Hash Tables

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I

Terminology

- Table
  - An **<u>abstract data type</u>** that stores & retrieves records according to their search key values

- Record
  - Each individual <u>row</u> in the table
  - Example:
    - A database of student records
    - So each record will have a pid, first name, last name, SSN, address, phone, email, etc.

Record Example

sid (key)	name	score
0012345	andy	81.5
0033333	betty	90
0056789	david	56.8

This is an example of a table.

...

Each individual row is a record.

9903030	tom	73
9908080	bill	49

...

Consider this problem. We want to store 1,000 student records and search them by student id.

Motivation

- Problem:
  - Given this table of records
  - We need to be able to:
    - Add new records
    - Delete records
    - Search for records

  - What's the most efficient way of doing this?

Motivation

- Problem:
  - What's the most efficient way of doing this?
  - Use an **array** to store the records, in **unsorted** order
    - Running time:
      - Adding a record:
        - O(1) since we simply add at the end of the unsorted array
      - Deleting a record:
        - Very slow, or O(n), since we have to search through the entire array to find the desired record to delete
        - We then have a "hole" in the array.
        - We can quickly fill that hole by moving the last element into it, which can happen in O(1) time.
      - Search for a record:
        - Very slow, or O(n), since we search through the entire table

Motivation

- Problem:
  - What's the most efficient way of doing this?
  - Use an **array** to store the records, in **sorted** order
    - Running time:
      - Adding a record:
        - Must insert at correct position
        - And then ALL other records, after insertion spot, must be moved
        - Very slow, or O(n)
      - Deleting a record:
        - Must find the record to delete, O(n)
        - Must fill the "hole", which means moving all other items, O(n)
      - Search for a record:
        - Binary search!
        - Fast, or O(logn)

Motivation

- **Problem:**
  - What's the most efficient way of doing this?
  - Use a binary search tree to store the records
    - Running time:
      - Adding a record:
        - Inserting into proper position in BST
        - Fast, or O(logn)
      - Deleting a record:
        - Must find correct position to delete
        - Fast, or O(logn)
      - Search for a record:
        - Also Fast, or O(logn)

Motivation

- **Problem:**
  - What's the most efficient way of doing this?
  - Use a binary search tree to store the records
    - BSTs seem to be the best solution to this
    - But there's something that is WAAAAAY faster
      - Adding, Deleting, and Searching are all **O(1)**: **CONSTANT** time
    - A very simple, naive solution that you could come up with before even taking this class
    - Just use an array! But a special type of an array.
    - Specially, use an array that is SOOOOO large that every record has its own, exclusive cell in the array
    - Often called a <u>Direct Access Table</u>

Direct Access Table

	name	score
0		
:	:	:
123456789	andy	81.5
:	:	:
334561894	betty	90
:	:	:
589224751	david	56.8
:	:	:
:	:	:
990847852	bill	49
:	:	:
999999999		

Assume we stored records based on a social security #.

One way is to store the records in a huge array

index 0..999999999

The index into array is simply an individuals SSN.

So this is VERY FAST

Adding, Deleting, and Searching: O(1)

Motivation

- Problem:
  - What's the most efficient way of doing this?
  - Use a **<u>Direct Access Table</u>**
    - So a Direct Access Table is WAAAAAY fast
    - But what is the obvious, HUUUGE problem???
    - Let's say we want to store 1000 students based on SSN
    - SSN is 9 digits
      - Assume the largest SSN is 999-99-9999
    - So we need an array that is 1 BILLION in size
    - So, yeah, this direct access table is O(1) in speed
    - But it is **<u>O(stupid)</u>** in size and memory
      - HUGE overkill to have an array of 1 billion to store 1000 records

Motivation

- **We need a better solution!**
  - We want constant add/delete/search time
  - And a **reasonably sized array**
  - What we ideally want:
    - Let's say we want to store 1000 students
    - So ideally, we only want an array of size 1000
      - So we don't waste space
    - But we still want the "direct access" that results in $O(1)$ lookup time
    - How can we do this?
      - Remembering that it was the SIZE of the array that allowed for direct access in the first place

Motivation

- **What we ideally want:**
  - This array is size 1000
    - And we will place students into this array based on their SSN.
  - So we need a way of mapping a SSN to an index
  - Example:
    - We want SSN: 527-44-7521 to somehow refer to index 368.
  - If we can do that, then we accomplish our goal

