# ECSE-322

16 January 2008

Lecture 6

Hashing and Searching

Arrays

Vector

1-D Array

Storage

Complexity

$O(N)$

Big O

"Time"

"Space"

90% zeroes?

Sparse Storage

# Hashing

- There is a need to store data which, for most of the domain, is zero.
- There is a requirement to minimize the space taken by the non-zero elements.
- There is a requirement to minimize the time to find a data item (if it exists) (see the previous algorithm)

$O(n)$

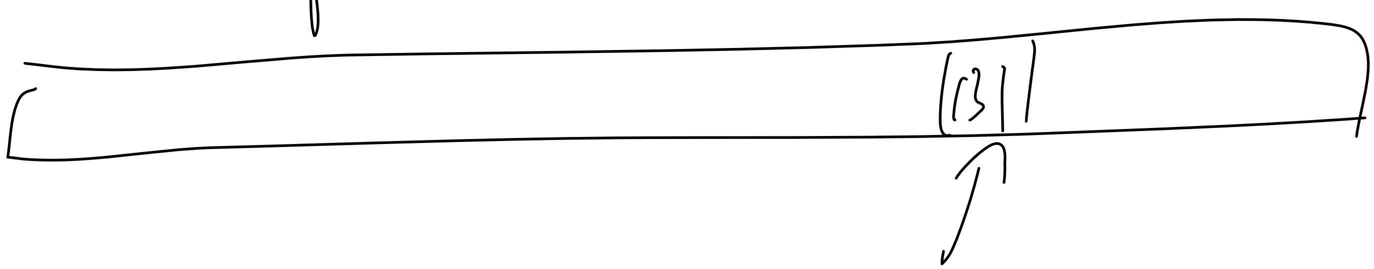
# Hashing

- With the previous structure how do you answer the question:

  *"Does the coefficient of x to the 131 exist?*

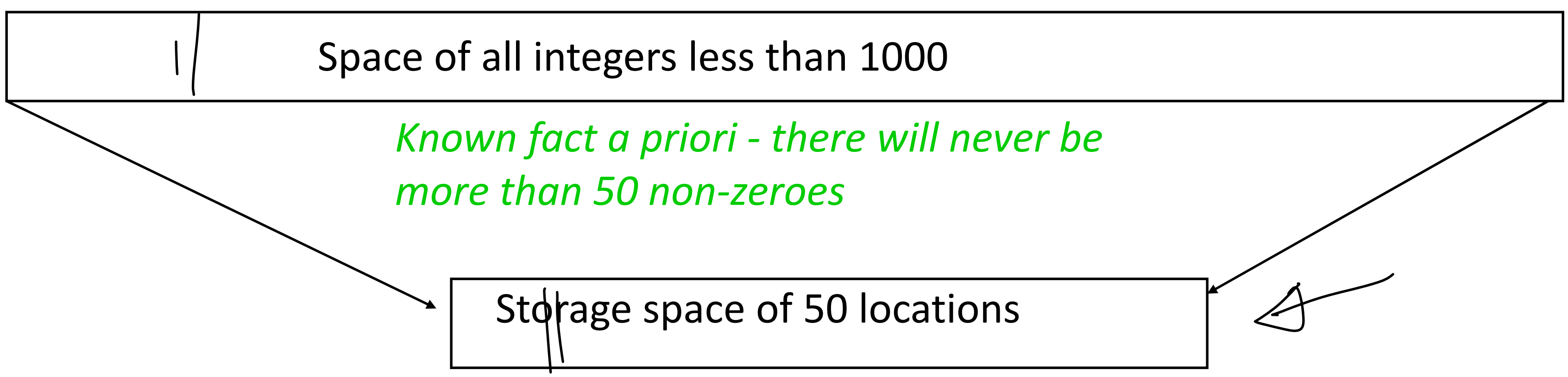
- We need a method of directly accessing the storage location for the coefficient of 131...

$$p = a_0 + a_1 x^1 \ldots$$



# Hashing

- Idea:
  - Map the data domain into a smaller space such that each location in the original space has a defined (not unique) position in the new space..



# Hashing

- Example:

$p2(x)=21x+3.5x^{499}+0.2x^{1000}$

498

$\frac{499}{499} = 1$

Apply this:

$h_0(j)=[j/499]+1$ where j is the exponent

| exponent | $h_0(j)$ |
|---|---|
| 1 | 1 |
| 499 | 2 |
| 1000 | 3 |

# Hashing

- The storage of the polynomial in an array would then be:

| | | | |
|---|---|---|---|
| a: | 21 | 1 | |
| coefficient | 3.5 | 499 | $p2(x)=21x+3.5x^{499}+0.2x^{1000}$ |
| | 0.2 | 1000 | |
| exponent | Coeff | exp | |

Row number found from $h_0(j)$

# Hashing

- How is a coefficient retrieved in this system?

