Introduction to Algorithms

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

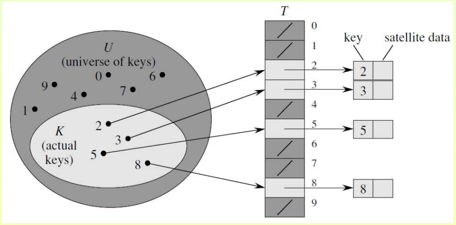
Subsection 1

Direct-Address Tables

Direct-Address Tables

- **Direct addressing** is a simple technique that works well when the universe U of keys is reasonably small.
- Suppose an application needs a dynamic set in which each element has a key drawn from the universe $U = \{0, 1, \ldots, m-1\}$, where m is not too large and no two elements have the same key.
- To represent the dynamic set, we use an array, or **direct-address table**,

  denoted by $T[0 \ldots m-1]$, in which each position, or **slot**, corresponds to a key in the universe U: Slot k points to an element in the set with key k. If the set contains no element with key k, then $T[k] = \text{NIL}$.



The Dictionary Operations

```
DirectAddressSearch($T, k$)
1. return $T[k]$
```

```
DirectAddressInsert($T, x$)
1. $T[x.\text{key}] = x$
```

```
DirectAddressDelete($T, x$)
1. $T[x.\text{key}] = \text{NIL}$
```

- Each of these operations requires $O(1)$ time.
- For some applications, the elements in the dynamic set can be stored in the direct-address table's slot instead of in an object external to the direct-address table, thus saving space.
- Moreover, it is often unnecessary to store the key field of the object, since if we have the index of an object in the table, we have its key, but some way to tell if the slot is empty must be present.

Subsection 2

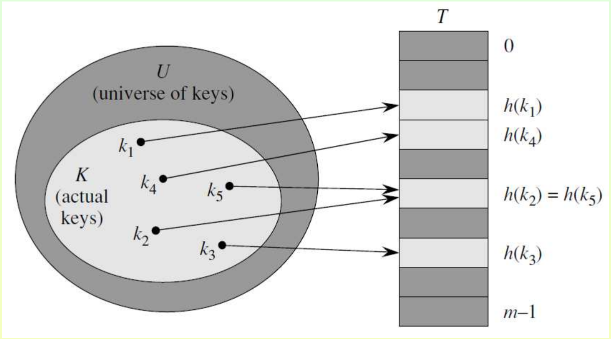
Hash Tables

Hash Tables

- If the universe U is large, storing a table T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer.
- Furthermore, the set K of keys actually stored may be so small relative to U that most of the space allocated for T would be wasted.
- When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a **hash table** requires much less storage than a direct address table.
  - The storage requirements can be reduced to $\Theta(|K|)$ while we maintain the benefit that searching for an element still requires only $O(1)$ time.
  - The drawback is that this bound is for the average time, whereas for direct addressing it holds for the worst-case time.
- With direct addressing, an element with key k is stored in slot k, whereas with hashing, this element is stored in slot $h(k)$, i.e., a **hash function** h is used to compute the slot from the key k.

How Hash Tables Work

- The hash function h maps the universe U of keys into the slots of a hash table $T[0 \dots m-1]$: $h : U \to \{0, 1, \dots, m-1\}$.



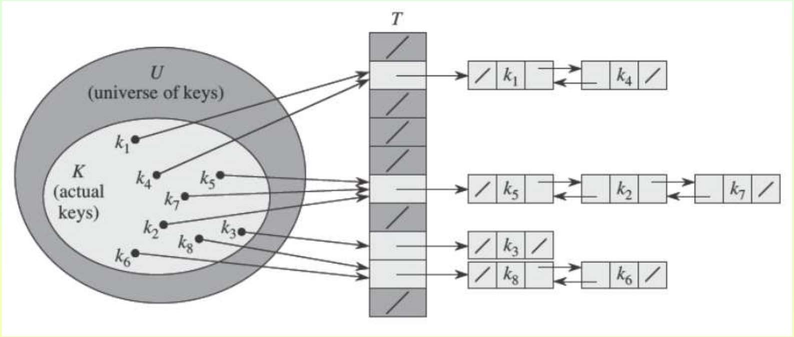
We say that an element with key k **hashes** to slot $h(k)$ or that $h(k)$ is the **hash value** of key k.

