Hashing

Professor Yeali S. Sun
National Taiwan University
Introduction

- Even with $O(N \cdot \log_2 N)$ complexity, when $N$ is very large, the number of steps to retrieve/update/insert/delete data may be still large.
  - Example: for a balanced binary search tree $O(N \cdot \log_2 N)$ complexity.
    - e.g, $N=10,000$, worst case is 13
Introduction (cont’d)

• Some applications desire methods similar to “table operations that requires no search,” i.e. indexing
  – To locate (and insert/delete/update) a data item virtually instantaneously
    - O(1)

```
<table>
<thead>
<tr>
<th>Search Key</th>
<th>Address Calculator</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>N-1</td>
<td>...</td>
</tr>
</tbody>
</table>
```

Array T
Applications

- **Time is vital to applications like**
  - 911 call – phone number -> caller’s address
  - Routing table - **Destination IP** – next hop to go
  - Firewall – ip address, port number, **URL-based dispatching**
  - QoS (Quality of Service) – for mission-critical traffic, real-time multimedia applications, VoIP, etc.
  - Accounting/billing
  - Commercial database systems
Hashing

• **Hash function** - address calculator
  – has an arbitrary integer as an argument
  – maps an integer into an array (hash table) index $y = h(x)$
  – If not an integer, one can map the key into an integer
    • E.g., convert a string into an integer

• **Hash table**
  – where the data items are stored
Hashing (cont’d)

• Identifier space and identifiers in use
• Perfect Hash
  – A perfect hash function maps each search key into a unique location of the hash table
  – A perfect hash function is possible if you know all the search keys.

• Typically, collision occurs.
  – One or more items map to the same location
• Collision-resolution schemes
  – “Overflow” handling
Hash Functions - Requirements

• Be easy and fast to compute
• Place items evenly throughout the hash table
Hash Functions:
Examples

1. Selecting digits
2. Folding
3. Modulo arithmetic
Method#1: Selecting Digits

• Assume search keys (identifiers) are made of \( N \) digits
• Select *some* digits to form the index to the hash table
• Example
  – Nine-digit employee ID number 001364825
  – Select 4\(^{th}\) and 9\(^{th}\) digits
  – \( H(001364825)=35 \)
  – \( T[35] \)
Method#1: Selecting Digits (cont’d)

Advantage
• Easy and fast

Disadvantages
• Generally they do NOT evenly distribute the items in the hash table
  • One must be careful about the digits selected to avoid skewed distribution
• Example: social security number in USA
  – The first three digits are based on the geographic region.
  – If the first three digits are used, map all people from the same state into the same location of the hash table.
Method#1: Selecting Digits - Digit Analysis

1. Assume *all* the identifiers in the table are *known in advance*

2. Each identifier $x$ is interpreted as a number using some radix $r$

3. Examine the digits of each identifier

4. *Delete digits having the most skewed distributions*

5. The number of remaining digits is small enough to give an address in the range of the hash table
Method#2: Folding

• **Add all digits to obtain the index**

• Example – 001364825 (nine digits)
  – 0+0+1+3+6+4+8+2+5=29
  – T[29]
  – 0<=h(search key)<=81

• **Heavy collision**

• **Form groups - to increase the size of the hash table**
  – 001 + 364 + 825 = 1,190
  – T[1190]
  – 0<=h(search key)<=3*999=2,997
Method#3: Modulo Arithmetic

• A simple and effective way of hashing.
  \[ h(x) = x \mod \text{tableSize} \]

• Example:
  - \( \text{tableSize}=101 \)
  - \( H(x)=y \): It maps any integer \( x \) into the range from 0 to 100.
Converting a char string to an integer

• Search key is a character string, such as a name.

• Approach #1
  - Assign each char an integer value, e.g.,
    • The ASCII values N=78, O=79, etc.
    • Or A=0, .., Z=25.
  - Add these numbers -> get another integer

• It will not be unique to the character string.
Converting a char string to an integer (cont’d)

• Approach #2
  - Write the numeric value for each char in binary
  - Concatenate the results
  - Example,
    • N=14, or 01110
    • O=15, or 01111
    • T=20, 10100
    • E=5, 00101
    • “NOTE” = 01110 01111 10100 00101 (474,757
  - \( h(x) = x \mod \text{tableSize} \)
Converting a char string to an integer (cont’d)

- **Approach #3**
  - Write the numeric value for each char in binary
  - Concatenate the results
  - Evaluate the expression
  - Example,
    - N=14, or $01110_2$
    - O=15, or $01111_2$
    - T=20, $10100_2$
    - E=5, $00101_2$
    - NOTE = 01110 01111 10100 00101 ($474,757_{10}$)

\[
14 \cdot 32^3 + 15 \cdot 32^2 + 20 \cdot 32^1 + 5 \cdot 32^0
\]

- Each char is a 5-bit binary number and $2^5=32$
Converting a char string to an integer (cont’d)

• Horner’s rule – to make the computation fast
  – Factor the expression to minimize the number of arithmetic operations.

• We have

\[ ((14 \cdot 32 + 15) \cdot 32 + 20) \cdot 32 + 5 \]

\[ 14 \cdot 32^3 + 15 \cdot 32^2 + 20 \cdot 32^1 + 5 \cdot 32^0 \]
Collision Resolution –

• Open addressing
• Restructuring the hash table
Collision Resolution – Open Addressing

• Method
  – To insert a new item into the table
  – If the hash function indicates a location in the hash table already occupied
  – One “probes” for some other empty or open location to place the item.