131

Step 1:

compute the array index by using the function $h_0(j)$

Step 2:

compare the exponent in the array location to determine if it is the one wanted

Step 3:

extract the coefficient from the array.

$\frac{131}{499} + 1 = 1$

# Hashing

- What happens if we want the coefficient of $x^{502}$?
  - *The process fails at step 2 because the coefficient ~~6~~ doesn't exist!*

$\frac{502}{499} \times 1 \quad = 2$

# Hashing

- How does this all work?
  - Assume that we have randomly arriving data but we know there is a maximum of M items…
  - Set up an array of length M (big enough to hold all the data)
  - Assume there is a *key* associated with each element (doesn't have to be a number - in the example it is the exponent)

# Hashing

- Using the key, *e*, define a mapping function such that an index, I, in the range ~~0~~ 1 $<=I<=M$ is created…
  3
  - e.g.
  - $h_0(e) = e \bmod M + 1$

M+1

$\frac{M+1}{1}$

$1 \to M$

# Example

Consider the polynomial:

$$p2(x) = 3.576x^{131} - 0.106x^{337} + 1.03x^{858} - 5.664x^{945}$$

Store it as a linear array:

| Coefficient | location in array (key) |
|---|---|
| 0 | 1 |
| 0 | 2 |
| … | … |
| 3.576 | 131 |
| 0 | 132 |
| … | … |
| -0.106 | 337 |
| … | … |
| 1.03 | 858 |
| … | … |
| -5.664 | 945 |

Set M = 6 (size of available space)

Apply hashing function
$h_0(e) = e \bmod M + 1$

| Coeff. | Exponent |
|---|---|
| 1.030 | 858 |
| -0.106 | 337 |
| 0.000 | 0 |
| -5.664 | 945 |
| 0.000 | 0 |
| 3.576 | 131 |

# Example

M=6 gives:

| Coeff. | Exponent |
|---|---|
| 1.030 | 858 |
| -0.106 | 337 |
| ~~0.000~~ | ~~0~~ |
| -5.664 | 945 |
| ~~0.000~~ | ~~0~~ |
| 3.576 | 131 |

M=5 gives

| Coeff. | Exponent |
|---|---|
| -5.664 | 945 |
| 3.576 | 131 |
| -0.106 | 337 |
| 1.030 | 858 |
| ~~0.000~~ | ~~0~~ |

The order the elements appear in the array depends on the hashing function - *not their original order or the order in which they were received*.

# Hashing

- What could go wrong?

# Hashing

- What could go wrong?
  - More than one element could map to the same place

*a COLLISION occurs*

- How can this be solved?

# Hashing - Resolving Collisions

- If a collision occurs - use a second hashing function.
- If another collision occurs - use a third
  - Generate a family of hashing functions!

$$h_k(e), \; k = 0,1,\ldots$$

# Hashing - Resolving Collisions

- The process of storage becomes:
  - 1. Apply the hashing function
  - 2. Check if the location computed is empty
  - 3. If it is, store the data element in it
  - 4. If it is not empty, apply the next hashing function and repeat from step 2 until success or there are no hashing functions left.

