Hashing Algorithms

R. K. Ghosh

IIT Kanpur

Hashing & Dictionary Operations

Hashing

- Design a data structure for an ADT which allows dictionary operations, namely:
  1. Insert
  2. Delete
  3. Search (exact match, key exists or not)
- Dictionary manipulates a large number of elements, sometimes exceeding a million or so.

Linear List and Trees

- We could use balanced binary search trees for dictionary operations.
  - Require O($\log n$) time.
  - Search could give closest match.
- Linear lists also can support all dictionary operations.
  - With unsored lists takes O(n) time for delete/search. But insert requires O(1) time.
  - With sorted lists search/delete can be performed in O($\log n$) time. But insert requires O(n) time.

Hashing

- Every operation should be doable in expected $O(1)$ time when large number of keys are invoved.
- Hashing is the answer. It has two components.
  1. A Hash function, and
  2. A Hash table.
- How it operates?
  - Takes a key (of item), and computes a value and performs the operation in expected O(1) time using O(n) space.
  - Inserts, extracts or deletes the record from entry from the table indexed by the computed value.

Hashing Requirements

- **Uniformity**: The hashing function should distribute every key equally likely in the range space.
- **Low cost**: Cost of executing hashing function should small.
- **Determinism**: For a given input same hash value must be generated by a hash function.

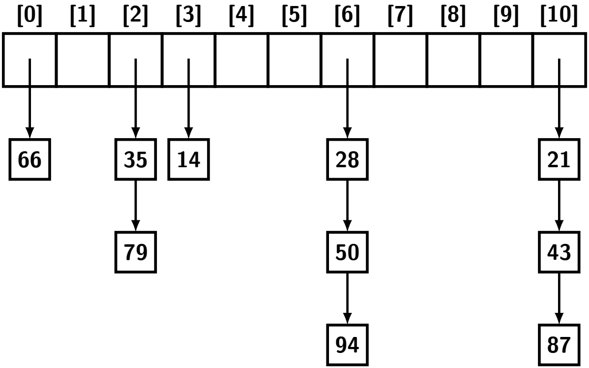
Types of Hashing

- Hash table basically stores an array of pointers to actual records.
- A NULL pointer means no record key mapped to the table entry.
- Two types of hashing:
  - **Open hashing** or **Separate chaining** and
  - **Closed hashing** or **Open addressing**.

Common Hash Functions

- Division method.
- Multiplication method.
- Mid square method.
- Folding method.

Division Method



- Uses hash function $h(x) = x \mod 11$.

Division is not a Preferred Method

- Consider the hash function that uses sum of ASCII codes of characters of k.
- It is a bad function, because it will map any string which is a permutation of same set of characters, e.g., ”break” ”brake”
- It is also not good for any set of characters whose character code sum up to same value, e.g., ”build” and ”dealt”.
- That is all strings which end up with equal ASCII sum.

Division is not a Preferred Method

- $h(k) = k \mod m$.
- If $m = 2^p$, using hash function " $\mod$ " would map any k to its lower order p bits.
- In fact, any key of the form $k = (am + x)$ would map to $h(x)$, even if m is prime.

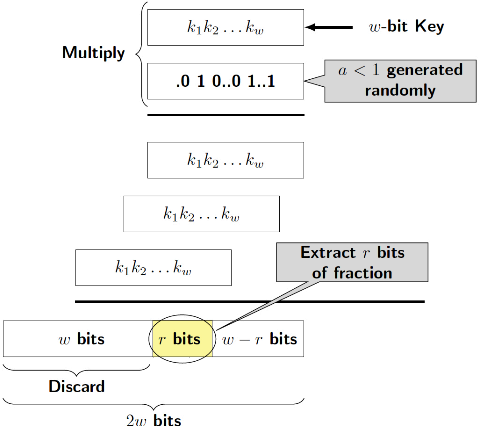
Division is not a Preferred Method

Let the base of number system be b and $b \equiv 1 \pmod{m}$:
$(b - 1 = qm)$

$$
\begin{aligned}
k \mod m &= \left(\sum_{0}^{r} b^i k_i\right) \mod m \\
&= \left(\sum_{0}^{r} (qm+1)^i k_i\right) \mod m \\
&= \sum_{0}^{r} k_i \mod m
\end{aligned}
$$

- Which means division function is bad.
- If m (table size) is a prime not close to 2^p or 10^p $(b = 10)$ then it may be ok in practice.

