Data Structures for Sets

- Many applications deal with sets.
  - Compilers have symbol tables (set of vars, classes)
  - Dictionary is a set of words.
  - Routers have sets of forwarding rules.
  - Web servers have set of clients, etc.

- A set is a collection of members
  - No repetition of members
  - Members themselves can be sets

- Examples
  - Set of first 5 natural numbers: \{1,2,3,4,5\}
  - \{x \mid x \text{ is a positive integer and } x < 100\}
  - \{x \mid x \text{ is a CA driver with } > 10 \text{ years of driving experience and 0 accidents in the last 3 years}\}
### Set Operations

<table>
<thead>
<tr>
<th>Binary operations</th>
<th>Member</th>
<th>Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member Order (=, &lt;, &gt;)</td>
<td>Find, insert, delete, split, ...</td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td>Find, insert, delete, split, ...</td>
<td>Union, intersection, difference, equal, ...</td>
</tr>
</tbody>
</table>

- **Unary operation**: min, max, sort, makenull, ...

...
Observations

• Set + Operations define an ADT.
  - A set + insert, delete, find
  - A set + ordering
  - Multiple sets + union, insert, delete
  - Multiple sets + merge
  - Etc.

• Depending on type of members and choice of operations, different implementations can have different asymptotic complexity.
Set ADT: Union, Intersection, Difference

AbstractDataType SetUID

instance multiple sets

operations

union \((s_1, s_2)\): \(\{x \mid x \text{ in } s_1 \text{ or } x \text{ in } s_2\}\)
intersection \((s_1, s_2)\): \(\{x \mid x \text{ in } s_1 \text{ and } x \text{ in } s_2\}\)
difference \((s_1, s_2)\): \(\{x \mid x \text{ in } s_1 \text{ and } x \text{ not in } s_2\}\)
Examples

- **Sets**: Articles in Yahoo Science (A), Technology (B), and Sports (C)
  - Find all articles on Wright brothers.
  - Find all articles dealing with sports medicine
- **Sets**: Students in CS10 (A), CS20 (B), and CS40 (C)
  - Find all students enrolled in these courses
  - Find students registered for CS10 only
  - Find students registered for both CS10 and CS20
  - Etc.
Set UID Implementation: Bit Vector

- Set members known and finite (e.g., all students in CS dept)

students

courses

A set
Set UID Implementation: Bit Vector

- Set members known and finite (e.g., all students in CS dept)

Operations
- Insert: \( u[k] \);
- Delete: \( u[k] \);
- Find: \( u[k] \);

Complexity: \( O(1) \): \( n \) size of the set
Set UID Implementation: Bit Vector

- Set members known and finite (e.g., all students in CS dept)

- Operations
  - Union: \( u[k] = x[k] \mid y[k] \);
  - Intersection: \( u[k] = x[k] \& y[k] \);
  - Difference: \( u[k] = x[k] \& \sim y[k] \);

- Complexity: \( O(n) \): \( n \) size of the set
Set UID Implementation: linked lists

- Bit vectors great when
  - Small sets
  - Known membership

- Linked lists
  - Unknown size and members
  - Two kinds: Sorted and Unsorted
Set UID Complexity: **Unsorted** Linked List

- **Intersection**
  
  For \( k = 1 \) to \( n \) do
  
  Advance set \( A \) one step to find \( k \)th element;
  Follow set \( B \) to find that element in \( B \);
  If found then
  
  Append element \( k \) to set \( AB \)

  End

- Searching for each element can take \( n \) steps.

- Intersection worst-case time \( O(n^2) \).
Set UID Complexity: **Unsorted Linked List**

- **Operations**
  - Insert: \( u[k] \);
  - Delete: \( u[k] \);
  - Find: \( u[k] \);

- **Operations**
  - Union: \( u[k] = x[k] \cup y[k] \);
  - Intersection: \( u[k] = x[k] \cap y[k] \);
  - Difference: \( u[k] = x[k] \cap \sim y[k] \);
Set UID Complexity: Sorted Lists

- The list is sorted; larger elements are to the right.
- Each list needs to be scanned only once.
- At each element: increment and possibly insert into A&B, constant time operation.
- Hence, sorted list set-set ADT has $O(n)$ complexity.
- A simple example of how even trivial algorithms can make a big difference in runtime complexity.
Set UID Complexity: Sorted Lists

- **Operations**
  - Insert: $u[k]$;
  - Delete: $u[k]$;
  - Find: $u[k]$;

- **Operations**
  - Union: $u[k] = x[k] \mid y[k]$;
  - Intersection: $u[k] = x[k] \& y[k]$;
  - Difference: $u[k] = x[k] \& \sim y[k]$;
Set UID: Sorted List Intersection

- **Case A** \( \text{set}_A = \text{set}_B \)
  - Include \( \text{set}_A \) (or \( \text{set}_B \)) in \( \text{set}_{AB} \)
  - Increment \( \text{set}_A \)
  - Increment \( \text{set}_B \)

- **Case B** \( \text{set}_A < \text{set}_B \)
  - Increment \( \text{set}_A \) Until
  - \( \text{set}_A = \text{set}_B (A) \)
  - \( \text{set}_A > \text{set}_B (C) \)
  - \( \text{set}_A == \text{null} \)

- **Case C** \( \text{set}_A > \text{set}_B \)
  - Increment \( \text{set}_B \) Until
  - \( \text{set}_A = \text{set}_B (A) \)
  - \( \text{set}_A < \text{set}_B (B) \)
  - \( \text{set}_B == \text{null} \)

- **Case D** \( \text{set}_A == \text{null} \) or \( \text{set}_B == \text{null} \)
  - terminate
Dictionary ADTs

• Maintain a set of items with distinct keys with:
  - \textit{find} (k): find item with key k
  - \textit{insert} (x): insert item x into the dictionary
  - \textit{remove} (k): delete item with key k

• Where do we use them:
  - Symbol tables for compiler
  - Customer records (access by name)
  - Games (\textit{positions, configurations})
  - Spell checkers
  - Peer to Peer systems (access songs by name)
Dictionaries

• A dictionary is a collection of elements each of which has a **unique search key**
  – Uniqueness criteria may be relaxed (multiset)
  – (I.e. do not force uniqueness)
• Keep track of current members, with periodic insertions and deletions into the set
• Examples
  – Membership in a club, course records
  – Symbol table (contains duplicates)
  – Language dictionary (WordSmith, Webster, WordNet)
• Similar to database
## Course Records