0		
:	:	:
150	842-33-5821	Andy
:	:	:
368	527-44-7521	Betty
:	:	:
527	452-85-6829	David
:	:	:
	:	:
884	651-54-3218	Bill
:	:	:
999		

Magic Address Calculator

- Solution:
  - Let's build a make-believe function:
    - the "<u>magic address calculator</u>"
    - The input to this function is the "key" (ie. SSN)
    - The function converts this SSN into an index into the reasonably sized array
    - Ideally, each SSN will "map" into its own index in the array
  - So this is still in **<u>constant time</u>**!
    - Assuming the "magic address calculator" does the conversion in constant time   …which it does!
  - And we are using a **<u>reasonably sized array</u>**!
  - This is the concept of a hash table.

Terminology

- ## **Hash table**
  - An **array of table items**, where the index is calculated by a **hash function**
  - Searching in a hash table:
    - Let's say you are searching for a record with key 4256
    - To find an item in a hash table, you do NOT follow the standard protocol of searching the entire table, record by record, comparing the key you are looking for to the key in each record.
    - Rather, we use a **hash function** on the search key to quickly calculate the index of the item
      - The hash function converts the key into the correct index into the table

Terminology

- ## **Hash function**

  - A mathematical calculation that maps the search key to an index in a hash table
    - Should be <u>fast to calculate</u>
      - Time for calculation should be $O(1)$
    - Should <u>distribute items evenly</u>

- ## **Hashing**

  - A way to access a table (array) in relatively constant (quick) time
    - Uses a hash function & collision resolution scheme

Hash Example

- **UCF System for storing student records**
  - Could store everyone's records with name, address, and telephone number using **<u>SSN</u>** as the **<u>search key</u>**
    - Could use entire SSN, but wastes too much space
      - Again, SSN's have 9 digits…that's 1 BILLION different #'s to account for
      - But UCF has only 50,000 students...so in an array of size 1 BILLION, only 50,000 spots will be used
      - **<u>EPIC WASTE!</u>**
      - On a side note, there will be no "collisions"
      - Each record will have its own, personal spot in the array based on its key (phone number)

Hash Example

- **UCF System for storing student records**
  - Could store everyone's records with name, address, and telephone number using **<u>SSN</u>** as the **<u>search key</u>**
    - Better to use last five digits of SSN number
    - For example, instead of using HashTable [589475127] to access that record, use HashTable[75127]
    - Now you need an array of size 100,000
      - Since we are using 5 digits
        - The array can go from index 0 to index 99999
    - So this is still twice the # of UCF students
    - BUT, much better than an array of size 1 BILLION

Hash Example

- **UCF System for storing student records**
  - Could store everyone's records with name, address, and telephone number using **<u>SSN</u>** as the **<u>search key</u>**
    - Better to use last five digits of SSN number
    - However, there is a chance of **<u>collisions</u>**
      - SSN # 589475127 and SSN # 428475127 have the same last five digits
      - So they will end up "mapping" to the same index in the array
        - This is called a "**<u>collision</u>**"
      - That is CLEARLY a problem.
        - Can't store two items in one index of the array
      - So, we will need to know how to handle collisions
        - Will discuss in a bit

Hash Function

- A hash function is written h(x)=i
  - **h** is the name of the hash function
  - **x** is the record search key
    - Such as the SSN in our example
  - **i** is the output of the hash function
    - which refers to an index in they array (hash table)
  - Let's say we are trying to add to a hash table
    - Once i is calculated, we can then add the record at HashTable[i]

Hash Function

- A hash function is written h(x)=i
  - In the UCF student example, h(589475127)=75127
  - So now we can take the record (name, address, phone, etc.) of the student with SSN 589475127
  - and we can store that record at HashTable[75127]
  - So this mock UCF hash function simple takes a phone number and keeps the last five digits
    - Hash functions can be as easy or as difficult as you want

Example Hash Functions

■ Three simple hash functions for integers

1. Selecting digits
2. Folding
3. Modulo arithmetic

■ Again, these are just examples!