Multiplication Method



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

Multiplication Method

- This hash is random, because the middle bits of the result of multiplication depends on all bits of key.
- Optimal choice of a depends on keys.
- Consider the following example:
  - Let $m = 100$, $a = 1/3$.
  - For $k = 10$, $\lfloor 100 * (10 * 0.33...) \rfloor = 33$.
  - For $k = 11$, $\lfloor 100 * (11 * 0.33...) \rfloor = 36$
  - For $k = 12$, $\lfloor 100 * (12 * 0.33...) \rfloor = 39$
- Knuth claims a good choice is: $a \approx \left(\sqrt{5} - 1\right) / 2$ = 0.618033988749895.

Comparison of Two Methods

	$m = 1000$	
key	$a = 0.6180339887 49895$ $h(k) = \lfloor (m * (k * a \mod 1)) \rfloor$	$h(k) = k \mod m$
123456	931	456
123459	785	459
123496	652	496
123956	947	956
129456	131	456
193456	269	456
923456	650	456

Clearly, multiplication function distributes keys more evenly.

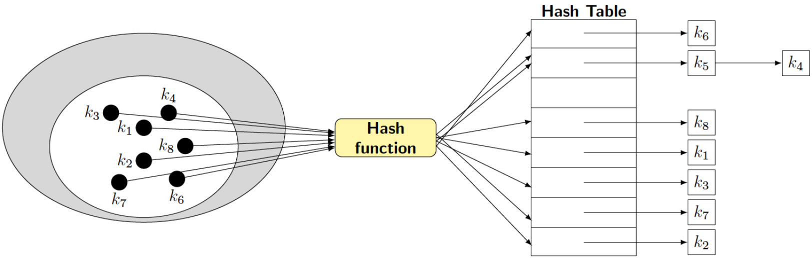
Mid-square Method

- Squares the key value and extracts same middle r values.
  - If $k = 1234$, then $k^2 = 1522756$.
  - Let table size = 100, we extract middle 2 digits $h(k) = 27$.
  - In above example we always choose 3rd and 4th digit from right.
- Like multiplication method middle r digits depend on most or all digits of the original key.

Folding Method

- Divide the key into a number of parts of equal lengths $k_1, k_2, \ldots k_p$.
- Only k_p may have less number of digits.
- Add up the parts, and ignore the last carry.
- Suppose we have 100 as table size and have following keys: 5678, 345 and 568901.
  - Parts of 5678: 56 and 78 $\implies$ 56+78=134, ignore carry, $h(5678) = 34$.
  - Parts of 345: 34 and 5 $\implies$ 34+5 = 39, so $h(345) = 39$.
  - Parts of 568901: 56, 89 and 01 $\implies$ 56+89+01 = 146, so ignore carry, $h(568901) = 46$.

Hashing by Chaining



Implemented as an array of pointers to a linked list.

Hashing by Chaining

- For inserting an element perform following steps:
  1. Compute the hash value of the element
  2. Access the pointer in the array indexed by hash value, prepend to the list.
- Collisions resolved by chaining the elements in a linked list.
- For a deletion/search perform following steps:
  1. Obtain the hash value
  2. Access the corresponding chain to find the value.

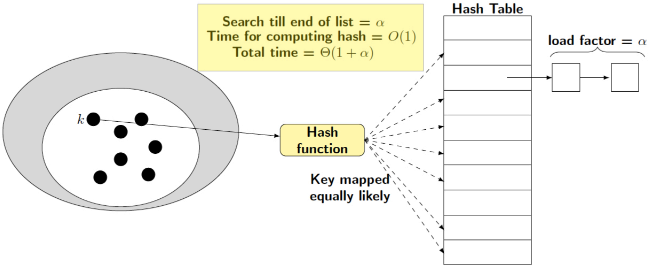
Hashing by Chaining

- Simple uniform hash function means that each key is equally likely to be hashed into any slot.
- Let $P(k)$ be the probability that k is represented in the table.
- Distributiveness means each slot $j = 0, 1, \ldots, m-1$ equally likely to be occupied:

$$\sum_{k|h(k)=j} P(k) = \frac{1}{m}.$$

- The expected length of any chain = $\frac{n}{m}$ which is called **load factor** and denoted by α.

Hashing by Chaining



- For an unsuccessful search, the number links traversed is $1+\alpha$ excluding the NULL.
- For successful search it is: $1+\alpha/2$.
  - One link (to element) has to be traversed.
  - In an average half the links ($\alpha/2$) are traversed.