<table>
<thead>
<tr>
<th>key</th>
<th>student name</th>
<th>hw1</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Stan Smith</td>
<td>49</td>
<td>...</td>
</tr>
<tr>
<td>124</td>
<td>Sue Margolin</td>
<td>56</td>
<td>...</td>
</tr>
<tr>
<td>125</td>
<td>Billie King</td>
<td>34</td>
<td>...</td>
</tr>
<tr>
<td>167</td>
<td>Roy Miller</td>
<td>39</td>
<td>...</td>
</tr>
</tbody>
</table>
Dictionary ADT

- simple container methods: `size()`, `isEmpty()`, `elements()`

- query methods: `findElement(k)`, `findAllElements(k)`

- update methods: `insertItem(k, e)`, `removeElement(k)`, `removeAllElements(k)`

- special element `NO_SUCH_KEY`, returned by an unsuccessful search
How to Implement a Dictionary?

- Sequences / Arrays
  - ordered
  - unordered
- Binary Search Trees
- Skip lists
- Hashtables
Naïve Implementations

• The simplest possible scheme to implement a dictionary is “log file” or “audit trail”.
  - Maintain the elements in a linked list, with insertions occurring at the head.
  - The search and delete operations require searching the entire list in the worst-case.
  - Insertion is $O(1)$, but find and delete are $O(n)$.

• A sorted array does not help, even with ordered keys. The search becomes fast, but insert/delete take $O(n)$. 

20
Recall Arrays …

- Unordered array
  - unordered sequence
    - 34 → 14 → 12 → 22 → 18

- searching and removing takes $O(\cdot)$ time
- inserting takes $O(\cdot)$ time
- applications to log files (frequent insertions, rare searches and removals)
More Arrays

• Ordered array

• searching takes $O(\log n)$ time (binary search)

• inserting and removing takes $O(n)$ time

• application to look-up tables (frequent searches, rare insertions and removals)

• Apply binary search
Binary Searches

- narrow down the search range in stages
- “high-low” game
- `findElement(22)`
Recall Binary Search Trees...

- Implement a dictionary with a BST
  - A binary search tree is a binary tree $T$ such that
  - each internal node stores an item $(k, e)$ of a dictionary.
  - keys stored at nodes in the left subtree of $v$ are less than or equal to $k$.
  - keys stored at nodes in the right subtree of $v$ are greater than or equal to $k$. 
An Alternative to Arrays

• Unordered Array:
  – insertion: $O(1)$
  – search: $O(n)$

• Ordered Array
  – insertion: $O(n)$
  – search: $O(\log n)$

• Skip Lists:
  – insertion: $O(\log n)$
  – search: $O(\log n)$
  – And avoid the fixed-size drawback of arrays!
Skip Lists

- good implementation for a dictionary
- a series of lists \( \{S_0, S_1, \ldots, S_k\} \)
- each list \( S_i \) stores a sorted subset of the dictionary \( D \)
Skip Lists

- list $S(i+1)$ contains items picked at random from $S(i)$
- each item has probability 50% of being in the upper level list
  - like flipping a coin
- $S0$ has $n$ elements
- $S1$ has about $n/2$ elements
- $S2$ has about $n/4$ elements
- ...
Traversing Positions in a Skip List

• Assume a node $P$ in the skip list
  – $\text{after}(p)$
  – $\text{before}(p)$
  – $\text{below}(p)$
  – $\text{above}(p)$

• Running time of each operation?
Operations in a Skip List

• Use skip lists to implement dictionaries
→

• Need to deal with
  – Search
  – Insert
  – Remove
Searching

- Search for key $K$
- Start with $p =$ the top-most, left position node in the skip list
- two steps:
  1. if below($p$) is null then stop
     - we are at the bottom
  2. while key($p$) < $K$ move to the right
     go back to 1
Searching

• Search for 27
More Searching

- Search for 74
Pseudocode for Searching

Algorithm SkipSearch(k)

Input: Search key k

Output: Position p in S such that p has the largest key less than or equal to k

p = top-most, left node in S

while below(p) != null do
   p ← below(p)
   while key(after(p)) ≤ k do
      p ← after(p)
   
return p
Running Time Analysis

• log n levels $\Rightarrow$ $O(\log n)$ for going down in the skip list
• at each level, $O(1)$ for moving forward
  – why? works like a binary search
  – in skip lists, the elements in list $S(i+1)$ play the role of search dividers for elements in $S(i)$
  – (in binary search: mid-list elements to divide the search)
Insertion in Skip Lists

• First: identify the place to insert new key k
  → node p in S0 with largest key less or equal than k

• Insert new item(k,e) after p

• with probability 50%, the new item is inserted in list S1
  → with probability 25%, the new item is inserted in list S2
    • with probability 12.5%, the new item is inserted in list S3
      → with probability 6.25%, the new item is inserted in list S4
Insertion in Skip Lists

- Insert 29
Pseudocode for Insertion

**Algorithm** SkipInsert(k, e)

**Input:** Item (k, e)

**Output:** -

\[ p \leftarrow \text{SkipSearch}(k) \]

\[ q \leftarrow \text{insertAfterAbove}(p, \text{null}, \text{Item}(k, e)) \]

**while** random( ) \leq 50% **do**

**while** (above(p) == null) **do**

\[ p \leftarrow \text{before}(p) \]

\[ p \leftarrow \text{above}(p) \]
Running Time for Insertion?

• Search position for new item \((k,e)\)
  – \(O(\log n)\)

• Insertion
  – \(O(1)\)

• Total running time
  – \(O(\log n)\)
Removal from Skip Lists

- Easier than insertion

- Locate item with key k to be removed
- if no such element, return NO SUCH KEY
- otherwise, remove Item(k,e)
- remove all items found with above(Item(k,e))
Removal from Skip Lists

- Remove 18
- Running time?

S0\(-\infty\)
S1\(-\infty\)
S2\(-\infty\)
S3\(-\infty\)
Efficient Implementation of Skip Lists

- use DoublyLinkedList implementation
- + two additional pointers
  - above
  - below
- For a LinkedList \(\rightarrow\) provide pointer to head
- For a DoublyLinkedList \(\rightarrow\) provide pointers to head and tail
- For a SkipList \(\rightarrow\) ??
Hash Tables: Intuition

• Hashing is function that maps each key to a location in memory.