■ Remember the goal here

■ Given some key (ie. SSN, student ID, phone #, etc)

■ We want to make an "smaller" version of that key

■ Because when a key is smaller, that means the size of the array needed can also be smaller

■ Use this new key to index the record

3 Simple Hash Functions

- Selecting digits hash function
    - Instead of using the whole integer, only select several digits
        - For example, if you have the SS#123-45-6789, just use the first 3 digits
        - h(123456789)=123
        - This is like the example we already did
    - Fast & easy to calculate, but usually does not distribute randomly
        - The first three numbers of a social security number are based on location, so people of the same state usually have the same SS#

3 Simple Hash Functions

- Folding hash function
  - Add the digits of the integer together
    - For example, if you have the SS#123-45-6789, add all the digits together
    - h(123456789)=1+2+3+4+5+6+7+8+9=45 with hash table index range $0 \leq h(\text{search key}) \leq 81$
  - Can add in different ways for hash tables of different sizes
    - h(123456789)=123+456+789=1368 with hash table index range $0 \leq h(\text{search key}) \leq 2997$

3 Simple Hash Functions

- Modulo arithmetic hash function
  - Using modulus as a hash function
    - $h(x) = x$ mod tableSize
  - Using a prime number as tableSize reduces collisions
    - For tableSize = 31,
      $h(123456789) = 123456789$ mod $31 = 2$
      with hash table index range $0 \leq h(\text{search key}) \leq 30$

Hash Functions

- Hash functions only need to be designed to operate on integers

  - Although objects such as strings can be used as a search key, they can be easily converted into an integer value

    - Then apply hash function to the integer value

Convert String to Integer

■ Ways to convert a string to an integer

1. Assign A to Z the numbers 0 to 25, and add the integers together

2. Use the ASCII or Unicode integer value for each character, and add the integers together

3. Use the binary number for the ASCII or Unicode integer value for each character, and concatenate the binary numbers together

Convert String to Integer

- Examples of converting a string to an integer
  1. "ABC" would be 0 + 1 + 2 = 3
  2. "ABC" would be 65 + 66 + 67 = 198
  3. "ABC" would be 01000001 + 01000010 + 01000011 = 010000010100001001000011 = 4,276,803

Terminology

- **Perfect hash function**
  - Ideal situation where hash function maps each search key into a different location in the hash table
    - Telephone numbers would all map to different indexes
- **Collision**
  - When a hash function maps two or more search keys into the same location in the hash table
    - h(key1) = h(key2), so have the same index value

Example Collision

- Need to store the student records of ICS 211 students based on student ID
  - Student ID has 8 digits, so need array of size 100,000,000
  - This is a waste of space, so instead use an array of size 31, with hash function h(x) = x mod 31
  - h(12345678)=h(26508090)=21 is an example of a collision
    - Both should be stored at table[21]

Brief Interlude: FAIL Picture

Daily UCF Bike Fail

Courtesy of
Unika Cole

Collision Resolution

- In case of a collision, a collision resolution scheme must be implemented
  - Assigns the search keys with the same hash function to different locations in the hash table
    - Whenever possible, items should be placed evenly in the hash table in order to avoid these collisions
  - Or we use another method called Bucket Hashing or Separate Chaining

Resolving Collisions

- Two main approaches to collision resolution
    1. Open addressing
    2. Restructure the hash table
        - Bucket Hashing
        - Separate Chaining

Open Addressing

- Open addressing
    - Probe (search) for open locations in the hash table

- Probe sequence
    - The sequence of locations that are examined for a possible open location to put the next item

Open Addressing

- Three types of probing
  1. Linear probing
  2. Quadratic probing
  3. Double hashing

Open Addressing

- Linear probing
  - In the case of a collision, keep going to the next hash table location until find an open location
    - In other words, if table[i] is occupied, check table[i+1], table[i+2], table[i+3], …
    - Need 3 states for each hash table location: empty, occupied, deleted

- Common problem
  - Items tend to cluster together in the hash table

Open Addressing

- Linear probing example
  - Table size = 31
  - Hash function = key mod 31
    - h(1234) = 25   table[25] = 1234
    - h(4055) = 25+1   table[26] = 4055
    - h(3962) = 25+2   table[27] = 3962
    - h(5853) = 25+3   table[28] = 5853
    - h(1766) = 30   table[30] = 1766
    - h(1270) = 30+1   table[0] = 1270 (wraps around)
    - All other table entries are empty

Open Addressing

- Empty, occupied, & deleted states
  - Assume we delete record #3962
  - This state must be changed to occupied (not empty), so we can still locate record #5853
    - h(1234) = 25   table[25] = 1234
    - h(4055) = 25   table[26] = 4055
    - delete(3962)   table[27] = "deleted"
    - h(5853) = 25   table[28] = 5853
    - no record added   table[29] = "empty"
    - h(1766) = 30   table[30] = 1766
    - h(1270) = 30   table[0] = 1270 (wraps around)