Dealing with Collisions

- If two keys hash to the same slot a **collision** arises.
- There are effective techniques for resolving the conflict created by collisions:
    - By choosing a hash function h appearing to be "random" we may minimize the number of collisions.
      A hash function h, however, must be deterministic in that a given input k should always produce the same output $h(k)$.
    - Since $|U| > m$, avoiding collisions altogether is impossible.
      A well-designed, "random"-looking hash function can minimize the number of collisions, but a method for resolving the collisions that do occur is still required.
- The simplest collision resolution technique is called **chaining**.
- An alternative method for resolving collisions is **open addressing**.

Collision Resolution by Chaining

- In chaining, we put all the elements that hash to the same slot in a linked list:



- Slot j contains a pointer to the head of the list of all stored elements that hash to j; If there are no such elements, slot j contains NIL.

Dictionary Operations when Using Chaining

- The dictionary operations on a hash table T are easy to implement when collisions are resolved by chaining.

```
ChainedHashInsert($T, x$)
1. insert $x$ at the head of list $T[h(x.\text{key})]$
```

```
ChainedHashSearch($T, k$)
1. search for an element with key $k$ in list $T[h(k)]$
```

```
ChainedHashDelete($T, x$)
1. delete $x$ from the list $T[h(x.\text{key})]$
```

- The worst-case running time for insertion is $O(1)$.
- For searching the worst-case running time is proportional to the length of the list.
- For deletion of an element x, assuming the lists are doubly linked, we need $O(1)$ time also.

Analysis of Hashing with Chaining

- We look at the time it takes to search for an element with a given key.
- Given a hash table T with m slots that stores n elements, we define the **load factor** α for T as $\frac{n}{m}$, i.e., the average number of elements stored in a chain.
- The analysis will be in terms of α, which can be less than, equal to, or greater than 1.
- The worst-case behavior of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n.

  The worst-case time for searching is thus $\Theta(n)$ plus the time to compute the hash function.
- The average performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

  In the analysis, we assume **simple uniform hashing**, i.e., that any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to.

Setting Up the Analysis of the Average Case

- For $j = 0, 1, \ldots, m-1$, denote the length of the list $T[j]$ by n_j.
- Then $n = n_0 + n_1 + \ldots + n_{m-1}$.
- The average value of n_j is $E[n_j] = \alpha = \frac{n}{m}$.
- We assume that the hash value $h(k)$ can be computed in $\mathrm{O}(1)$ time.
- Then the time required to search for an element with key k depends linearly on the length $n_{h(k)}$ of the list $T[h(k)]$.
- Setting aside the $\mathrm{O}(1)$ time required to compute the hash function and to access slot $h(k)$, let us consider the expected number of elements examined by the search algorithm.

  This is the number of elements in the list $T[h(k)]$ that are checked to see if their keys are equal to k.

  We consider two cases:
  - The search is unsuccessful: no element in the table has key k.
  - The search successfully finds an element with key k.

The Unsuccessful Search Case

Theorem

In a hash table in which collisions are resolved by chaining, an unsuccessful search takes expected time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

- Under the assumption of simple uniform hashing, any key k not already stored in the table is equally likely to hash to any of the m slots.

  The expected time to search unsuccessfully for a key k is the expected time to search to the end of list $T[h(k)]$, which has expected length $E[n_{h(k)}] = \alpha$.

  Thus, the expected number of elements examined in an unsuccessful search is α, and the total time required (including the time for computing $h(k)$) is $\Theta(1+\alpha)$.

The Successful Search Case

- In a successful search, the probability that a list is searched is proportional to the number of elements it contains.

Theorem

In a hash table in which collisions are resolved by chaining, a successful search takes time $\Theta(1+\alpha)$, on the average, under the assumption of simple uniform hashing.

- We assume that the element being searched for is equally likely to be any of the n elements stored in the table. The number of elements examined during a successful search for an element x is 1 more than the number of elements that appear before x in x's list. Elements before x in the list were all inserted after x was inserted, because new elements are placed at the front of the list. To find the expected number of elements examined, we take the average, over the n elements x in the table, of 1 plus the expected number of elements added to x's list after x was added to the list.