Pseudo Code Initialization

```
typedef struct hTnode {
      int val;
      struct node * next;
} node;

// Initialization of pointer array for hash table
void initializeHT(node * hashTable[], int m) {
      int i;
      for (i=0; i < m; i++)
         hashTable[i] = NULL;

}
```

Pseudo Code for Search

```
node *searchKey (node *hashTable[], int k) {
    node *p;
    p = hashTable[h(k)];
    while ((p != NULL) && (p->val != k))
        p = p->next;
    if (p->val == k)
        return p;
    else
        return NULL;
}
```

Pseudo Code for Insert

```
void insertKey(node * hashTable[], int k) {
     node * newNode;
     node *ptr = searchKey(hashTable,k);
     if (ptr == NULL) {
          newNode = (node *) malloc(sizeof(node));
          newNode->val = k;
          newNode->next = hashTable[h(k)]
          hashTable[h(k)] = newNode;
     }
}
```

Pseudo Code for Delete

```
void deleteKey (node *hashTable[], int k) {
    node *save, *p;
    save = NULL;
    p = hashTable[h(k)];
    while ( p!=NULL ) {
        save = p;
        p = p->next;
    }
    if (p != NULL) {
        save->next = p->next;
        free(p);
    } else
        print("value %d not found\n", k);
}
```

Abstract Algebraic Structures

- Applications of number systems in computer algorithms motivates the study algebraic structures like groups, rings, integral domains, fields
- Provide an abstract framework for understanding and exploring properties of number systems
- Connecting the properties of the number systems to other seemingly different systems like collections of matrices or collections of determinants, etc.
- You will learn more about algebraic structure in either discrete maths or in cryptography.
- Here let us just start with a very short primer.

Groups

- A group $G = (S, +)$ is a collection of elements S with one operation "+" with following properties.
- **Associativity**: a+(b+c) = (a+b)+c.
- **Identity**: $\exists$ unique object $I \in G$ such that:
  - $\forall a \in S$, a+I = a = I+a.
- **Inverse**: For every $a \in S$, $\exists$ a unique $a^{-1} \in S$, such that $a.a^{-1} = I = a^{-1}.a$

Commutativity

It is Not a requirement. When commutativity holds, such a group is called commutative or abelian group.

Examples of Groups

- Z: with operation "+".
- Z_n: with operation mod n with "+".
- Q: set of all rational number of with operation "+".
- R^*: set of all nonzero real numbers with "$\times$".
- Set of all invertible functions with function composition $f \circ (g \circ h) = (f \circ g) \circ h$.
- $GL(2, R)$: set of 2$\times$2 invertible matrices with matric multiplication.
- Note that $GL(2, R)$ is noncommuative.

Rings

- A ring is a set R together with two operations: "+" and $\times$ which has the following propertis.
- $(R, +)$ is a commutative group.
- R is associative under $\times$
- **Multiplicative Identity**: R contains a unique element 1 such that $\forall r \in R$, $r \times 1 = r = 1 \times r$.
- **Distributivity**: Operation $\times$ distributes over "+".

Examples of Rings

- Z_n with addition and multiplication.
- $M(2, R)$, the set of 2×2 matrices with matrix addition and matrix multiplication.
- $R[x]$: set of polynomials with real coefficients with addition and multiplication of polynomials.

Field

- $(F, +, \times)$ is a field if
  - $(F, +, \times)$ is a commutative ring. (Both ring operations are commutative.)
  - For each element $x \in F$, $\exists$ a unique element $x^{-1} \in F$ which is its multiplicative inverse, i.e., $a \times a^{-1} = 1 = a^{-1} \times a$.

Examples of Field

- Z_p, when p is a prime.
- Q, R, C
- $Q[\sqrt{2}] = \{a\sqrt{2} + b : a, b \in Q\}$

Multiplicative Inverse

- Consider a result from finite field before actual proof.
- For any prime m, the set of integers

$$\mathcal{Z}_m = \{0, 1, \ldots, m-1\}$$

  with modulo m operations (+, *) defines a field.
- In a field every nonzero element has a unique multiplicative inverse.

Finding Multiplicative Inverse

- Find inverse of 19 modulo 392.
- Use Euclidean algorithm to detect if inverse exists:

$$\begin{aligned}
392 &= 19 * 20 + 12 \\
19 &= 12 + 7 \\
12 &= 7 + 5 \\
7 &= 5 + 2 \\
5 &= 2 * 2 + 1
\end{aligned}$$

- Now use backward substitution produce the inverse 227.