Open Addressing

- Quadratic probing
  - Instead of checking the next location sequentially, check the next location based on a sequence of squares
    - In other words, if table[i] is occupied, check table[i+1^2], table[i+2^2], table[i+3^2], …
    - Still have clustering (called "secondary clustering"), but this method is not as problematic as linear probing

Open Addressing

■ Quadratic probing example

   ■ Table size = 31

   ■ Hash function = key mod 31

     ■ $h(1234) = 25$ table[25] = 1234

     ■ $h(4055) = 25+1^2$ table[26] = 4055

     ■ $h(3962) = 25+2^2$ table[29] = 3962

     ■ $h(5853) = 25+3^2$ table[3] = 5853 (wraps around)

     ■ $h(1766) = 30$ table[30] = 1766

     ■ $h(1270) = 30+1^2$ table[0] = 1270 (wraps around)

     ■ All other table entries are empty

Open Addressing

■ Double hashing

■ Use two hash functions, where second hash function determines the step size to next hash table index

■ Some restrictions

■ $h_2(searchKey)$!= 0 (step size should not be zero)

■ h_2 != h_1 (avoids clustering)

Open Addressing

- Double hashing example
  - Table size = 31
  - Hash function #1 = key mod 31
  - Hash function #2 = 23 − (key mod 23)
    - h1(1234) = 25   table[25] = 1234
    - h1(4055) = 25, h2(4055) = 16 (+25),table[10] = 4055
    - h1(3962) = 25, h2(3962) = 17 (+25), table[11] = 3962
    - h1(5853) = 25, h2(5853) = 12 (+25), table[6] = 5853
    - h1(1766) = 30   table[30] = 1766
    - h1(1270) = 30, h2(1270) = 18 (+30), table[17] = 1270
    - All other table entries are empty

Open Addressing

- Double hashing example
  - h1(key) = key mod 13
  - h2(key) = 11 − (key mod 11)
    - If key = 30, probe sequence would be 4, 7, 10, 0, 3, 6, 9, 12, 2, 5, 8, 11, 1 (step 3 each time)
    - If key = 50, probe sequence would be 11, 3, 8, 0, 5, 10, 2, 7, 12, 4, 9, 1, 6 (step 5 each time)

Open Addressing

- If table size is prime, then probe sequence will visit all table locations

- With open addressing, increasing table size will reduce collisions

  - When increasing the size, the hash function needs to be reapplied to every item in the old hash table to place it in the new hash table

Restructuring the Hash Table

- How is a hash table restructured for collision resolution?

  - The structure of the hash table is changed so that the same index location can store multiple items

- Two ways to restructure a hash table for collision resolution

  1. Bucket hashing
  2. Separate chaining

Restructuring the Hash Table

■ Bucket hashing

  ■ A hash table that has an array at each location table[i], so that items of the same hash index are stored here

  ■ Choosing the size of the bucket is problematic

    ■ If too small, will have collisions

    ■ If too big, will waste space

Restructuring the Hash Table

- **Bucket hashing example**
  - Table size = 31
  - Hash function = key mod 31
    - h(1234) = 25   table[25][0] = 1234
    - h(4055) = 25   table[25][1] = 4055
    - h(3962) = 25   table[25][2] = 3962
    - h(5853) = 25   table[25][3] = 5853
    - h(1766) = 30   table[30][0] = 1766
    - h(1270) = 30   table[30][1] = 1270
  - All other table entries are empty

Restructuring the Hash Table

- Separate chaining
  - A hash table that has linked list (a chain) at each location table[i], so that items of the same hash index are stored here
  - Size of the table is dynamic
    - Less problematic than static bucket implementation

Restructuring the Hash Table

- Separate chaining example
  - Table size = 31
  - Hash function = key mod 31
    - h(1234) = 25, table[25]=>1234
    - h(4055) = 25, table[25]=>4055=>1234
    - h(3962) = 25, table[25]=>3962=>4055=>1234
    - h(5853) = 25, table[25]=>5853=>3962=>4055=>1234
    - h(1766) = 30, table[30]=>1766
    - h(1270) = 30, table[30]=>1270=>1766