• A key’s location does not depend on other elements, and does not change after insertion.
  • unlike a sorted list

• A good hash function should be easy to compute.

• With such a hash function, the dictionary operations can be implemented in \( O(1) \) time.
One Simple Idea: Direct Mapping

Perm #

1
2
3
8
13
14

Student Records
Hashing: the basic idea

• Map key values to hash table addresses
  *keys* $\rightarrow$ *hash table address*

  This applies to find, insert, and remove

• Usually: *integers* $\rightarrow$ \{0, 1, 2, ..., \(H_{size}-1\)}

  Typical example: \(f(n) = n \mod H_{size}\)

• Non-numeric keys converted to numbers
  - *For example, strings converted to numbers as*
    - Sum of ASCII values
    - First three characters
Hashing: the basic idea

Perm # (mod 9)

Student Records

9
10
20
39
4
14
8
Hashing:

- **Choose a hash function** $h$; it also determines the hash table size.
- **Given an item** $x$ with key $k$, put $x$ at location $h(k)$.
- **To find if** $x$ **is in the set**, check location $h(k)$.
- **What to do if** more than one keys hash to the same value. This is called collision.
- **We will discuss two methods to handle collision:**
  - Separate chaining
  - Open addressing
Separate chaining

- Maintain a list of all elements that hash to the same value
- Search -- using the hash function to determine which list to traverse

```cpp
find(k, e)
HashVal = Hash(k, Hsize);
if (TheList[HashVal].Search(k, e))
    then return true;
else return false;
```

- Insert/deletion—once the “bucket” is found through Hash, insert and delete are list operations

```cpp
class HashTable {
    .....  
    private:
        unsigned int Hsize;
        List<E,K> *TheList;
    .....```

47
53 = 4 \times 11 + 9
53 \mod 11 = 9

insert 53
Analysis of Hashing with Chaining

• **Worst case**
  - All keys hash into the same bucket
  - a single linked list.
  - insert, delete, find take $O(n)$ time.

• **Average case**
  - Keys are uniformly distributed into buckets
  - $O(1+N/B)$: $N$ is the number of elements in a hash table, $B$ is the number of buckets.
  - If $N = O(B)$, then $O(1)$ time per operation.
  - $N/B$ is called the **load factor** of the hash table.
Open addressing

- If collision happens, alternative cells are tried until an empty cell is found.

- Linear probing:
  *Try next available position*
Linear Probing (insert 12)

12 = 1 \times 11 + 1
12 \mod 11 = 1
Search with linear probing (Search 15)

$$15 = 1 \times 11 + 4$$

$$15 \mod 11 = 4$$

NOT FOUND!
int HashTable<E,K>::hSearch(const K& k) const
{
    int HashVal = k % D;
    int j = HashVal;
    do {
        // don’t search past the first empty slot (insert should put it there)
        if (empty[j] || ht[j] == k) return j;
        j = (j + 1) % D;
    } while (j != HashVal);
    return j; // no empty slot and no match either, give up
}

bool HashTable<E,K>::find(const K& k, E& e) const
{
    int b = hSearch(k);
    if (empty[b] || ht[b] != k) return false;
    e = ht[b];
    return true;
}
Deletion in Hashing with Linear Probing

• Since empty buckets are used to terminate search, standard deletion does not work.
• One simple idea is to not delete, but mark.
• Insert: put item in first empty or marked bucket.
• Search: Continue past marked buckets.
• Delete: just mark the bucket as deleted.
• Advantage: Easy and correct.
• Disadvantage: table can become full with dead items.
Deletion with linear probing: LAZY (Delete 9)

9 = 0 \times 11 + 9
9 \mod 11 = 9

FOUND !
Eager Deletion: fill holes

Remove and find replacement:
- Fill in the hole for later searches

