Hash Tables

Hash functions

Open addressing
Review: hash table purpose

• We want to have rapid access to a dictionary entry based on a search key
• The key comes from an extremely large key space
• We have an array which stores a limited number of elements
  – There should be a mathematical relation between the search key and the array index in our table
  – Hash function!
A hash function is a function that maps key values to array indexes.

Hash functions are performed in two steps:
- Map the key value to an integer
- Map the integer to a legal array index

Hash functions should have the following properties:
- Fast
- Deterministic
- Uniformity
Hash function speed

• Hash functions should be fast and easy to calculate
  – Access to a hash table should be nearly instantaneous and in constant time
  – Most common hash functions require a single division on the representation of the key
  – Converting the key to a number should also be able to be performed quickly
A hash function must be *deterministic*

- For a given input it must always return the same value
  - Otherwise it will not generate the same array index
  - And the item will not be found in the hash table
- Hash functions should therefore not be determined by
  - System time
  - Memory location
  - Pseudo-random numbers
Scattering data

• A typical hash function usually results in some *collisions*
  – Where two different search keys map to the same index
  – A *perfect* hash function avoids collisions entirely
    • Each search key value maps to a different index

• The goal is to *reduce* the number and effect of collisions

• To achieve this the data should be distributed evenly over the table
Possible values

i.e. the key space

• Any set of values stored in a hash table is an instance of the universe of possible values

• The universe of possible values may be much larger than the instance we wish to store
  – There are many possible combinations of 10 letters
  – But we might want a hash table to store 1,000 names
Uniformity

• A good hash function generates each value in the output range with the same probability
  – That is, each legal hash table index has the same chance of being generated

• This property should hold for the universe of possible values and for the expected inputs
  – The expected inputs should also be scattered evenly over the hash table
A bad hash function

• A hash table is to store 1,000 numeric estimates that can range from 1 to 1,000,000
  – Hash function \( h(\text{estimate}) = \text{estimate} \mod n \)
    • Where \( n = \text{array size} = 1,000 \)

• Is the distribution of values from the universe of all possible values uniform?
  – What about the distribution of expected values?
Another bad hash function

• A hash table is to store 676 names
  – The hash function considers just the first two letters of a name
    • Each letter is given a value where a = 1, b = 2, …
    • Function = (1\text{st} \text{ letter} \times 26 + \text{value of 2\text{nd} letter}) \mod 676

• Is the distribution of values from the universe of all possible values uniform?
  – What about the distribution of expected values?
General principles

- Use the entire search key in the hash function
- If the hash function uses modulo arithmetic make the table size a prime number
- A simple and (usually) effective hash function is
  - Convert the key value to an integer, $x$
  - $h(x) = x \mod \text{tablesize}$
    - Where $\text{tablesize}$ is the first prime number larger than twice the size of the number of expected values

- But be aware that designing a good hash function is a complex subject and beyond the scope of this course!
Converting strings to integers

• In the previous examples, we had a convenient numeric key which could be easily converted to an array index
  – what about non-numeric keys (e.g. strings)?

• Strings are already numbers (in a way)
  – e.g. 7/8-bit ASCII encoding
  – "cat", 'c' = 0110 0011, 'a' = 0110 0001, 't' = 0111 0100
  – "cat" becomes 6,513,012
Strings to integers

• If each letter of a string is represented as an 8-bit number then for a length $n$ string
  
  $\text{value} = c_0 \cdot 256^{n-1} + \ldots + c_{n-2} \cdot 256^1 + c_{n-1} \cdot 256^0$

  • For large strings, this value will be very large
    
    • And may result in overflow (i.e. 64-bit integer, 9 characters will overflow)

• This expression can be factored
  
  $(\ldots (c_0 \cdot 256 + c_1) \cdot 256 + c_2) \cdot \ldots) \cdot 256 + c_{n-1}$

  • This technique is called *Horner's Method*

  • This minimizes the number of arithmetic operations

  • Overflow can then be prevented by applying the modulo operator after each expression in parentheses
Horner’s method example

• Consider the integer representation of some string, e.g. "Grom"
  – $71 \times 256^3 + 114 \times 256^2 + 111 \times 256^1 + 109 \times 256^0$
  – $= 1,191,182,336 + 7,471,104 + 28,416 + 109 = 1,198,681,965$

• Factoring this expression results in
  – $(((71 \times 256 + 114) \times 256 + 111) \times 256 + 109) = 1,198,681,965$

• Assume that this key is to be hashed to an index using the hash function $key \% 23$
  – $1,198,681,965 \% 23 = 4$
  – $((((71 \% 23) \times 256 + 114) \% 23 \times 256 + 111) \% 23 \times 256 + 109) \% 23 = 4$
Open addressing
• A collision occurs when two different keys are mapped to the same index
  – Collisions may occur even when the hash function is good
  – Inevitable due to pigeonhole principle
• There are two main ways of dealing with collisions
  – Open addressing
  – Separate chaining
Open addressing

• Idea – when an insertion results in a collision look for an empty array element
  – Start at the index to which the hash function mapped the inserted item
  – Look for a free space in the array following a particular search pattern, known as *probing*

• There are three major open addressing schemes
  – Linear probing
  – Quadratic probing
  – Double hashing
Linear probing

- The hash table is searched sequentially
  - Starting with the original hash location
  - For each time the table is probed (for a free location) add one to the index
    - Search $h(search\ key) + 1$, then $h(search\ key) + 2$, and so on until an available location is found
    - If the sequence of probes reaches the last element of the array, wrap around to $arr[0]$
- Linear probing leads to primary clustering
  - The table contains groups of consecutively occupied locations
  - These clusters tend to get larger as time goes on
    - Reducing the efficiency of the hash table
Linear probing example

- Hash table is size 23
- The hash function, \( h(x) = x \mod 23 \), where \( x \) is the search key value
- The search key values are shown in the table

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   |   |   |   |   |   | 29|   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|   |   |   |   |   |   | 32|   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|   |   |   |   |   |   |    |58|   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|   |   |   |   |   |   |    |21|   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
• Insert 81, $h = 81 \mod 23 = 12$
• Which collides with 58 so use linear probing to find a free space
• First look at 12 + 1, which is free so insert the item at index 13
Linear probing example

- Insert 35, $h = 35 \mod 23 = 12$
- Which collides with 58 so use linear probing to find a free space
- First look at $12 + 1$, which is occupied so look at $12 + 2$ and insert the item at index 14
Linear probing example

- Insert 60, $h = 60 \mod 23 = 14$
- Note that even though the key doesn’t hash to 12 it still collides with an item that did
- First look at 14 + 1, which is free
Linear probing example

- Insert 12, \( h = 12 \mod 23 = 12 \)
- The item will be inserted at index 16
- Notice that primary clustering is beginning to develop, making insertions less efficient

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   |   |   |   |   |   | 29|   | 32|   | 58 | 81 | 35 | 60 |   |   |   |   |   |   |   |   | 21 |   |   |   |
Try It!

• Insert the items into a hash table of 29 elements using linear probing:
  – 61, 19, 32, 72, 3, 76, 5, 34

• Using a hash function: $h(x) = x \mod 29$

• Using a hash function: $h(x) = (x \ast 17) \mod 29$
Readings for this lesson

• Thareja
  – Chapter 15.5.1 (Linear probing)

• Next class
  – Thareja Chapter 15.5.1 (quadratic probing, double hashing)
  – Chapter 15.5.2 (chaining)

• Midterm 2 solution posted to course website! See Piazza for document password – same as midterm 1 solution

• Please bring a pencil to class next Monday for TA evaluation forms!