The Successful Search Case (Cont'd)

- Let x_i denote the ith element inserted into the table, $i = 1, 2, \ldots, n$. Let $k_i = x_i.\text{key}$. For keys k_i and k_j, we define the indicator random variable $X_{ij} = \mathrm{I}\{h(k_i) = h(k_j)\}$.
  Under the assumption of simple uniform hashing, we have $\Pr\{h(k_i) = h(k_j)\} = \frac{1}{m}$. So $E[X_{ij}] = \frac{1}{m}$.
  Thus, the expected number of elements examined in a successful search is

$$\begin{aligned}
&E\left[\frac{1}{n}\sum_{i=1}^n(1+\sum_{j=i+1}^n X_{ij})\right] = \frac{1}{n}\sum_{i=1}^n(1+\sum_{j=i+1}^n E[X_{ij}]) \\
&= \frac{1}{n}\sum_{i=1}^n(1+\sum_{j=i+1}^n \frac{1}{m}) = 1 + \frac{1}{nm}\sum_{i=1}^n(n-i) \\
&= 1 + \frac{1}{nm}(\sum_{i=1}^n n - \sum_{i=1}^n i) = 1 + \frac{1}{nm}\left(n^2 - \frac{n(n+1)}{2}\right) \\
&= 1 + \frac{n-1}{2m} = 1 + \frac{n/m}{2} - \frac{n/m}{2n} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}.
\end{aligned}$$

  Thus, the total time required for a successful search (including the time for computing the hash function) is $\Theta(2 + \frac{\alpha}{2} - \frac{\alpha}{2n}) = \Theta(1+\alpha)$.

Subsection 3

Hash Functions

Interpreting Keys as Natural Numbers

- Most hash functions assume that the universe of keys is the set $\mathbb{N} = \{0, 1, 2, \ldots\}$ of natural numbers.
- If the keys are not natural numbers, we find a way to interpret them as natural numbers.
- For example, we can interpret a character string as an integer expressed in suitable radix notation.

  Thus, we might interpret the identifier `pt` as the pair of decimal integers (112, 116), since $\mathtt{p} = 112$ and $\mathtt{t} = 116$ in the ASCII character set.

  Expressed as a radix-128 integer, `pt` becomes $(112 \cdot 128) + 116 = 14452$.
- Since we can usually devise some such method for interpreting each key as a (possibly large) natural number, we assume that the keys are natural numbers.

The Division Method

- In the division method for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m:

$$h(k) = k \mod m.$$

- E.g., if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$.
- This requires only a single division, so it is quite fast.
- We avoid certain values of m: For example, m should not be a power of 2, since, if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k.
- A prime not too close to an exact power of 2 is often a good choice for m.

The Multiplication Method

- The multiplication method operates in two steps:
  - First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA.
  - Then, we multiply this value by m and take the floor of the result.

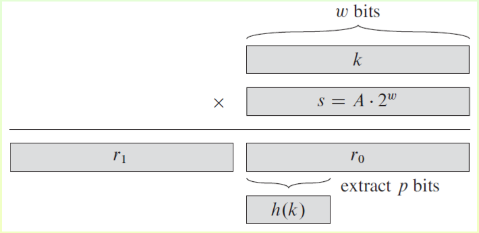
  In short, the hash function is

$$h(k) = \lfloor m(kA \mod 1) \rfloor,$$

  where "$kA \mod 1$" means the fractional part of kA, i.e., $kA - \lfloor kA \rfloor$.
- An advantage of the multiplication method is that the value of m is not critical.
- We typically choose it to be a power of 2 ($m = 2^p$, for some p), since we can then easily implement the function.

Implementation of the Multiplication Method

- Suppose that the word size of a machine is w bits.
- Suppose that k fits into a single word.
- Let A be a fraction of the form $\frac{s}{2^w}$, where s is an integer, $0 < s < 2^w$.



- We first multiply k by the w-bit integer $s = A \cdot 2^w$.
- The result is a 2^w-bit value $r_1 \cdot 2^w + r_0$.
- The p-bit hash value consists of the p most significant bits of r_0.