```c
remove(j)
{
    i = j;
    empty[i] = true;
    i = (i + 1) % D; // candidate for swapping
    if ((not empty[i]) and i!=j) {
        r = Hash(ht[i]); // where should it go without collision?
        // can we still find it based on the rehashing strategy?
        if not ((j<r<=i) or (i<j<r) or (r<=i<j))
            then break; // yes find it from rehashing, swap
        i = (i + 1) % D; // no, cannot find it from rehashing
    }
    if (i!=j and not empty[i])
        then {
            ht[j] = ht[i];
            remove(i);
        }
}
```
Eager Deletion Analysis (cont.)

- If not full
  - After deletion, there will be at least two holes
  - Elements that are affected by the new hole are
    - Initial hashed location is cyclically before the new hole
    - Location after linear probing is in between the new hole and the next hole in the search order
  - Elements are movable to fill the hole
Eager Deletion Analysis

• The important thing is to make sure that if a replacement \((i)\) is swapped into deleted \((j)\), we can still find that element. How can we not find it?
  - If the original hashed position \((r)\) is circularly in between deleted and the replacement

\[
\begin{array}{c}
\text{j} & \text{r} & i \\
\text{i} & \text{j} & \text{r} \\
\text{r} & \text{i} & \text{j} \\
\text{j} & \text{i} & \text{r} \\
\end{array}
\]

Will not find \(i\) past the empty green slot!

\[
\begin{array}{c}
\text{i} & \text{r} & \text{ } \\
\end{array}
\]

Will find \(i\)
Quadratic Probing

- **Solves the clustering problem in Linear Probing**
  - Check $H(x)$
  - If collision occurs check $H(x) + 1$
  - If collision occurs check $H(x) + 4$
  - If collision occurs check $H(x) + 9$
  - If collision occurs check $H(x) + 16$
  - ...

- $H(x) + i^2$
Quadratic Probing (insert 12)

12 = 1 x 11 + 1
12 mod 11 = 1
Double Hashing

- **When collision occurs use a second hash function**
  - \( \text{Hash}_2(x) = R - (x \mod R) \)
  - \( R \): greatest prime number smaller than table-size

- **Inserting 12**
  \( H_2(x) = 7 - (x \mod 7) = 7 - (12 \mod 7) = 2 \)
  - Check \( H(x) \)
  - If collision occurs check \( H(x) + 2 \)
  - If collision occurs check \( H(x) + 4 \)
  - If collision occurs check \( H(x) + 6 \)
  - If collision occurs check \( H(x) + 8 \)
  - \( H(x) + i \ast H_2(x) \)
Double Hashing (insert 12)

\[
12 = 1 \times 11 + 1 \\
12 \mod 11 = 1 \\
7 - 12 \mod 7 = 2
\]

<table>
<thead>
<tr>
<th>0</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>28</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>31</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
</tr>
</tbody>
</table>
Comparison of linear and random probing
Rehashing

• If table gets too full, operations will take too long.
• Build another table, twice as big (and prime).
  - Next prime number after 11 x 2 is 23
• Insert every element again to this table
• Rehash after a percentage of the table becomes full (70% for example)
Good and Bad Hashing Functions

• Hash using the wrong key
  - Age of a student
• Hash using limited information
  - First letter of last names (a lot of A’s, few Z’s)
• Hash functions choices:
  - keys evenly distributed in the hash table
• Even distribution guaranteed by “randomness”
  - No expectation of outcomes
  - Cannot design input patterns to defeat randomness
Examples of Hashing Function

- $B=100$, $N=100$, keys = $A0$, $A1$, ..., $A99$
- $\text{Hashing}(A12) = (\text{Ascii}(A)+\text{Ascii}(1)+\text{Ascii}(2)) / B$
  - $H(A18)=H(A27)=H(A36)=H(A45)$ ...
  - Theoretically, $N(1+N/B)= 200$
  - In reality, 395 steps are needed because of collision
- How to fix it?
  - $\text{Hashing}(A12) = (\text{Ascii}(A)\times2^2+\text{Ascii}(1)\times2+\text{Ascii}(2))/B$
  - $H(A12)! = H(A21)$
- Examples: numerical keys
  - Use $X^2$ and take middle numbers
Hash Functions

• Hash Functions perform two separate functions:
  1 – Convert the string to a key.
  2 – Constrain the key to a positive value less than the size of the table.

• The best strategy is to keep the two functions separate so that there is only one part to change if the size of the table changes.
Hash Functions - Key Generation

- There are different algorithms to convert a string to a key.
- There is usually a trade off between efficiency and completeness.
- Some very efficient algorithms use bit shifting and addition to compute a fairly uniform hash.
- Some less efficient algorithms give a weight to the position of each character using a base related to the ASCII codes. The key is guaranteed unique, but can be very long. Also, some of the precision gained will be lost in compression.
Hash Functions - Compression

• The compression technique generally falls into either a division method or a multiplication method.
• In the division method, the index is formed by taking the remainder of the key divided by the table size.
• When using the division method, ample consideration must be given to the size of the table.
• The best choice for table size is usually a prime number not too close to a power of 2.
Hash Functions - Compression

• In the multiplication method, the key is multiplied by a constant $A$ in the range: $0 < A < 1$

• Extract the fractional part of the result and multiply it by the size of the table to get the index.

• $A$ is usually chosen to be 0.618, which is approximately: $(\sqrt{5} - 1)/2$

• The entire computation for the index follows:

  $\text{index} = \text{floor}(\text{table\_size} \times ((\text{key} \times A) \% 1))$
Hash Functions - Example

```c
int hash(char * key)
{
    int val = 0;
    while(*key != '\0')
    {
        val = (val << 4) + (*key);
        key++;
    }
    return val;
}

int compress(int index, int size)
{
    return abs(index % size);
}
```
Collision Functions

- \( H_i(x) = (H(x) + i) \mod B \)
  - Linear probing
- \( H_i(x) = (H(x) + ci) \mod B \) (\( c > 1 \))
  - Linear probing with step-size = \( c \)
- \( H_i(x) = (H(x) + i^2) \mod B \)
  - Quadratic probing
- \( H_i(x) = (H(x) + i \times H_2(x)) \mod B \)
Probability of collision (1)

Choice of the hash function

- The requirements **high load factor** and **small number of collisions** are in conflict with each other. We need to find a suitable compromise.
- For the set $S$ of keys with $|S| = n$ and buckets $B_0$, ..., $B_{m-1}$:
  - for $n > m$ conflicts are inevitable
  - for $n < m$ there is a (residual) probability $P_K(n,m)$ for the occurrence of at least one collision.
Probability of collision (2)

How can we find an estimate for $P_K(n,m)$?

• For any key $s$ the probability that $h(s) = j$ with $j \in \{0, \ldots, m - 1\}$ is:
  
  $P_K[h(s) = j] = 1/m$, provided that there is an equal distribution.

• We have $P_K(n,m) = 1 - P_{\neg K}(n,m)$, if $P_{\neg K}(n,m)$ is the probability that storing of $n$ elements in $m$ buckets leads to no collision.
Probability of collision (3)

On the probability of collisions

• If $n$ keys are distributed sequentially to the buckets $B_0, \ldots, B_{m-1}$ verteilt (with equal distribution), each time we have $P[h(s) = j] = 1/m$.

$$P_K(n, m) = 1 - P(1) \times P(2) \times \ldots \times P(n) = 1 - \frac{m(m - 1) \ldots (m - n + 1)}{m^n}$$
Probability of collision (3)

On the probability of collisions

• The probability $P(i)$ for no collision in step $i$ is $P(i) = (m - (i - 1))/m$

• Hence, we have

For example, if $m = 365$, $P(23) > 50\%$ and $P(50) \approx 97\%$ (“birthday paradox”)
Analysis of Open Hashing

• Effort of one Insert?
  - Intuitively - that depends on how full the hash is

• Effort of an average Insert?

• Effort to fill the Bucket to a certain capacity?
  - Intuitively - accumulated efforts in inserts
Analysis of Open Hashing

• Effort to search an item (both *successful* and *unsuccessful*)?
• Effort to delete an item (both *successful* and *unsuccessful*)?
  - Same effort for successful search and delete?
  - Same effort for unsuccessful search and delete?
The End
More on hashing

- Extensible hashing
  - Hash table grows and shrinks, similar to B-trees
Issues:

• **What do we lose?**
  - Operations that require ordering are inefficient
  - **FindMax**: $O(n)$  
    $O(\log n)$ Balanced binary tree
  - **FindMin**: $O(n)$  
    $O(\log n)$ Balanced binary tree
  - **PrintSorted**: $O(n \log n)$  
    $O(n)$ Balanced binary tree

• **What do we gain?**
  - **Insert**: $O(1)$  
    $O(\log n)$ Balanced binary tree
  - **Delete**: $O(1)$  
    $O(\log n)$ Balanced binary tree
  - **Find**: $O(1)$  
    $O(\log n)$ Balanced binary tree

• **How to handle Collision?**
  - Separate chaining
  - Open addressing
CSCE 3110
Data Structures & Algorithm Analysis

Rada Mihalcea
http://www.cs.unt.edu/~rada/CSCE3110

Hashing
Reading: Chap.5, Weiss
How to Implement a Dictionary?

- Sequences
  - ordered
  - unordered
- Binary Search Trees
- Skip lists
- Hashtables
Hashing

- Another important and widely useful technique for implementing dictionaries
- Constant time per operation (on the average)
- Worst case time proportional to the size of the set for each operation (just like array and chain implementation)
Basic Idea

• Use *hash function* to map keys into positions in a *hash table*

**Ideally**

• If element *e* has key *k* and *h* is hash function, then *e* is stored in position *h(k)* of table

• To search for *e*, compute *h(k)* to locate position. If no element, dictionary does not contain *e*. 
Example

- Dictionary Student Records
  - Keys are ID numbers (951000 - 952000), no more than 100 students
  - Hash function: \( h(k) = k - 951000 \) maps ID into distinct table positions 0-1000

