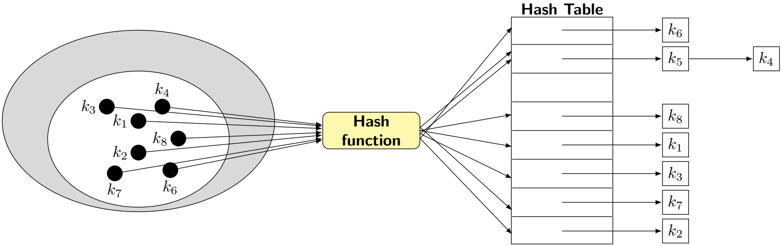
Hashing by Chaining



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.

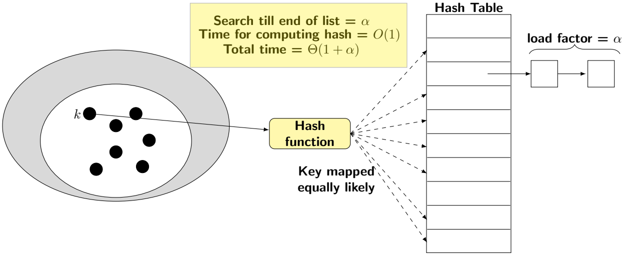
Hashing by Chaining

- 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 α excluding the NULL.
- For successful search it is: $1 + \alpha/2$.
  - One link has to be traversed any way.
  - In an average half the links will be 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);
}
```

Universal Hash Function

- Universal hashing defines a family of hash functions $\mathcal{H}$.
- A randomly chosen hash is picked from $\mathcal{H}$ to mapp the keys.
- The idea is that a good hashing scheme 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.

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 $x \neq y$, $|\{h \in \mathcal{H} : h(x) = h(y)\}| = \frac{|\mathcal{H}|}{m}$.

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

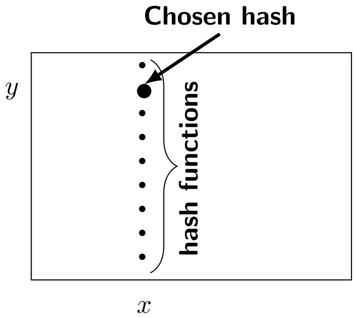
Theorem

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:

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

$\frac{n}{m} = \alpha$ is known as load factor.

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.

Universal Hash Function

Proof.

Let C_x be the random variable denoting the number of keys in table T colliding with x. Define

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

Then $E(c_{xy}) = 1/m$ and $C_x = \sum_{y \in T - \{x\}} c_{xy}$ □

Universal Hash Function

Proof (contd).

Now derive $E(C_x)$:

$$\begin{aligned} E(C_x) &= E\left(\sum_{y \in T - \{x\}} c_{xy}\right) \\ &= \sum_{y \in T - \{x\}} E(c_{xy}), \text{ by linearity of expectations} \\ &\leq \sum_{y \in T - \{x\}} \frac{1}{m} = \frac{n-1}{m} \end{aligned}$$

□

Universal Hash Function

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 $\leq 1 + \alpha$.

$\square$

Constructing a Universal Hash Function

- Works when m is prime.
- Every key is decomposed into $r+1$ digits of base m, where $0 \leq k_i \leq m-1$ (where m is table size).
- For example, let size m = 11, and key=46793.
- key is represented as vector: $\langle 4, 6, 7, 9, 3 \rangle$ and its value is $3 * 11^0 + 9 * 11^2 + 7 * 11^3 + 6 * 11^4 + 4 * 11^5$.
- Then 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 actually means picking of a random hash function. In other words, a serves as an index for picking a random hash functions.
- Compute $h_a(k) = \left(\sum_{0 \leq i \leq r} a_i k_i\right) \mod m$

Size of Set of Hash Functions

- How many vectors of length $r + 1$ can be there, where each value can be a m base digit?
  - It will be m^{r+1}.
- So there are m^{r+1} hash functions or possible choices for vectors $\langle a_0, a_1, \ldots, a_r \rangle$.
- Now we have to prove that these hash functions form a universal set of hash functions.

Finite Fields: A Digression from Hashing

- 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.

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 Hashing

Theorem

