Outline

- Hash Table:
  - Motivation
  - Direct Access Table
  - Hash Table

- Solutions for Collision Problem:
  - Open Addressing:
    - Linear Probing
    - Quadratic Probing
    - Dynamic Table Expansion
  - Separate Chaining
Motivation

- We have to store some records and perform the following:
  - add new records
  - delete records
  - search a record by key

- Find a way to do these efficiently!
Consider this problem. We want to store 1,000 student records and search them by student id.

<table>
<thead>
<tr>
<th>pid (key)</th>
<th>name</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>0012345</td>
<td>andy</td>
<td>81.5</td>
</tr>
<tr>
<td>0033333</td>
<td>betty</td>
<td>90</td>
</tr>
<tr>
<td>0056789</td>
<td>david</td>
<td>56.8</td>
</tr>
<tr>
<td>9903030</td>
<td>tom</td>
<td>73</td>
</tr>
<tr>
<td>9908080</td>
<td>bill</td>
<td>49</td>
</tr>
</tbody>
</table>

...
Existing Data Structures

- Use an array to store the records, in unsorted order
  - add - add the records as the last entry, very fast $O(1)$
  - delete a target - slow at finding the target, fast at filling the hole (just take the last entry) $O(n)$
  - search - sequential search, slow $O(n)$

- Use an array to store the records, keeping them in sorted order
  - add - insert the record in proper position, much record movement, slow $O(n)$
  - delete a target - how to handle the hole after deletion? Much record movement, slow $O(n)$
  - search - binary search, fast $O(\log n)$
Existing Data Structures

- Binary Search Tree:
  - add: insert the record in proper position, fast $O(\log n)$
  - delete a target: fast $O(\log n)$
  - search: fast $O(\log n)$
One way is to store the records in a huge array (index 0..9999999). The index is used as the student id, i.e. the record of the student with pid 0012345 is stored at A[12345].
Direct Access Table

- **Pros:**
  - add - very fast $O(1)$
  - delete – very fast $O(1)$
  - search – very fast $O(1)$

- **Cons:**
  - Waste a lot of memory.
  - Use a table of TEN MILLION entries to store ONE THOUSAND records.
Imagine that we have such a magic function **Hash**. It maps the key (sid) of the 1000 records into the integers 0..999, **one to one**. No two different keys maps to the same number.

<table>
<thead>
<tr>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>H('0012345') = 134</td>
</tr>
<tr>
<td>H('0033333') = 67</td>
</tr>
<tr>
<td>H('0056789') = 764</td>
</tr>
<tr>
<td>H('9908080') = 3</td>
</tr>
</tbody>
</table>

**Hash Function**

function Hash(key): integer;
To store a record, we compute \( \text{Hash(pid)} \) for the record and store it at the location \( \text{Hash(pid)} \) of the array.

To search for a student, we only need to peek at the location \( \text{Hash(target sid)} \).

\[
\begin{array}{cccc}
0 & : & : & : \\
3 & 9908080 & \text{bill} & 49 \\
67 & 0033333 & \text{betty} & 90 \\
134 & 0012345 & \text{andy} & 81.5 \\
764 & 0056789 & \text{david} & 56.8 \\
999 & : & : & :
\end{array}
\]

\( \text{H('0012345')} = 134 \)
\( \text{H('0033333')} = 67 \)
\( \text{H('0056789')} = 764 \)

... 

\( \text{H('9908080')} = 3 \)
Hash Table with Perfect Hash

- Such magic function is called perfect hash
  - add – very fast $O(1)$
  - delete – very fast $O(1)$
  - search – very fast $O(1)$

- But it is generally difficult to design perfect hash. (e.g. when the potential key space is large)
# Cost Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Search</td>
<td>Insert</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>log (N)</td>
<td>(N)</td>
</tr>
<tr>
<td>Unsorted Array</td>
<td>(N)</td>
<td>(1)</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>(N)</td>
<td>(N)</td>
</tr>
<tr>
<td>Hash Table w/ Perfect Hash</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>


Issues in hashing

- Each hash should generate a unique number. If two different items produce the same hash code we have a **collision** in the data structure. Then what?