```
array table[1001]
...
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>buckets</td>
</tr>
</tbody>
</table>
Analysis (Ideal Case)

• O(b) time to initialize hash table (b number of positions or buckets in hash table)

• O(1) time to perform *insert*, *remove*, *search*
Ideal Case is Unrealistic

• Works for implementing dictionaries, but many applications have key ranges that are too large to have 1-1 mapping between buckets and keys!

Example:
• Suppose key can take on values from 0 .. 65,535 (2 byte unsigned int)
• Expect \( \approx 1,000 \) records at any given time
• Impractical to use hash table with 65,536 slots!
Hash Functions

• If key range too large, use hash table with fewer buckets and a hash function which maps multiple keys to same bucket:

\[ h(k_1) = \beta = h(k_2): \text{k}_1 \text{ and } k_2 \text{ have collision at slot } \beta \]

• Popular hash functions: hashing by division

\[ h(k) = k \% D, \text{ where } D \text{ number of buckets in hash table} \]
Collision Resolution Policies

• Two classes:
  – (1) Open hashing, a.k.a. separate chaining
  – (2) Closed hashing, a.k.a. open addressing

• Difference has to do with whether collisions are stored *outside the table* (open hashing) or whether collisions result in storing one of the records at *another slot in the table* (closed hashing)
Closed Hashing

- Associated with closed hashing is a rehash strategy: “If we try to place $x$ in bucket $h(x)$ and find it occupied, find alternative location $h_1(x)$, $h_2(x)$, etc. Try each in order, if none empty table is full,”
- $h(x)$ is called home bucket
- Simplest rehash strategy is called linear hashing
  \[ h_i(x) = (h(x) + i) \mod D \]
- In general, our collision resolution strategy is to generate a sequence of hash table slots (probe sequence) that can hold the record; test each slot until find empty one (probing)
Example Linear (Closed) Hashing

- D=8, keys a, b, c, d have hash values h(a)=3, h(b)=0, h(c)=4, h(d)=3

Where do we insert d? 3 already filled

Probe sequence using linear hashing:
- $h_1(d) = (h(d)+1)\%8 = 4\%8 = 4$
- $h_2(d) = (h(d)+2)\%8 = 5\%8 = 5^*$
- $h_3(d) = (h(d)+3)\%8 = 6\%8 = 6$
- etc.
- 7, 0, 1, 2

Wraps around the beginning of the table!

<table>
<thead>
<tr>
<th></th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>d</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
Operations Using Linear Hashing

- Test for membership: `findItem`
- Examine $h(k)$, $h_1(k)$, $h_2(k)$, …, until we find $k$ or an empty bucket or home bucket
- If no deletions possible, strategy works!
- What if deletions?
  - If we reach empty bucket, cannot be sure that $k$ is not somewhere else and empty bucket was occupied when $k$ was inserted
  - Need special placeholder `deleted`, to distinguish bucket that was never used from one that once held a value
- May need to reorganize table after many deletions
Performance Analysis - Worst Case

- Initialization: $O(b)$, $b$ # of buckets
- Insert and search: $O(n)$, $n$ number of elements in table; all $n$ key values have same home bucket
- No better than linear list for maintaining dictionary!
Performance Analysis - Avg Case

• Distinguish between successful and unsuccessful searches
  – Delete = successful search for record to be deleted
  – Insert = unsuccessful search along its probe sequence

• Expected cost of hashing is a function of how full the table is: load factor $\alpha = n/b$

• It has been shown that average costs under linear hashing (probing) are:
Improved Collision Resolution

• Linear probing: \( h_i(x) = (h(x) + i) \mod D \)
  – all buckets in table will be candidates for inserting a new record before the probe sequence returns to home position
  – clustering of records, leads to long probing sequences

• Linear probing with skipping: \( h_i(x) = (h(x) + ic) \mod D \)
  – \( c \) constant other than 1
  – records with adjacent home buckets will not follow same probe sequence

• (Pseudo)Random probing: \( h_i(x) = (h(x) + r_i) \mod D \)
  \( r_i \) is the \( i^{th} \) value in a random permutation of numbers from 1 to \( D-1 \)
  – insertions and searches use the same sequence of "random" numbers
1. What if next element has home bucket 0?
   \[ h(k) = k \mod 11 \] 
   \[ \rightarrow \text{go to bucket 3} \]
   Same for elements with home bucket 1 or 2!
   Only a record with home position 3 will stay.
   \[ \Rightarrow p = 4/11 \text{ that next record will go to bucket 3} \]

2. Similarly, records hashing to 7, 8, 9 will end up in 10

3. Only records hashing to 4 will end up in 4 \( (p=1/11) \); same for 5 and 6

next element in bucket 3 with \( p = 8/11 \)
Hash Functions - Numerical Values

• Consider: \( h(x) = x \% 16 \)
  – poor distribution, not very random
  – depends solely on least significant four bits of key