# Linear Hashing

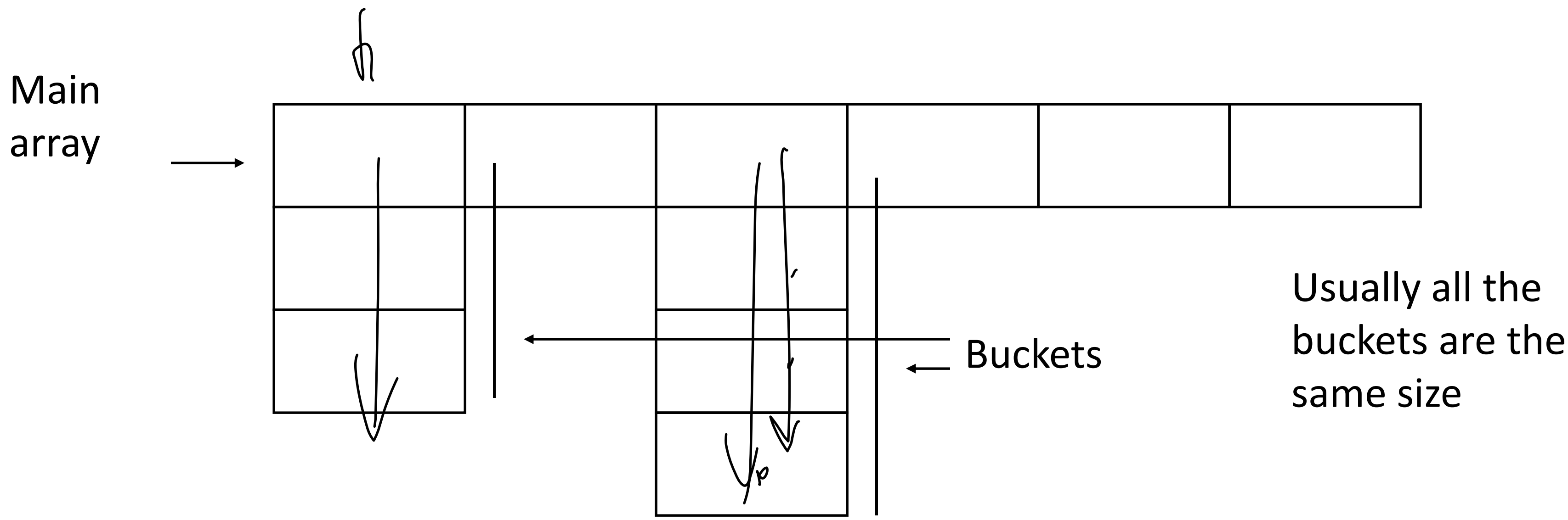
- A simple family of hashing functions

  $$h_k(e) = (h_0(e) + k) \bmod M$$

  - The implication is that, if the desired location is not empty, the next location is tried.
  - This family of functions will wrap around in the storage space.
  - Each attempt to store is known as a *probe*
    - Thus if the first function results in a collision and the second function is successful, there are 2 probes.

# Bucket Hashing

- An alternate approach to resolving collisions is to add an array at each storage location ( a *bucket* ). In a collision, the next location in the bucket is used..



# Retrieving Data

- Basically use the same process as storing data - use the hashing function..
- Problem:
  - If a collision occurred in storing data, then it may not be stored where you expect it.
  - Each retrieved item must be checked to see if it is the right one.

# Retrieving Data

- E.g.
  - If we want X and the hashing function points to $a_i$, it should not be assumed that $a_i$ contains X.
  - If a collision occurred in storing X, it will have been stored elsewhere.
  - Thus the keys of $a_i$ and X must be checked to see if they match
  - If they do not match… *Rehash with the next function.*

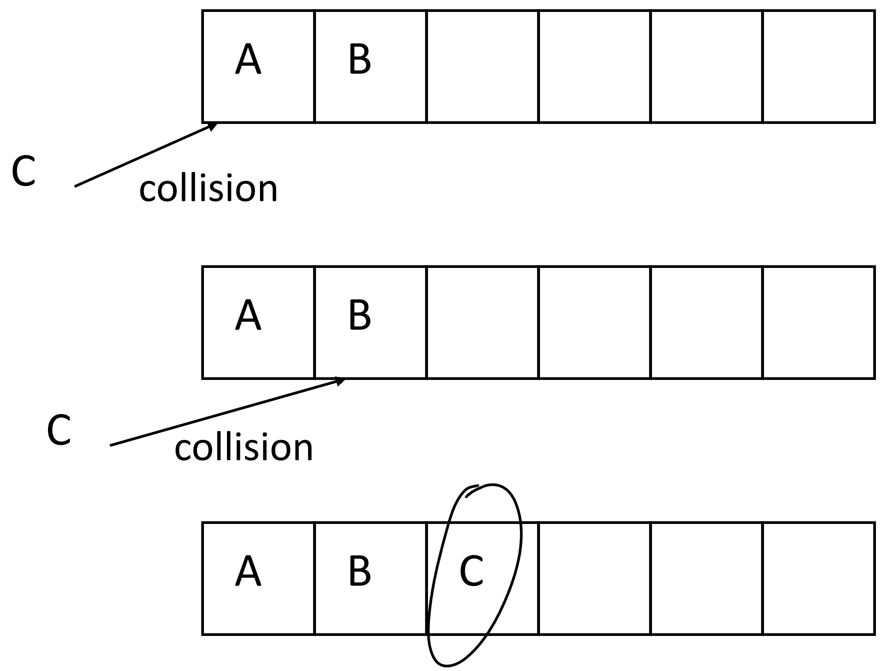
# Retrieving Data

- How do we know if X exists?
- If $a_i$ is not empty we cannot prove X does not exist until M attempts have been made.
- What happens if an item was deleted *after X was stored*?

# Retrieving Data

- Example

To store C:




C stored after 2 collisions, i.e. 3 probes

# Retrieving Data

| | | | | | |
|---|---|---|---|---|---|
| A | B | C | | | |

Now delete B..

| | | | | | |
|---|---|---|---|---|---|
| A | 0 | C | | | |

Retrieve C..

How can C ever be found?

| | | | | | |
|---|---|---|---|---|---|
| A | | C | | | |

Algorithm fails when it reaches here..

# Performance of Hashing

- Rough calculations based on bounds..
  - Place N elements in a large array M (N<<M) - assume few or no collisions
    - One fetch-hash-store cycle per item
    - O(N)
  - Place N elements in an array M (N=M) with maximum collisions
    - Average element placed after N/2 attempts
    - $O(N^2)$

# Searching

- Data is only stored so that it can be found.
- In a hashing system, data is retrieved by following the hashing function.
- What if the hashing functions are not known or the data was stored unhashed?