Finding Multiplicative Inverse

$$
\begin{aligned}
1 &= 5 + (-2) * 2 \\
&= 5 + (-2) * (7 + (-1) * 5) \\
&= 3 * 5 + (-2) * 7 \\
&= 3 * (12 + (-1) * 7) + (-2) * 7 \\
&= 3 * 12 + (-5) * 7 \\
&= 3 * 12 + (-5) * (19 + (-1) * 12) \\
&= 8 * 12 + (-5) * 19 \\
&= 8 * (392 + (-20) * 19) + (-5) * 19 \\
&= 8 * 392 + (-160) * 19 + (-5) * 19 \\
&= 8 * 392 + (-165) * 19 \\
&= 227 * 19
\end{aligned}
$$

Finite Field

For example consider $m = 7$, the elements of field are {0, 1, 2, 3, 4, 5, 6}.

z	1	2	3	4	5	6
z^{-1}	1	4	5	2	3	6

- Note that m has to be prime to become a field with modulo operation.
- Let us take $m = 10$, then elements of field: $\{1, 2, \ldots 9\}$.
- Clearly, 2 does not have any inverse in $\mathcal{Z}_{10}$.

Universal Hash Function

- So far we relied on the fact that the input is random.
- Our analysis of cost of hash operation depended on randomness of input.
- Now we let input to be arbitrary but distinct.
- But choose hash function randomly from a hash family $\mathcal{H}$.
- So analysis will depend on randomness of hash function.

Universal Hash Function

- The underlying idea is that a good hash function may emerge through a competition among the rival developers.
  - Apart from hashing programs being tested against a bench mark suite, they can also be tested by the rivals.
  - The rivals would create test cases to defeat each other's hashing schemes.
- Hashing scheme is called universal, as it will work against any adversary with the promised expectation.
- The expectation is on hash and not on the input distribution.

Universal Hash Function

- The only way one can win is to prevent an adversary from gaining an insight by using randomization.
- So, choose one at random out of several hash functions.
- An adversary can examine your code, but does not exactly know which hash will be used.
- It guarantees that for any two distinct keys x, and y the probability of collision is: $1/m$, where m is the table size.

Universal Hash Function

Definition

Let U be a universe of keys, and let $\mathcal{H}$ be a finite collection of hash functions mapping U to $\{0, 1, \ldots, m-1\}$.

Definition

$\mathcal{H}$ is universal, if for all distinct keys $x \neq y$, the number of hash functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is precisely $\frac{|\mathcal{H}|}{m}$.

From definiton 2, if h chosen randomly from $\mathcal{H}$ we have:

$$\frac{\text{\# functions mapping } x \text{ and } y \text{ to same location}}{\text{Total \# of functions}} = \frac{\frac{|\mathcal{H}|}{m}}{|\mathcal{H}|} \leq \frac{1}{m}$$

Universal Hash Function

Theorem

Suppose n keys to be hashed into a table of size m, then choose a hash function h randomly from the set $\mathcal{H}$, Under the stated conditions, the expected number of collisions with any key x is given by:

$$E(\text{\# of collision with } x) \leq 1 + \frac{n}{m} = 1 + \alpha$$

Universal Hash Function

Proof.

Let C_{xy} be the random variable denoting if y collides with x:

$$C_{xy} = \begin{cases} 1, \text{ if } h(x) = h(y) \\ 0, \text{ otherwise} \end{cases}$$

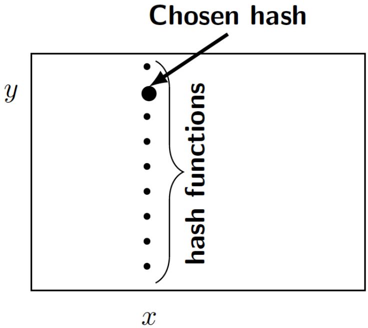
Then expected value of number of collisions

$E(C_x) = E(\sum_{y \neq x} C_{xy}) = \sum_{y \neq x} E(C_{xy})$ (by linearity)

$E(C_{xy}) = Pr[h(x) = h(y)] = 1/m$ (by universality of hash),

So $E(C_x) = \sum_{y \neq x}(1/m) = (n-1)/m \leq n/m$ $\square$

Universal Hash Function



- The theorem essentially implies that if a set of universal hash function exists then choosing a hash function from this set ensures that keys are evenly distributed.

Proof (contd).

- In the above expression we only considered the cases when x and y are distinct.
- Since x collides with itself 1 more probe will necessary for x to account for all keys that collide with x.
- So, the expected number of probes will be less than equal to $\leq 1 + \alpha$.