• Better, \textit{mid-square} method
  – if keys are integers in range 0,1,…,\( K \), pick integer C such that \( DC^2 \) about equal to \( K^2 \), then

\[
h(x) = \lfloor x^2 / C \rfloor \% D
\]

extracts middle \( r \) bits of \( x^2 \), where \( 2^r = D \) (a base-D digit)
Hash Function – Strings of Characters

• Folding Method:

```java
int h(String x, int D) {
    int i, sum;
    for (sum=0, i=0; i<x.length(); i++)
        sum+= (int)x.charAt(i);
    return (sum%D);
}
```

– sums the ASCII values of the letters in the string

• ASCII value for “A” =65; sum will be in range 650-900 for 10 upper-case letters; good when
Hash Function – Strings of Characters

- Much better: Cyclic Shift

```java
static long hashCode(String key, int D) {
    int h=0;
    for (int i=0; i<key.length(); i++) {
        h = (h << 4) | (h >> 27);
        h += (int) key.charAt(i);
    }
    return h % D;
}
```
Open Hashing

- Each bucket in the hash table is the head of a linked list
- All elements that hash to a particular bucket are placed on that bucket’s linked list
- Records within a bucket can be ordered in several ways
  - by order of insertion, by key value order, or by frequency of access order
Open Hashing Data Organization

0
1
2
3
4
D-1

...
Open hashing is most appropriate when the hash table is kept in main memory, implemented with a standard in-memory linked list.

We hope that number of elements per bucket roughly equal in size, so that the lists will be short.

If there are $n$ elements in set, then each bucket will have roughly $n/D$.

If we can estimate $n$ and choose $D$ to be roughly as large, then the average bucket will have only one or two members.
Analysis Cont’d

Average time per dictionary operation:

- $D$ buckets, $n$ elements in dictionary $\implies$ average $n/D$ elements per bucket
- *insert, search, remove* operation take $O(1+n/D)$ time each
- If we can choose $D$ to be about $n$, constant time
- Assuming each element is likely to be hashed to any bucket, running time constant, independent of $n$
Comparison with Closed Hashing

• Worst case performance is $O(n)$ for both

• Number of operations for hashing
  – 23  6  8  10  23  5  12  4  9  19
  – $D=9$
  – $h(x) = x \% D$
Hashing Problem

- Draw the 11 entry hashtable for hashing the keys 12, 44, 13, 88, 23, 94, 11, 39, 20 using the function \((2i+5) \mod 11\), closed hashing, linear probing

- Pseudo-code for listing all identifiers in a hashtable in lexicographic order, using open hashing, the hash function \(h(x) = \text{first character of } x\). What is the running time?
Hash Tables

www.cs.ucf.edu/courses/cot4810/fall04/presentations/Hash_Tables.ppt

COT4810
Ken Pritchard
2 Sep 04
History

- The term hashing was apparently developed through an analogy to compare the way the key would be mangled in a hash function to the standard meaning of hashing being to chop something up.
- 1953 – Hashing with chaining was mentioned in an internal IBM memorandum.
- 1956 – Hashing was mentioned in a publication by Arnold I. Dumey, *Computers and Automation*
- 1968 – Random probing with secondary clustering was described by Robert Morris in *CACM* 11
- 1973 – Donald Knuth wrote *The Art of Computer Programming* in which he describes and analyzes hashing in depth.
Description

• A hash table is a data structure that stores things and allows insertions, lookups, and deletions to be performed in O(1) time.
• An algorithm converts an object, typically a string, to a number. Then the number is compressed according to the size of the table and used as an index.
• There is the possibility of distinct items being mapped to the same key. This is called a collision and must be resolved.
Hash Code Generator → Number → Index

Key →

Smith → 7

Bob Smith
123 Main St.
Orlando, FL 327816
407-555-1111
bob@myisp.com
Collision Resolution

• There are two kinds of collision resolution:
  1 – Chaining makes each entry a linked list so that when a collision occurs the new entry is added to the end of the list.
  2 – Open Addressing uses probing to discover an empty spot.
• With chaining, the table does not have to be resized. With open addressing, the table must be resized when the number of elements is larger than the capacity.
Collision Resolution - Chaining

• With chaining, each entry is the head of a (possibly empty) linked list. When a new object hashes to an entry, it is added to the end of the list.

• A particularly bad hash function could create a table with only one non-empty list that contained all the elements. Uniform hashing is very important with chaining.

• The load factor of a chained hash table indicates how many objects should be found at each location, provided reasonably uniform hashing. The load factor $LF = \frac{n}{c}$ where $n$ is the number of objects stored and $c$ is the capacity of the table.

• With uniform hashing, a load factor of 2 means we expect to find no more than two objects in one slot. A load factor less than 1 means that we are wasting space.
Chaining

Bob Smith
123 Main St.
Orlando, FL 327816
407-555-1111
bob@myisp.com

Jim Smith
123 Elm St.
Orlando, FL 327816
407-555-2222
jim@myisp.com
Collision Resolution – Open Addressing

• With open addressing, the table is probed for an open slot when the first one already has an element.
• There are different types of probing, some more complicated than others. The simplest type is to keep increasing the index by one until an open slot is found.
• The load factor of an open addressed hash table can never be more than 1. A load factor of .5 indicates that the table is half full.
• With uniform hashing, the number of positions that we can expect to probe is $1/(1 - LF)$. For a table that is half full, $LF = .5$, we can expect to probe $1/(1 - .5) = 2$ positions. Note that for a table that is 95% full, we can expect to probe 20 positions.
Probing

Bob Smith
123 Main St.
Orlando, FL 327816
407-555-1111
bob@myisp.com

Jim Smith
123 Elm St.
Orlando, FL 327816
407-555-2222
jim@myisp.com
Hash Functions

• Hash Functions perform two separate functions:
  1 – Convert the string to a key.
  2 – Constrain the key to a positive value less than the size of the table.