The construction of family of hash functions as specified by random choice of $\langle a_0, a_1, \ldots, a_r \rangle$ is universal.

Proof.

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

□

Proof for Construction of Universal Hashing

Proof (contd).

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

Therefore,

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

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

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

□

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

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

- 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 which produces m^r different values of a_0's.
- So, the possibility of a clash in $h_a(x)$ and $h_a(y)$ is:

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

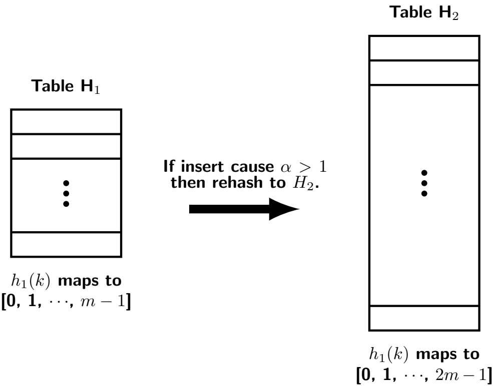
- Therefore, construction as suggested is universal.

□

Choosing Table Size m

- Ideally, m should be sufficient for all possible keys.
- But could run into problem of sparsity.
  - For example, table for airport codes (3 letters), size requirement: $26^3 = 17576$.
  - Not all three letter codes are valid airport codes.
- Realistically, $m = O(n)$ (upper bound) where n estimated number of keys.
- However, initially choose a small number then grow or shrink 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.

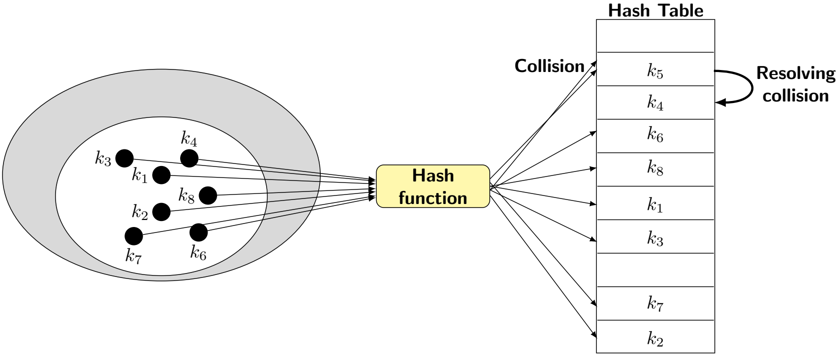
Expansion of Table Size

- 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

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

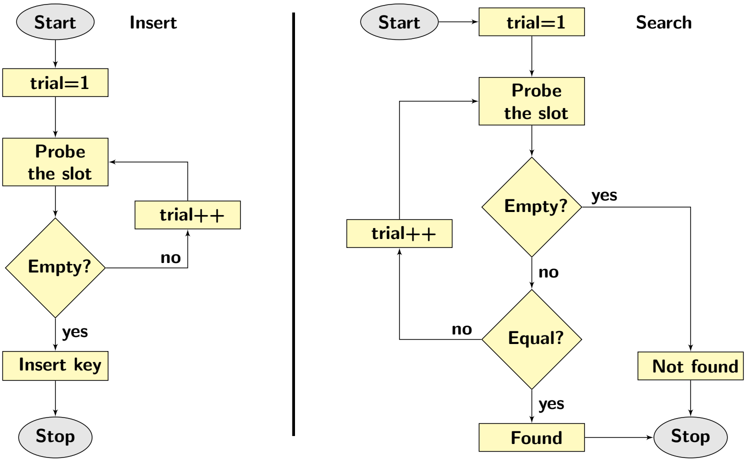
  which is a permutation of the slots 1, 2, ..., $m-1$.
- The idea is: the entire table should be used.
- Equivalently, the probe sequence should eventually be able to discover if any empty slot is left.

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.

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.

- Primary clustering occurs in linear probing, as consecutive groups of occupied slots keep growing.

Solving Primary Clustering

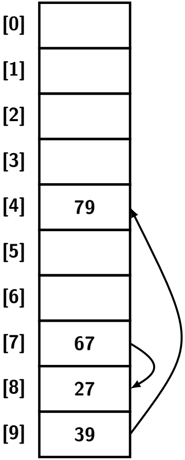
- Quadratic probing uses: $h(k,i) = (h(k) + i^2) \mod m$
- Solves primary clustering as it moves i^2 slots from the point where collision occurs.
  - Probe sequence: $h(x)$, $h(x)+1$, $h(x)+4$, etc.
  - So it probes at distances 1, 3, 5, 7,...
  - So, at most half the slots are explored as alternative locations.
  - Consequently, if a table is half full it always lead to the same set of slots being probed creating a new clustering phenomenon called Secondary clustering.

Secondary Clustering

Definition (Secondary Clustering)

This clustering is less severe. It happens if the two keys have same initial hash value. Secondary clustering occurs both with linear probing and quadratic probing.

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:

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

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.