$\square$

Constructing a Universal Hash Function

- Works when table size m is prime.
- Decompose every key k into an r-digit integer $k_{r-1}k_{r-2}\ldots k_0$ of base-m.
- Eg., with m = 11, k=46793 is represented as vector: $\langle 4, 6, 7, 9, 3\rangle$ and its value is $3 * 11^0 + 9 * 11^2 + 7 * 11^3 + 6 * 11^4 + 4 * 11^5$.
- Next pick a random vector $a = \langle a_0, a_1, \ldots, a_r\rangle$, where $0 \leq a_i \leq m - 1$.
  - Picking vector a means picking of a random hash function.
  - $\langle a_0, a_1, \ldots, a_{r-1}\rangle$ essentially becomse index for picking a random hash function.
- Compute dot product hash

$$h_a(k) = \left(\sum_{0\leq i\leq r-1} a_i k_i\right) \mod m$$

Size of Set of Hash Functions

- There can be at most m^r vectors of length r, with each value being a m-base digit.
- So m^r hash functions correspond to possible choices for vectors $\langle a_0, a_1, \ldots, a_{r-1} \rangle$.
- Now we have to prove that these hash functions form a universal set of hash functions.

Universal Hashing

Theorem

The construction of family of hash functions as specified by random choice of $\langle a_0, a_1, \ldots, a_{r-1} \rangle$ is universal.

Proof.

- We need to show that for any two distinct keys x and y, $Pr[h_a(x) = h_a(y)] \leq \frac{1}{m}$
- Decompose each of x and y as a r-digit base m integer.
- Since $x \neq y$, we should have $x_i \neq y_i$ at least at one position $0 \leq i \leq r - 1$.
- WLOG assume that $x_0 \neq y_0$ (may also be $x_d \neq y_d$).
- If they differ in another position arguments remain same.

□

Proof for Construction of Universal Hashing

Proof (contd).

$$h_a(x) = \sum_0^{r-1} a_i x_i, \text{ and } h_a(y) = \sum_0^{r-1} a_i y_i$$

Therefore,

$$\textstyle\sum_0^{r-1} a_i(x_i - y_i) \equiv 0 \pmod{m}$$

$$a_0(x_0 - y_0) + \textstyle\sum_1^{r-1} a_i(x_i - y_i) \equiv 0 \pmod{m}$$

$$a_0(x_0 - y_0) \equiv -\textstyle\sum_1^{r-1} a_i(x_i - y_i) \pmod{m}$$

□

Proof for Construction of Universal Hashing

Proof (contd).

- Since, $x_0 \neq y_0$, $x_0 - y_0$ is nonzero, $\exists$, $(x_0 - y_0)^{-1}$ in $\mathcal{Z}_m$.
- Multiply both side of above modulo expression by the inverse $(x_0 - y_0)^{-1}$.
- We get

$$a_0 \equiv \left(-\sum_{1}^{r-1} a_i(x_i - y_i)\right)(x_0 - y_0)^{-1}$$

- Which implies a_0 is a fixed value computed from a function of other a_i values.

$\square$

Proof for Construction of Universal Hashing

Proof (contd).

- So, once a set of a_i's , for $i > 0$, has been fixed, only one value of a_0 is possible.
- The number of possible choices of a_i's can be m^{r-1} which produces m^{r-1} different values of a_0's.
- So, the possibility of a clash in $h_a(x)$ and $h_a(y)$ is:

$$\frac{m^{r-1}}{m^r} = \frac{1}{m}.$$

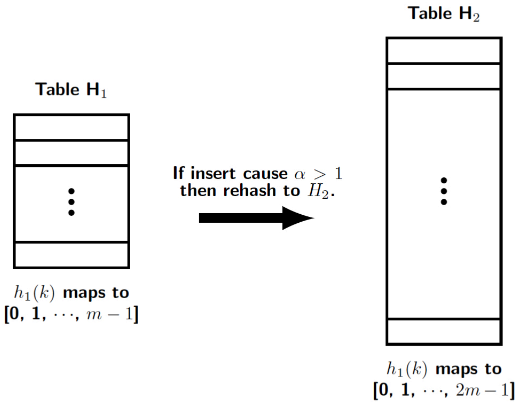
- Therefore, construction as suggested is universal.

$\square$

Choosing Table Size m

- Ideally, choice of m should be such that it is sufficient for all possible keys, i.e., O(n).
- But may lead to lot of empty slots.
  - E.g., consider airport codes (3 letters).
  - So, $n = 26^3 = 17576$.
  - But many of three letter codes are not valid airport codes.
- To handle problem of empty slots, initially choose a small number for m, then grow or shrink m according to requirement.

Expansion of Table Size



- If $n > m$ resize table to $2m$ or create a table of twice the size.

Expansion of Table Size

- Initially choose a small number.
- After doubling $\alpha = n/m = 1/2$ because $n = m/2$.
- Each item is now inserted from old table to new table using a new hash function h' .
- $m/2$ insertions can be done to new table before load factor exceeds 1 and table size is doubled again.
- So, the expansion cost $2m$ of growing table should be distributed over $m/2$ insertion cost = O($2m/0.5m$) which is O(1).
- Implies that due to distribution cost performance does not get affected.