• Three schemes
  - Linear probing
  - Quadratic probing
  - Double hashing
Linear Probing

• **Method**
  - Begin at the hash location
  - **Search the table sequentially**

• **Primary issue - clustering**
Linear Probing

• Drawbacks
  – Tends to create clusters of search keys (records)
  – As more search keys are entered, clusters tend to merge, leading to bigger clusters.
  – Result in inefficient search \( O(1)+O(N)=O(N) \)
Quadratic Probing – to disperse collision cluster

- Method
  - Begin at the hash location
  - Probing the table in the sequence of
    Table[h(Skey)],
    Table[h(Skey)+1^2],
    Table[h(Skey)+2^2],
    Table[h(Skey)+3^2], ...
- Still, the **same probe sequence** for data items mapping to the same location
Double Hashing

• Method
  – Primary hash
  – Secondary hash – determine the size of the steps taken

• Example
  – Double hashing during the insertion of 58, 14 and 91
    – $h_1(key) = key \mod 11$
    – $h_2(key) = 7 - (key \mod 7)$
    – $h_1(58)=3$, $h_2(58)=5$ - probe sequence is
      – 3,
      – 8 (3+5),
      – 2 ((8+5) mod table size, wrap around),
      – 7 (2+5),
      – 1 (7+5 mod table size) (wrap around),
      – 6, 0, 5, 10, 4, 9…

Hash table (size 11)
Double Hashing (cont’d)

- $h_1(key) = key \mod 11$
- $h_2(key) = 7 - (key \mod 7)$
- $h_1(14)=3$, $h_2(14)=7$ -> probe sequence is 3, 10, 6 (wrap around).

- $h_1(91)=3$, $h_2(91)=7$ -> probe sequence is 3, 10, 6 (wrap around), 2, 9, 5, 1, 8, 4, 0.

Hash table
Double Hashing (cont’d)

- If the size of the table and the size of the probe step are relatively prime, i.e. the greatest common divisor is 1, in double hashing, each of these probe sequences visits all the table locations!

- \( h_1(key) = key \mod 11 \)
- \( h_2(key) = 7 - (key \mod 7) \)
Open addressing - discussion

• If need to increase the hash table size
  -> Use dynamic memory allocation for the hash table.

• Issues
  – Cannot simply double the size of the array because the resulting size is not prime
  – Data copying would incur too much overhead.
    • The hash value changes when the tableSize changes.
  – All data items need to be re-hashed.
Re-structure hash table

• Idea – to change the structure of the hash table so that it can *accommodate more than one item in the same location*.

• Two schemes
  - **Buckets**
  - **Separate chaining**
    • Each hash-table location is a linked list
    • See Figure 12-49
Separate Chaining:
Each location of the hash table is a bucket which contains a pointer to a linked list.
Class HashTable
{
    public:
    ...
    protected:
        int hashIndex(KeyType searchKey); //hash function
    private:
        enum{HASH_TABLE_SIZE=101};
        typedef ptrType hashTableType[HASH_TABLE_SIZE];
        hashTableType table; // hash table
        int size; //size of ADT table
};
Class ChainNode
{
private:
    ChainNode();
    ChainNode(const KeyedItem & nodeItem, 
             ChainNode * nextNode=NULL):
            item(nodeItem), next(nextNode) {}
    KeyedItem item;
    ChainNode * next;
    friend class HashTable;
};
Separate Chaining

• A good method of resolving collisions.
• The size of the ADT table is dynamic:
  – The size of each linked list can be as long as necessary.
Efficiency of Hashing

- Issue: all possible values for x vs. only a very small fraction used in any reasonable applications.
Analysis of Hash Table Design

• The identifier density of a hash table is the ratio $n/T$.
  - $n$ is the number of identifiers in the table
  - $T$ is the total number of possible identifiers.

• The loading density or loading factor of a hash table is $\alpha = n/(sb)$ (b: buckets, s: storage size of a bucket)
  - $n<<T$ and $b<<T$

• Hash function $h$ must map several different identifiers into the same bucket.
Analysis of Hash Table Design (cont’d)

• Two identifiers $I_1$ and $I_2$ are said to be synonyms with respect to $h$ if $h(I_1) = h(I_2)$
  – Distinct synonyms are entered into the same bucket as long as all the $s$ slots in that bucket have not been used.

• An overflow is said to occur when a new identifier is hashed into a full bucket.

• A collision is said to occur when two non-identical identifiers are hashed into the same bucket.
Hash Table Operation Efficiency

• Time for insertion or search operations depend on
  – The time required to compute the hash function \( h \) and
  – The time to search one bucket

• \( s \) is usually small

• The time is independent of \( n \)

• Guideline
  – Choose a hash function that is both easy to compute and results in very few collisions.

• Impossible to avoid collisions because the ratio \( n/T \) is usually very small.
  – Need to handle overflows.
Uniform Hash Functions

- A **uniform hash function** does NOT result in a biased use of the hash table for **random inputs**, i.e.,
  \[
  \text{Prob}\{h(x)=i\}=\frac{1}{b} \text{ for all buckets}
  \]
The relative efficiency of four collision resolution methods

Successful search

Unsuccessful search

Linear probing

Quadratic probing, double hashing

Separated chaining

Average number of probes vs. loading density
The end. 😊