- To deal with collisions, two issues must be addressed:
  1. Hash functions must minimize collisions (there are strategies to do this).
  2. When collisions do occur, we must know how to handle them.
Collision Resolution

- Focus on issue #2 (collision resolution):
  - Assume the following hash function is a reasonably good one:
    \[ h(k) = k \% 1000 \text{ (hash code = last 3 digits)} \]
- Two ways to resolve collisions:
  - **Open Addressing**: every hash table entry contains only one key. If a new key hashes to a table entry which is filled, systematically examine other table entries until you find one empty entry to place the new key.
    - **Linear Probing**
    - **Quadratic Probing**
  - **Separate Chaining**: every hash table entry contains a pointer to a linked list of keys that hash to the same entry.
Open Addressing

- Store all keys in the hash table itself.
- Each slot contains either a key or NULL.
- To search for key $k$:
  - Compute $h(k)$ and examine slot $h(k)$. Examining a slot is known as a probe.
  - Case 1: If slot $h(k)$ contains key $k$, the search is successful.
  - Case 2: If this slot contains NULL, the search is unsuccessful.
  - Case 3: There’s a third possibility, slot $h(k)$ contains a key that is not $k$.

We compute the index of some other slot, based on $k$ and on which probe (count from 0: 0th, 1st, 2nd, etc.) we’re on. Keep probing until we either find key $k$ (successful search) or we find a slot holding NULL (unsuccessful search).
How to compute probe sequences

- **Linear probing:** Given auxiliary hash function $h$, the probe sequence starts at slot $h(k)$ and continues sequentially through the table, wrapping after slot $m - 1$ to slot 0. Given key $k$ and probe number $i$ ($0 \leq i < m$),
  
  $$h(k, i) = (h(k) + i) \mod m,$$
  
  $m$ is the size of the table.

- **Quadratic probing:** As in linear probing, the probe sequence starts at $h(k)$. Unlike linear probing, it examines cells 1, 4, 9, and so on, away from the original probe point:
  
  $$h(k, i) = (h(k) + i^2) \mod m$$
Open Addressing Example

- Three students:
  - <0000001, A, 81.3>
  - <0001001, B, 92.5>
  - <0002001, C, 99.0>

- Hash codes:
  - \( h(0000001) = 1 \mod 1000 = 1 \)
  - \( h(0001001) = 1001 \mod 1000 = 1 \)
  - \( h(0002001) = 2001 \mod 1000 = 1 \)
Linear Probing: $h(k, i) = (h(k) + i) \mod m$.

- In linear probing, collisions are resolved by sequentially scanning an array (with wraparound) until an empty cell is found.

$h(k) = 1$

<table>
<thead>
<tr>
<th>$i$</th>
<th>$h(k, i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action</th>
<th># probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store A</td>
<td>1</td>
</tr>
<tr>
<td>Store B</td>
<td>2</td>
</tr>
<tr>
<td>Store C</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>00000001</th>
<th>A</th>
<th>81.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0001001</td>
<td>B</td>
<td>92.5</td>
</tr>
<tr>
<td>2</td>
<td>0002001</td>
<td>C</td>
<td>99.0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>999</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Linear Probing Example

- Let’s say the next value to store, 352, hashed to location 3.

- We see a value is stored there, so what do we do?

- Now, if we want to search for 352?

- When do we know that we can stop searching for a value with this method?
  - When we hit an empty location.
Linear Probing Example

- Linear Probing Example (Shown on the board)
Quadratic Probing: \( h(k, i) = (h(k) + i^2) \mod m \)

- Quadratic probing eliminates the primary clustering problem of linear probing by examining certain cells away from the original probe point.

| 0 | 1  | 2  | 3  | 4  | 5  | ... | ...
|---|----|----|----|----|----|-----|-----|
| 00000001 | A  | 81.3 | 92.5 | 99.0 | ... | ...
| 0001001 | B  | \[h(k) = 1\]

<table>
<thead>
<tr>
<th>Action</th>
<th># probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store A</td>
<td>1</td>
</tr>
<tr>
<td>Store B</td>
<td>2</td>
</tr>
<tr>
<td>Store C</td>
<td>3</td>
</tr>
</tbody>
</table>
Let's say the next value to store, 352, hashed to location 3.

We see a value is stored there, so what do we do?

Now, if we want to search for 352?

When do we know that we can stop searching for a value with this method?
  - When we hit an empty location.

