Hashing - Basic Idea

- A mapping between the search keys and indices - efficient searching into an array
- Each element is found with one operation only

Hashing Example

Example:
- 1000 students
  - Identification number between 0 and 999
  - Use an array of 1000 elements.
- SSN of each student a 9-digit number
  - Much more elements than the number of the students - a great waste of space
The Approach

• Directly referencing records in a table using arithmetic operations on keys to map them onto table addresses

• Hash function: Function that transforms the search key into a table address

Direct-Address Tables

• The most elementary form of hashing

• Assumption: Direct one-to-one correspondence between the keys and numbers 0, 1, ..., m-1 (m - not very large)

• Array $A[m]$: Each position (slot) in the array contains a pointer to a record, or NULL

• Cost: The size of the array we need is determined by the largest key. Not very useful if there are only a few keys (i.e., the array is sparse)

Hash Functions

• Transform the keys into numbers within a predetermined interval to be used as indices in an array (table, hash table) to store the records

Hash Functions - Numerical Keys

• Keys - numbers

• If $M$ is the size of the array, then $h(\text{key}) = \text{key} \% M$

• This will map all the keys into numbers within the interval $[0 .. (M-1)]$
Hash Functions - Character Keys

- Keys - strings of characters
- Treat the binary representation of a key as a number, and then apply the hash function

How Keys are Treated as Numbers

- If each character is represented with $m$ bits, then the string can be treated as base-$m$ number

Example

A K E Y:
00001 01011 00101 11001 =

$1 \cdot 32^3 + 11 \cdot 32^2 + 5 \cdot 32^1 + 25 \cdot 32^0 = 44271$

Each letter is represented by its position in the alphabet. E.g., K is the 11-th letter, and its representation is 01011 (11 in decimal)

Long Keys

- If the keys are very long, an overflow may occur
- A solution to this problem is to apply the Horner's method in computing the hash function
Horner's Method

\[ a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a_1 x + a_0 x^0 = \]
\[ x (x (\ldots (x (a_n x + a_{n-1}) + a_{n-2}) + \ldots) + a_1) + a_0 \]

\[ 4x^5 + 2x^4 + 3x^3 + x^2 + 7x + 9 = \]
\[ x (x (x (4 x + 2) + 3) + 1) + 7) + 9 \]

The polynomial can be computed by alternating the multiplication and addition operations

Example

\[ \text{VERY LONG KEY} \]
\[ \begin{array}{cccccccccccc}
  0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1
\end{array} \]
\[ \begin{array}{cccccccccccc}
  0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1
\end{array} \]
\[ \begin{array}{cccccccccccc}
  0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1
\end{array} \]
\[ \begin{array}{cccccccccccc}
  0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1
\end{array} \]

\[ \begin{array}{cccccccccccc}
  22 & 5 & 18 & 25 & 12 & 15 & 14 & 7 & 11 & 5 & 25
\end{array} \]

\[ \begin{array}{cccccccccccc}
  22 & 32^{10} + 5 & 32^5 + 18 & 32^8 + 25 & 32^7 + 12 & 32^6 + 15 & 32^5 + 7 & 32^4 + 11 & 32^2 + 5 & 32^1 + 25 & 32^0
\end{array} \]

\[ ((((((((22.32 + 5)32 + 18)32 + 25)32 + 12)32 + 15)32 + 14)32 + 7)32 + 11)32 + 5)32 + 25 \]

Compute the hash function by applying the mod operation at each step, thus avoiding overflowing

\[ h_0 = (22.32 + 5) \mod M \]
\[ h_1 = (32.h_0 + 18) \mod M \]
\[ h_2 = (32.h_1 + 25) \mod M \]

\[ \ldots \]

Code

```c
int hash32(char[] name, int tbl_size){
    key_length = name.length;
    int h = 0;
    for (int i=0; i<key_length; i++)
        h = (32*h+name[i]) % tbl_size;
    return h;
}
```
Hash Tables