Hash Tables

- Summary:
  - We use a hash table to accomplish $O(1)$ access time into a table
    - While keeping the table to a reasonable size
    - Use a hash function to map the record "keys" into an index in the hash table
    - Collisions are bound to happen and are taken care of using several possible methods
  - Comparison of Implementations (slowest to quickest)
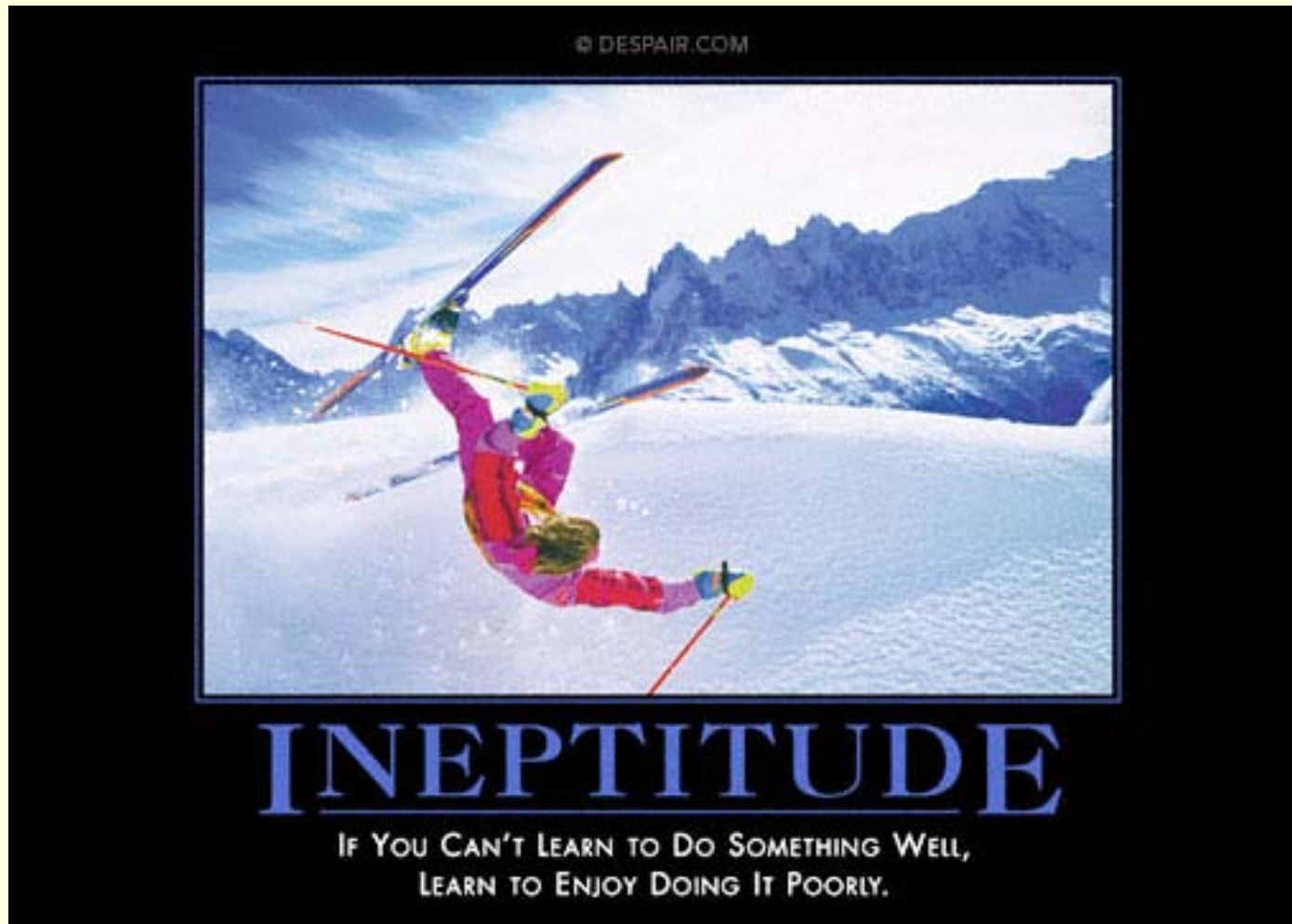   - Linear probing, quadratic probing, double hashing, separate chaining

Hash Tables

WASN'T THAT MOMENTOUS!

Daily Demotivator

Hash Tables

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I