• The best strategy is to keep the two functions separate so that there is only one part to change if the size of the table changes.
Hash Functions - Key Generation

- There are different algorithms to convert a string to a key.
- There is usually a trade off between efficiency and completeness.
- Some very efficient algorithms use bit shifting and addition to compute a fairly uniform hash.
- Some less efficient algorithms give a weight to the position of each character using a base related to the ASCII codes. The key is guaranteed unique, but can be very long. Also, some of the precision gained will be lost in compression.
Hash Functions - Compression

- The compression technique generally falls into either a division method or a multiplication method.
- In the division method, the index is formed by taking the remainder of the key divided by the table size.
- When using the division method, ample consideration must be given to the size of the table.
- The best choice for table size is usually a prime number not too close to a power of 2.
Hash Functions - Compression

- In the multiplication method, the key is multiplied by a constant \( A \) in the range:
  \[
  0 < A < 1
  \]
- Extract the fractional part of the result and multiply it by the size of the table to get the index.
- \( A \) is usually chosen to be 0.618, which is approximately:
  \[
  (\sqrt{5} - 1)/2
  \]
- The entire computation for the index follows:
  \[
  index = \text{floor}(table\_size \times ((key \times A) \% 1))
  \]
Hash Functions - Example

```c
int hash(char * key)
{
    int val = 0;
    while(*key != '\0')
    {
        val = (val << 4) + (*key);
        key++;
    }
    return val;
}

int compress(int index, int size)
{
    return abs(index % size);
}
```
Dynamic Modification

• If the total number of items to store in the table are not known in advance, the table may have to be modified. In any event, the insert and delete functions should track the number of items.

• If the table uses chaining, it will not have to be modified unless the user wants to specify a maximum load value so that performance does not deteriorate.

• If the table uses open addressing, the table will have to resized when full before another item can be stored; however, it might be resized before that point based on the load factor so that performance does not deteriorate.
Hash Table Uses

• Compilers use hash tables for symbol storage.
• The Linux Kernel uses hash tables to manage memory pages and buffers.
• High speed routing tables use hash tables.
• Database systems use hash tables.
Example

C Example

C# Example

Java Example
Summary

• A hash table is a convenient data structure for storing items that provides O(1) access time.
• The concepts that drive selection of the key generation and compression functions can be complicated, but there is a lot of research information available.
• There are many areas where hash tables are used.
• Modern programming languages typically provide a hash table implementation ready for use in applications.
References


Theory I
Algorithm Design and Analysis

(5 Hashing)

Prof. Th. Ottmann
The dictionary problem

Different approaches to the dictionary problem:

• Previously: Structuring the set of actually occurring keys: lists, trees, graphs, ...

• Structuring the complete universe of all possible keys: hashing

Hashing describes a special way of storing the elements of a set by breaking down the universe of possible keys. The position of the data element in the memory is given by computation directly from the key.
Hashing

Dictionary problem:
Lookup, insertion, deletion of data sets (keys)

Place of data set $d$: computed from the key $s$ of $d$
$\rightarrow$ no comparisons
$\rightarrow$ constant time

The memory is divided in $m$ containers (buckets) of the same size.
Examples: Hash tables - examples

- Compilers
  i int 0x87C50FA4
  j int 0x87C50FA8
  x double 0x87C50FAC
  name String 0x87C50FB2

- Environment variables (key, attribute) list
  EDITOR=emacs
  GROUP=mitarbeiter
  HOST=vulcano
  HOSTTYPE=sun4
  LPDEST=hp5
  MACHTYPE=sparc

- Executable programs
  PATH="/bin:/usr/local/gnu/bin:/usr/local/bin:/usr/bin:/bin:"
class TableEntry {
    private Object key, value;
}

abstract class HashTable {
    private TableEntry[] tableEntry;
    private int capacity;

    // Constructor
    HashTable (int capacity) {
        this.capacity = capacity;
        tableEntry = new TableEntry[capacity];
        for (int i = 0; i <= capacity - 1; i++)
            tableEntry[i] = null;
    }

    // the hash function
    protected abstract int h (Object key);

    // insert element with given key and value (if not there already)
    public abstract void insert (Object key Object value);

    // delete element with given key (if there)
    public abstract void delete (Object key);

    // locate element with given key
    public abstract Object search (Object key);
}
} // class hashTable
1. **Size of the hash table**
   Only a small subset $S$ of all possible keys (the universe) $U$ actually occurs.

2. **Calculation of the address of a data set**
   - keys are not necessarily integers
   - index depends on the size of hash table

In Java:

```java
public class Object {
    ...
    public int hashCode() {
        ...
    }
    ...
}
```
Hash function (1)

Set of keys $S$

Universe $U$ of all possible keys

Hash function $h$

Hash table $T$

$(H(U) \subseteq [-2^{31}, 2^{31} - 1])$

$h(s) =$ hash address

$h(s) = h(s') \Leftrightarrow s$ and $s'$ are synonyms with respect to $h$

address collision
Definition: Let $U$ be a universe of possible keys and \{\(B_0, \ldots, B_{m-1}\)\} a set of $m$ buckets for storing elements from $U$. Then a hash function is a total mapping

$$h : U \rightarrow \{0, \ldots, m-1\}$$

mapping each key $s \in U$ to a number $h(s)$ (and the respective element to the bucket $B_{h(s)}$).

- The bucket numbers are also called hash addresses, the complete set of buckets is
Address collisions

• A hash function $h$ calculates for each key $s$ the number of the associated bucket.

• It would be ideal if the mapping of a data set with key $s$ to a bucket $h(s)$ was unique (one-to-one): insertion and lookup could be carried out in constant time ($O(1)$).

• In reality, there will be collisions: several elements can be mapped to the same hash address.
Hashing methods

Example for $U$: all names in Java with length $\leq 40 \rightarrow |U| = 62^{40}$

If $|U| > m$ : address collisions are inevitable

Hashing methods:

1. Choice of a hash function that is as "good" as possible
2. Strategy for resolving address collisions

Load factor $\alpha$:

$$\alpha = \frac{\text{#stored keys}}{\text{size of the hash table}} = \frac{|S|}{m} = \frac{n}{m}$$
Requirements for good hash functions

Requirements

• A collision occurs if the bucket $B_{h(s)}$ for a newly inserted element with key $s$ is already taken.

• A hash function $h$ is called perfect for a set $S$ of keys if no collisions will occur for $S$.

• If $h$ is perfect and $|S| = n$, then $n \leq m$. The load factor of the hash table is $n/m \leq 1$.