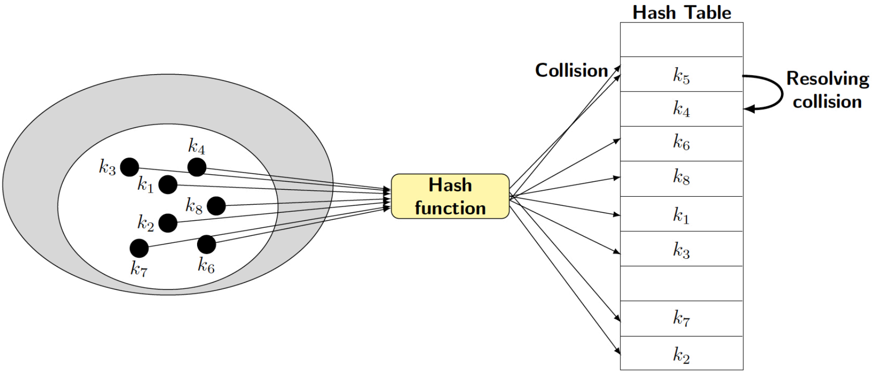
Expansion of Table Size

- Starting from size 1, the expansion cost until reaching size n: $1 + 2 + 4 + \ldots + 2^{\log n}$.
- Deletes only help, so the cost will be $O(n)$.
- But starting with size 1 growing by 1 each time would cost: $1 + 2 + 3 + \ldots + n = O(n^2)$

Shrinking Table Size

- With large number of deletions, table size requirement goes down.
- Shrink the table when $n = m/2$, wait for next $m/4$ deletion to halve the size.
- So, cost of $m/2$ distributed over $m/4$ deletions, and the cost per deletion is O($0.5m/0.25m$) = O(1).
- Shrinking cost is thus O(1) still.
- But then if insert and delete happen alternatively when $n = m/2$, then growing and shrinking oscillates.
- So, shrink table only when $n = m/4$.

Hashing with Open Address



Hashing with Open Address

- A hash function would work properly if it can specify the order of probing for empty slots.
- $h : U \times \{0, 1, \ldots, m-1\}_{trials} \rightarrow \{0, 1, \ldots, m-1\}$
- It produces a vector (assuming m probes)

$$h(k, 1), h(k, 2), \ldots, h(k, m-1),$$

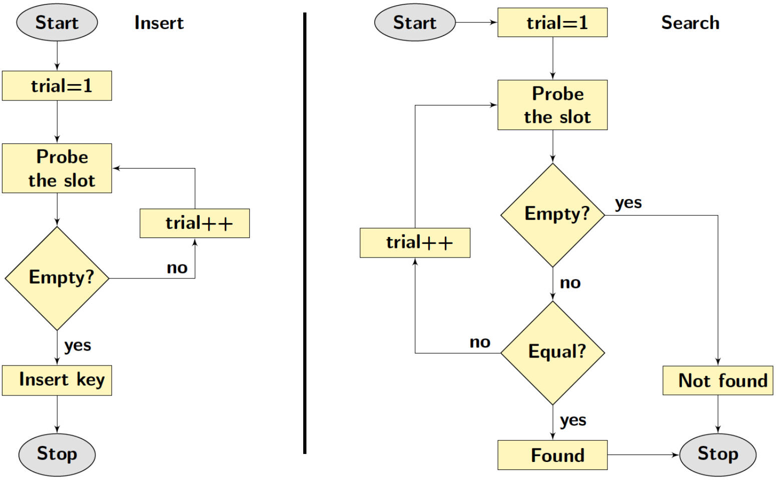
  which is a permutation of the slots 1, 2, ..., $m - 1$.
- The idea is: the entire table should be used.
- Equivalently, the probe sequence should eventually be able to discover if any empty slot is left.

Hashing with Open Address

0	
1	567
2	139
3	**598**
4	225
5	
6	455
7	

- $h(567, 1) = 1$, $h(139, 1) = 2$, $h(225, 1) = 4$ & $h(455, 1) = 6$.
- Now we have to insert 598, and $h(598, 1) = 2$, but find the slot occupied, so first trial fails.
- Assuming $h(598, 2) = 6$, second trial also fails
- Finally, on third trial $h(598, 3) = 3$, and slot 3 is found to be empty.

Hashing with Open Address



Deletion in Open Addressing

- When a deletion happens, then instead of making the slot empty (flag), mark it **deleted**.
- So, **search** and **insert** must change a little.
- **Search** must make sure to skip slots marked both **deleted** or **occupied**.
- **Insert** must treat slot marked **deleted** as an empty slot.