• **Index:** Integer generated by a hash function between 0 and M-1

• Initially, blank slots
  - Sentinel value, or a special field in each slot

Size of the Table

• **Table size** M: Different from the number of records N

• **Load factor:** \( \lambda = \frac{N}{M} \)

• M must be prime to ensure even distribution of keys

**COLLISION RESOLUTION:**

SEPARATE CHAINING
Collision Resolution

- Collision Resolution
  - Separate Chaining
  - Open Addressing
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
  - Rehashing
- Extendible Hashing

Hash Tables - Collision

- Problem: Many-to-one mapping
  - A potentially huge set of strings → a small set of integers

- Collision: Having a second key into a previously used slot

- Collision resolution: Deals with keys that are mapped to same slots

Separate Chaining

- Invented by H. P. Luhn, an IBM engineer, in January 1953

- Idea: Keys hashing to same slot are kept in linked lists attached to that slot

- Useful for highly dynamic situations, where the number of the search keys cannot be predicted in advance

Example

Key: A S E A R C H I N G E X A M P L E
Hash: 1 8 5 1 7 3 8 9 3 7 5 2 1 2 5 1 5
(M = 11)

Separate chaining:

0 1 2 3 4 5 6 7 8 9 10
= L M N = E = G H I =
  A X C P R S =
  A = = E = =
  A = E
  = =
Separate Chaining - Length of Lists
- \( N \) - number of keys
- \( M \) - size of table
- \( N/M \) - average length of the lists

Separate Chaining - Search
- Unsuccessful searches go to the end of some list
- Successful searches are expected to go half the way down some list

Separate Chaining - Choosing Table Size \( M \)
- Relatively small so as not to use up a large area of contiguous memory
- But large enough so that the lists are short for more efficient sequential search

Separate Chaining - Other Chaining Options
- Keep the lists ordered - useful if there are much more searches than inserts, and if most of the searches are unsuccessful
- Represent the chains as binary search tree. Extra effort needed - not efficient
Separate Chaining – Advantages and Disadvantages

- **Advantages**
  - Used when memory space is a concern
  - Easily implemented
- **Disadvantages**
  - Unevenly distributed keys – long lists: Search time increases, many empty spaces in the table

**Collision Resolution**

- Collision Resolution
  - Separate Chaining
  - Open Addressing
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
  - Rehashing
- Extendible Hashing

**Open Addressing**

- Invented by A. P. Ershov and W. W. Peterson in 1957, independently
  - **Idea**: Store collisions in the hash table
  - **Table size**: Must be at least twice the number of the records

**Collison Resolution: Open Addressing Extendible Hashing**
Open Addressing
If collision occurs, next probes are performed following the formula:

\[ h_i(x) = (\text{hash}(x) + f(i)) \mod \text{Table\_Size} \]

where:
- \( h_i(x) \) is an index in the table to insert \( x \)
- \( \text{hash}(x) \) is the hash function
- \( f(i) \) is the collision resolution function.
- \( i \) is the current attempt to insert an element.

Open Addressing
- **Problems with delete:** A special flag is needed to distinguish deleted from empty positions.
  - Necessary for the search function - if we come to a "deleted" position, the search has to continue as the deletion might have been done after the insertion of the sought key.
  - The sought key might be further in the table.

Linear Probing  \[ f(i) = i \]

**Insert:** If collision - probe the next slot
  - If unoccupied - store the key there
  - If occupied - continue probing next slot

**Search:**
- a) Match - successful search
- b) Empty position - unsuccessful search
- c) Occupied and no match - continue probing

If end of the table - continue from the beginning.