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td></td>
<td></td>
<td></td>
<td>173</td>
<td>281</td>
<td></td>
<td>461</td>
<td></td>
<td>352</td>
<td></td>
</tr>
</tbody>
</table>
An Issue with Quadratic Probing

- For a hash table of size \( m \), after \( m \) probes, all array elements should have been examined.
- This is true for Linear Probing, but NOT always true for Quadratic Probing (Why?)
- Insertion in Quadratic Probing: How do we know that eventually we will find a "free" location in the array, instead of looping around all filled locations?
  - if the table size is \textit{prime}, AND the table is at least \textit{half empty}, quadratic probing will always find an empty location.
Dynamic Table Expansion

- What if we don’t know how many records we'll have to store in a hash table before we set it up?

- Expand the hash table:
  1. Pick a prime number that is approximately twice as large as the current table size.
  2. Use this number to change the hash function.
  3. Rehash ALL the values already stored in the table.
  4. Now, hash the value to be stored in the table.
An array of linked lists.
Insert new items to the front of the corresponding linked list.

Key: 0001001
name: B
Score: 92.5

Key: 0000001
name: A
Score: 81.3

Key: 0002001
name: C
Score: 99.0
Separate Chaining

- Good hash function, appropriate hash size:
  - Few collisions. Add, delete, search very fast \( O(1) \)
- Otherwise...
  - some hash value has a long list of collided records
  - add - just insert at the head fast \( O(1) \)
  - delete a target - delete from unsorted linked list slow \( O(n) \)
  - search - sequential search slow \( O(n) \)
Summary

- A data structure to support (add, delete, search) operations efficiently.
- Hash table includes an array and a hash function.
- Properties of a good hash function:
  - simple and quick to calculate
  - even distribution, avoid collision as much as possible
- Collision Resolution:
  - Open Addressing:
    - Linear Probing
    - Quadratic Probing
  - Separate Chaining
The Hash Function

- Designing a good hash function
- Let’s say we were storing Strings, we want the hash function to map an arbitrary String to an integer in the range of the hash table array.

- Example:
  - $F(w) = \text{ascii value of the first character of } w$

- Why is this a poor choice?
  1) It’s designed for an array of only size 26. (Or maybe bigger if we allow non-alphabetic)
  2) More words start with certain letters than others.
The Hash Function

What if we used the following function:

\[ f(c_0c_1...c_n) = (\text{ascii}(c_0) + \text{ascii}(c_1) +...+ \text{ascii}(c_n)) \]

The problem is even if the table size is big, even 10,000, then the highest value an 8 letter string could hash to is 8*127 = 1016.

Then you would NOT use nearly 90% of the hash locations at all.

Resulting in many collisions.
The Hash Function

- Another idea in the book:
  - Each character has an ascii value less than 128. Thus, a string could be a representation of a number in base 128.
  - For example the string “dog” would hash to:
    - \[ \text{ascii}(\text{d}) \times 128^0 + \text{ascii}(\text{o}) \times 128^1 + \text{ascii}(\text{g}) \times 128^2 = \]
    - \[ 100 \times 1 + 111 \times 128 + 103 \times 128^2 = 1701860 \]

- What are the problems with this technique?
  1) small strings map to HUGE integers
  2) Just computing this function may cause an overflow.
The Hash Function

- How can we deal with these problems?
  - Using the mod operator
    
    \[
    f(c_0c_1...c_n) = (\text{ascii}(c_0)*128^0 + \text{ascii}(c_1)*128^1 +...+ \text{ascii}(c_n)*128^n) \mod \text{tablesize}
    \]
  
  - We can still get an overflow if we mod at the end.
  - So we can use Horner’s rule:
    
    ➢ Specifies how to evaluate a polynomial without calculating \(x^n\) in that polynomial directly:
      
      \[
      c_nx^n + c_{n-1}x^{n-1} + ... c_1x + c_0 =c_0 + x(c_1 + x(c_2 + ... + x(c_{n-1} + xc_n)...))
      \]
    
    \[
    (\text{ascii}(c_0)*128^0 + \text{ascii}(c_1)*128^1 +...+ \text{ascii}(c_n)*128^n) = \\
    \text{ascii}(c_0) + 128(\text{ascii}(c_1) + 128(\text{ascii}(c_2) + ...+(128(\text{ascii}(c_{n-1}) + 128 \text{ascii}(c_n)))...))
    \]
(ascii(c₀) * 128⁰ + ascii(c₁) * 128¹ + ... + ascii(cₙ) * 128ⁿ) =

ascii(c₀) + 128(ascii(c₁) + 128(ascii(c₂) + ... + (128(ascii(cₙ₋₁) + 128 ascii(cₙ))...)))