• A hash function is well chosen if
  – the load factor is as high as possible,
  – for many sets of keys the number of collisions is as small as possible,
  – it can be computed efficiently.
Example of a hash function

Example: hash function for strings

```java
public static int h (String s){
    int k = 0, m = 13;
    for (int i=0; i < s.length(); i++)
        k += (int)s.charAt (i);
    return ( k%m );
}
```

The following hash addresses are generated for \( m = 13 \).

<table>
<thead>
<tr>
<th>key s</th>
<th>( h(s) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>0</td>
</tr>
<tr>
<td>Hallo</td>
<td>2</td>
</tr>
<tr>
<td>SE</td>
<td>9</td>
</tr>
<tr>
<td>Algo</td>
<td>10</td>
</tr>
</tbody>
</table>
Probability of collision (1)
Choice of the hash function

• The requirements **high load factor** and **small number of collisions** are in conflict with each other. We need to find a suitable compromise.

• For the set $S$ of keys with $|S| = n$ and buckets $B_0, \ldots, B_{m-1}$:
  – for $n > m$ conflicts are inevitable
  – for $n < m$ there is a (residual) probability $P_K(n,m)$ for the occurrence of at least one
On the probability of collisions

- If \( n \) keys are distributed sequentially to the buckets \( B_0, \ldots, B_{m-1} \) verteilt (with equal distribution), each time we have
  \[
P_{\text{c}}(n, m) = 1 - P(1) \times P(2) \times \ldots \times P(n) = 1 - \frac{m(m-1) \ldots (m-n+1)}{m^n}
  \]
  \[P[h(s) = j] = \frac{1}{m}.
  \]
- The probability \( P(i) \) for no collision in step \( i \) is \( P(i) = \frac{(m - (i - 1))}{m} \)
- Hence, we have
Common hash functions

Hash functions used in practice:

• see: D.E. Knuth: *The Art of Computer Programming*

• For $U =$ integer the [divisions-residue method] is used:

  $$h(s) = \left( \sum_{i=0}^{k-1} B^i s_i \right) \mod 2^w \mod m$$

  $h(s) = (a \times s) \mod m$ (if $a \neq 0$, $a \neq m$, $m$ prime)

• For strings of characters of the form $s = s_0 s_1 \ldots s_{k-1}$ one can use:
Choice of the hash function
- simple and quick computation
- even distribution of the data (example: compiler)

(Simple) division-residue method

\[ h(k) = k \mod m \]

How to choose of \( m \)?

Examples:

a) \( m \) even \( \rightarrow \) \( h(k) \) even \hspace{1cm} k \) even

Problematic if the last bit has a meaning (e.g. 0 = female, 1 = male)
Multiplicative method (1)

Choose constant $\theta, 0 < \theta < 1$

1. Compute $k\theta \mod 1 = k\theta - \lfloor k\theta \rfloor$

2. $h(k) = \lfloor m(k\theta \mod 1) \rfloor$

Choice of $m$ is uncritical, choose $m = 2^p$:

Computation of $h(k)$:

$p$ Bits $= h(k)$
Multiplicative method (2)

Example:

\[ \theta = \frac{\sqrt{5} - 1}{2} \approx 0.6180339 \]

\[ k = 123456 \]

\[ m = 10000 \]

\[ h(k) = \left[ 10000(123456 \times 0.61803... \mod 1) \right] \]
\[ = \left[ 10000(76300,0041151... \mod 1) \right] \]
\[ = \left[ 41.151... \right] = 41 \]

Of all numbers \( 0 \leq \theta \leq 1 \), \( \frac{\sqrt{5} - 1}{2} \) leads to the most even distribution.
Universal hashing

Problem: if $h$ is fixed $\Rightarrow$ there are $x \subseteq U$ with many collisions

Idea of universal hashing:
Choose hash function $h$ randomly

$H$ finite set of hash functions

$$h \in H : U \rightarrow \{0, \ldots, m-1\}$$

Definition: $H$ is universal, if for arbitrary $x,y \in U$:

$$\frac{\left|\{h \in H| h(x) = h(y)\}\right|}{|H|} \leq \frac{1}{m}$$

Hence: if $x, y \in U, H$ universal, $h \in H$ picked randomly

$$\Pr_{H}(h(x) = h(y)) \leq \frac{1}{m}$$
Universal hashing

Definition:

\[ \delta(x, y, h) = \begin{cases} 1, & \text{if } h(x) = h(y) \text{ and } x \neq y \\ 0, & \text{otherwise} \end{cases} \]

Extension to sets:

\[ \delta(x, S, h) = \sum_{s \in S} \delta(x, s, h) \]

\[ \delta(x, y, G) = \sum_{h \in G} \delta(x, y, h) \]

Corollary: \( H \) is universal, if for any \( x, y \in U \)

\[ \delta(x, y, H) \leq \frac{|H|}{m} \]
A universal class of hash functions

Assumptions:

• |U| = p (p prime) and |U| = {0, ..., p-1}

•Let \( a \in \{1, ..., p-1\} \), \( b \in \{0, ..., p-1\} \) and \( h_{a,b} : U \rightarrow \{0,...,m-1\} \) be defined as follows:

\[
h_{a,b} = ((ax+b) \mod p) \mod m
\]

Then:

The set

\[
H = \{h_{a,b} | 1 \leq a \leq p, 0 \leq b \leq p\}
\]
Hash table $T$ of size 3, $|U| = 5$

Consider the 20 functions (set $H$):

- $x+0$  $2x+0$  $3x+0$  $4x+0$
- $x+1$  $2x+1$  $3x+1$  $4x+1$
- $x+2$  $2x+2$  $3x+2$  $4x+2$
- $x+3$  $2x+3$  $3x+3$  $4x+3$
- $x+4$  $2x+4$  $3x+4$  $4x+4$

each (mod 5) (mod 3)

and the keys 1 und 4

We get:

$(1*1+0) \mod 5 \mod 3 = 1 = (1*4+0) \mod 5$
Possible ways of treating collisions

Treatment of collisions:

• Collisions are treated differently in different methods.

• A data set with key $s$ is called a **colliding element** if bucket $B_{h(s)}$ is already taken by another data set.

• What can we do with colliding elements?
  1. **Chaining:** Implement the buckets as linked lists. Colliding elements are stored in these lists.
  2. **Open Addressing:** Colliding elements are stored in other vacant buckets. During storage and lookup, these are found through so-called **probing**.