---

**Example**

<table>
<thead>
<tr>
<th>Key:</th>
<th>A S E A R C H I N G E X A M P L E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash:</td>
<td>1 0 5 1 18 3 8 9 14 7 5 5 1 13 16 12 5</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18</th>
</tr>
</thead>
<tbody>
<tr>
<td>S A E R C G H I N A E X L M P E L E</td>
</tr>
<tr>
<td>* A * C G H I N * E * * * * X</td>
</tr>
<tr>
<td>* * * A * * * * L M P * * * * E</td>
</tr>
<tr>
<td>* - unsuccessful attempts</td>
</tr>
</tbody>
</table>
```
Linear Probing

- Disadvantage: “Primary clustering”
- Large clusters tend to build up
- Expected number of probes:
  - For insertions and unsuccessful searches:
    \[ \frac{1}{2}(1+1/(1- \lambda)^2) \]
  - For successful searches:
    \[ \frac{1}{2}(1+1/(1- \lambda)) \]

Quadratic Probing

- Use a quadratic function to compute the next index in the table to be probed
- The idea here is to skip regions in the table with possible clusters

Quadratic Probing

- In linear probing, we check the \( i^{th} \) position. If it is occupied, we check the \( i + 1^{st} \) position, next \( i + 2^{nd} \), etc.

- In quadratic probing, if the \( i^{th} \) position is occupied, we check the \( i+1^{st} \), next we check the \( i+4^{th} \), next \( i+9^{th} \), etc.

Double Hashing

- Purpose: To overcome the disadvantage of clustering
- A second hash function to get a fixed increment for the “probe” sequence
  - \( \text{Hash}_2 \) should never evaluate to 0
  
    \[ \text{Hash}_2(x) = R - (x \mod R) \]
  
    \( R \): Prime, smaller than table size
Rehashing

- Table size: \( M > N \)
- For small load factor, the performance is much better than for \( \lambda = N/M \) close to one
- Best choice: \( \lambda = 0.5 \)
- When \( \lambda > 0.75 \) - rehashing

Rehashing

- Build a second table twice as large as the original and rehash there all the keys of the original table
- Expensive operation
  - Running time \( O(N) \)
  - However, once done, the new hash table will have good performance

Extendible Hashing

- External storage
  - \( N \) records in total to store
  - \( M \) records in one disk block
- No more than two blocks are examined

Extendible Hashing

Idea:
- Keys are grouped according to the first \( m \) bits in their code.
- Each group is stored in one disk block.
- If some block becomes full, each group is split into two, and \( m+1 \) bits are considered to determine the location of a record
Example

- 4 disk blocks, each can contain 3 records
- 4 groups of keys according to the first two bits

<table>
<thead>
<tr>
<th>Directory</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010</td>
<td>01001</td>
<td>10001</td>
<td>11000</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>01010</td>
<td>10100</td>
<td>11010</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example (cont’d)

- New key to be inserted: 01011
- Block2 is full, so we start considering 3 bits

<table>
<thead>
<tr>
<th>Directory</th>
<th>000/001 (still on same block)</th>
<th>010</th>
<th>011</th>
<th>100/101</th>
<th>110/111</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010</td>
<td>01001 01100</td>
<td>10001</td>
<td>11000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>01010</td>
<td>----</td>
<td>11010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>01011</td>
<td>10100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Extendible Hashing

Size of the directory: \(2^D\)

\(2^D = O(N^{(1+1/M)} / M)\)

- \(D\) - the number of bits considered
- \(N\) - number of records
- \(M\) - number of disk blocks

Conclusion 1

- Hashing is a search method used when
  - sorting is not needed
  - access time is the primary concern
**Conclusion 2**

- **Time-space trade-off:**
  - No memory limitations - Use the key as a memory address (minimum amount of time)
  - No time limitations - Use sequential search (minimum amount of memory)

- **Hashing:** Gives a balance between these two extremes - a way to use a reasonable amount of both memory and time

**Conclusion 3**

- To choose a good hash function is an art
- The choice depends on the nature of keys and the distribution of the numbers corresponding to the keys

**Conclusion 4**

- Best course of action:
  - Separate chaining: If the number of records is not known in advance
  - Open addressing: If the number of the records can be predicted and there is enough memory available