))
      while self.slots[nextslot] != None and \
                   self.slots[nextslot] != key:
        nextslot = self.rehash(nextslot,len(self.slots))

      if self.slots[nextslot] == None:
        self.slots[nextslot]=key
        self.data[nextslot]=data
     else:
        self.data[nextslot] = data #replace
```


UMassAmherst

Hash Table Implementation

```
def hashfunction(self,key,size):
     return key%size

def rehash(self,oldhash,size):
    return (oldhash+1)%size
```


© 2021 Mike Zink

UMassAmherst

Hash Table Implementation

```
def get(self,key):
  startslot = self.hashfunction(key,len(self.slots))

  data = None
  stop = False
  found = False
  position = startslot
  while self.slots[position] != None and  \
                     not found and not stop:
     if self.slots[position] == key:
       found = True
       data = self.data[position]
    else:
       position=self.rehash(position,len(self.slots))
       if position == startslot:
         stop = True
  return data
```


UMassAmherst

Hash Table Implementation

```
def __getitem__(self,key):
    return self.get(key)

def __setitem__(self,key,data):
    self.put(key,data)
```

- Overload `__getitem__` and `__setitem__` to allow using “[]”
- This will make index operator available

UMassAmherst

Hash Table Analysis

- Best case: O(1)
- Analyze load factor λ
  - Small λ -> lower chance of collisions
  - Large λ -> table is filling up, more collisions

UMassAmherst

Hash Table Analysis

- Open addressing with linear probing
  - Successful search $\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$
  - Unsuccessful search $\frac{1}{2}\left(1+\left(\frac{1}{1-\lambda}\right)^2\right)$
- Chaining:
  - Successful search $1+\frac{1}{\lambda}$
  - Unsuccessful search λ

UMassAmherst

Next Steps

- Next lecture on Tuesday
- Discussion on Thursday
- Homework due Thursday

UMassAmherst
The Commonwealth's Flagship Campus