Resolving Collision by Linear Probing

- Let $h'(x) = x \mod m, m = 10$.
- $h(x, 0) = h'(x)$ and $h(x, i) = (h'(x) + i) \mod m$ for $i = 0, 1, 2, \ldots$.
- Let a collision occur for $x = 72$, i.e., slot 2 is occupied.
- Then $h(72, 1) = (h'(72) + 1) \mod 10 = 3$ is searched.
- If 3 is occupied then $h(72, 2) = (h'(72) + 2) \mod 10 = 4$ is searched.
- If 4 is occupied then $h(72, 3) = (h'(72) + 3) \mod 10 = 5$ is searched.
- Clustering known as primary clustering occurs in linear probing, as consecutive groups of occupied slots keep growing.

Primary Clustering

Definition (Primary Clustering)

Primary clustering occurs if a new key mapped into a previously occupied slot is moved to the next sequentially available slot. The keys tend to occupy consecutive slots. As a result, any new insertion falling into any of slots of the cluster causes it to grow by one.

- So, linear probing suffers from primary clustering problem.

Solving Primary Clustering

- Quadratic probing uses: $h(k,i) = (h'(k) + c_1.i + c_2.i^2) \mod m$, where c_1, c_2 are constants and $c_2 \neq 0$.
- It eliminates primary clustering.
- But to ensure all positions are searched, c_1, c_2 and m need to be constrained.
- Quadratic probing is free from primary clustering, may suffer from secondary clustering.
- If two keys have same initial hash values, then the same probe sequence will be followed for both.
- With quadratic probing, the probability for multiple collisions increases as the table becomes full.
- Usually encountered when the hash table is more than half full.

Example of Quadratic Probing

- Let $m = 10$ and initial hash function be $h'(x) = x \mod m$.
- Initial status of our hash table is:

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

- Let us start inserting 7 keys: 72, 27, 24, 36, 63, 81 and 101 into to the above table.
- For the first key 72, $h(72, 0) = (72 \mod 10 + 1.0 + 3.0) \mod 10 = 2 \mod 10 = 2$.
- So 72 inserted into postion 2.

Example of Quadratic Probing (contd)

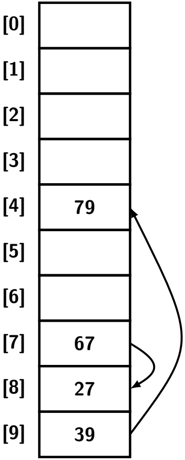
- Next 27 is inserted into position 7 because $h(27, 0) = (27 \mod 10 + 1.0 + 3.0) \mod 10 = 7 \mod 10 = 7$.
- Third key 24 is inserted into position 4 because $h(24, 0) = (24 \mod 10 + 1.0 + 3.0) \mod 10 = 6 \mod 10 = 6$.
- Fourth key 36 is inserted into position 6 because $h(36, 0) = (36 \mod 10 + 1.0 + 3.0) \mod 10 = 6 \mod 10 = 6$.
- Next 63 is inserted into position 3 because $h(63, 0) = (63 \mod 10 + 1.0 + 3.0) \mod 10 = 3 \mod 10 = 6$.

Example of Quadratic Probing (contd)

- Sixth key 81 is inserted into position 1 because $h(81, 0) = (81 \mod 10 + 1.0 + 3.0) \mod 10 = 1 \mod 10 = 1$.
- Last key 101 hashes into postion 1. But it is occupied.
- So, we compute $h(101, 1)$
- $h(101, 1) = (101 \mod 10 + 1.1 + 3.1) \mod 10 = 5 \mod 10 = 5$
- Since 5 is vacant 101 inserted into table position 5.
- The final table looks as follows:

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	101	36	27	-1	-1

Double Hashing



- Uses a second hash function for collision resolution.
  - It must never evaluate to 0.
  - It must ensure all slots are probed.
- Popular second hash function is: $h_2(k) = R - (k \mod R)$, where $R < m$ is a prime number.
- Example: $m = 10$, $R = 7$, insert keys: 67, 27, 39, 79 $h_1(x) = x \mod 10$ and $h_2(x) = 7 - (x \mod 7)$

Open Addressing: Unsuccessful Search

Let us analyze both unsuccessful/successful searches ignoring clustering, assuming all probes sequences are likely.

First consider unsuccessful search.

- Let p_i be probability of exactly i probes hitting occupied slots.
- Define probability q_i of at least i probes hitting occupied slots: $q_i = \left(\frac{n}{m}\right)^i = \alpha^i$.
- Expected number of probes in unsuccessful search:

p_1	p_2	p_3	$\cdots$	q_1
	p_2	p_3	$\cdots$	q_2
		p_3	$\cdots$	q_3
		$\vdots$		$\vdots$
		$\sum_1^\infty ip_i$		$\sum_1^\infty q_i$

$$1 + \sum_{i=1}^{\infty} i.p_i = 1 + \sum_{i=1}^{\infty} q_i = \frac{1}{1-\alpha}$$

Open Addressing: Successful Search

- If a key is inserted on $(i+1)$st attempt, the previous i searches must have failed.
- Probability for i unsuccessful searches is $1-(i/m)$ (at least i probes access occupied slots).
- The number of probes = $1/(1-(i/m)) = m/(m-i)$
- For a successful search, average number of probes is given by:

$$\frac{1}{n}\sum_{i=0}^{n-1}(\text{\# of probes in inserting key in } (i+1)st \text{ attempt})$$

Open Addressing: Successful Search

- Therefore, average number of probes for successful search:

$$\begin{aligned}\frac{1}{n}\sum_{i=0}^{n-1}\left(\frac{m}{m-i}\right) &= \frac{m}{n}\sum_{i=m}^{n-m+1}\left(\frac{1}{i}\right)\\ &\approx \frac{m}{n}\int_{i=m}^{n-m}\left(\frac{1}{x}dx\right)\\ &= \frac{m}{n}\ln\frac{m}{m-n}\\ &= \frac{1}{\alpha}\ln\left(1-\alpha\right)\end{aligned}$$

Perfect Hash Function

- Hash function is perfect if all lookups require O(1) time.
- It is possible only in situation where set of keys is known in advance.
- Construction of such specialized hash functions is tedious and primarily used for example in case of key words of a programming language.
- The basic hash function is of the form:

$$h(S) = S.len() + g(S[0]) + g(S[S.len() - 1]),$$ where

$g()$ is constructed using a different algorithm.

Perfect Hash Function

- It has three phases:
  - Computing frequences of letter in string S.
  - Ordering the words.
  - Searching: assigns a value, checks with assigned value is ok, or it leads to a clash. If yes try out an alternative value.

Perfect Hash Function

calliope
clio
erato
euterpe
melpomene
polyhymnia
terpsichore
thalia
urbania

Frequencies of first and last letter in word.

letter:	e	a	c	o	t	m	p	u
freq:	6	3	2	2	2	1	1	1

Now add the frequncies of first and last letter, determining word scores and sort them in that order.

calliope	8
clio	4
erato	8
euterpe	12
melpomene	7
polyhymnia	4
terpsichore	8
thalia	5
urania	4

Unsorted words

euterpe	12
calliope	8
erato	8
terpsichore	8
melpomene	7
thalia	5
clio	4
polyhymnia	4
urania	4

Sorted words

Perfect Hash Function

- Take the keys in order, and assign g values for the first and the last letter in such a way that each key gets a distinct value.

key	$g(key)$	$h(key)$	Slot of table
euterpe	$e = 0$	7	7 - Ok
calliope	$c = 0$	8	8 - Ok
erato	$o = 0$	5	5 - Ok
trepsichore	$t = 0$	11	2 - Ok
melpomene	$m = 0$	9	0 - Ok
thalia	$a = 0$	6	6 - Ok
polyhymnia	$p = 0$	10	1 - Ok
clio	none	4	4 - Ok

Perfect Hash Function

- Restrict the assignment step to a constant (say 5).
- As can be seen the assignment to the next key is not possible.

key	$g(key)$	$h(key)$	Slot of table
urania	$u = 0$	6	6 - Reject
urania	$u = 1$	7	7 - Reject
urania	$u = 2$	8	8 - Reject
urania	$u = 3$	9	0 - Reject
urania	$u = 4$	10	1 - Reject

Perfect Hash Function

- Change the assignment there and continue from there.

key	$g(key)$	$h(key)$	Slot of table
polyhymnia	$p = 0$	10	1 - Reject
polyhymnia	$p = 1$	11	2 - Reject
polyhymnia	$p = 2$	12	3 - Ok
urania	$u = 0$	6	1 - Reject
urania	$u = 1$	7	2 - Reject
urania	$u = 2$	8	3 - Reject
urania	$u = 3$	9	0 - Reject
urania	$u = 4$	10	1 - Ok

Summary

- Important hashing functions such as: dvision, multiplication, mid sequare and folding are discussed.
- Hashing by chaining, pseudocode and its analysis were presented.
- Universal hash function with its complete analysis were presented.
- Table growing and shrinking were also discussed.
- Hash with open addressing also discussed.
- Finally, an idea of perfect hashing presented with an example.