Universal Hashing

- **Universal hashing** can yield provably good performance on average, no matter which keys occur.
- At the beginning of execution, we select the hash function at random from a carefully designed class of functions.
- Randomly selecting the hash function causes the algorithm to potentially behave differently on each execution, even for the same input, guaranteeing good average-case performance for any input.
- Poor performance may occur only when the the choice of a random hash function causes the set of keys to hash poorly.

  However, he probability of this occurring is small and is the same for any set of keys of the same size.

Setting of Universal Hashing

- Let $\mathcal{H}$ be a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m-1\}$.
- Such a collection is said to be **universal** if, for each pair of distinct keys $k, \ell \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(\ell)$ is at most $\frac{|\mathcal{H}|}{m}$.
- In other words, with a hash function randomly chosen from $\mathcal{H}$, the chance of a collision between distinct keys k and ℓ is no more than the chance $\frac{1}{m}$ of a collision if $h(k)$ and $h(\ell)$ were randomly and independently chosen from the set $\{0, 1, \ldots, m-1\}$.
- We will show that a universal class of hash functions gives good average-case behavior.

Performance of Universal Hashing

Theorem

Suppose that a hash function h is chosen randomly from a universal collection of hash functions and has been used to hash n keys into a table T of size m, using chaining to resolve collisions.

- If key k is not in the table, then the expected length $E[n_{h(k)}]$ of the list that key k hashes to is at most the load factor $\alpha = \frac{n}{m}$.
- If key k is in the table, then the expected length $E[n_{h(k)}]$ of the list containing key k is at most $1 + \alpha$.

- The expectations here are over the choice of the hash function and do not depend on any assumptions about the distribution of the keys.
- For all keys $k \neq \ell$, define the indicator random variable $X_{k\ell} = \mathrm{I}\{h(k) = h(\ell)\}$. By the definition of a universal collection of hash functions, $\Pr\{h(k) = h(\ell)\} \leq \frac{1}{m}$. Therefore, $E[X_{k\ell}] \leq \frac{1}{m}$.

Performance of Universal Hashing (Cont'd)

- We define, for each key k, the random variable Y_k that equals the number of keys other than k that hash to the same slot as k. Then

$$E[Y_k] = E\left[\sum_{\substack{\ell \in T \\ \ell \neq k}} X_{k\ell}\right] = \sum_{\substack{\ell \in T \\ \ell \neq k}} E[X_{k\ell}] \leq \sum_{\substack{\ell \in T \\ \ell \neq k}} \frac{1}{m}.$$

  - If $k \notin T$, then $n_{h(k)} = Y_k$ and $|\{\ell : \ell \in T \text{ and } \ell \neq k\}| = n$. Thus,

$$E[n_{h(k)}] = E[Y_k] \leq \frac{n}{m} = \alpha.$$

  - If $k \in T$, then because key k appears in list $T[h(k)]$ and the count Y_k does not include key k, we have $n_{h(k)} = Y_k + 1$ and $|\{\ell \in T \text{ and } \ell \neq k\}| = n - 1$. Thus,

$$E[n_{h(k)}] = E[Y_k] + 1 \leq \frac{n-1}{m} + 1 = 1 + \alpha - \frac{1}{m} < 1 + \alpha.$$

Expected Time of a Sequence of Operations

- Universal hashing provides the desired payoff: it is impossible for an adversary to pick a sequence of operations that forces the worst-case running time.

Corollary

Using universal hashing and collision resolution by chaining in an initially empty table with m slots, it takes expected time $\Theta(n)$ to handle any sequence of n INSERT, SEARCH and DELETE operations containing $O(m)$ INSERT operations.

- Since the number of insertions is $O(m)$, we have $n = O(m)$. So, $\alpha = O(1)$. The INSERT and DELETE operations take constant time. The expected time for each SEARCH operation is $O(1)$. By linearity of expectation, the expected time for the entire sequence of n operations is $O(n)$. Since each operation takes $\Omega(1)$ time, the $\Theta(n)$ bound follows.

Designing a Universal Class of Hash Functions

- Choose a prime number p, large enough so that every possible key k is in the range 0 to $p-1$. Let $\mathbb{Z}_p$ denote the set $\{0, 1, \ldots, p-1\}$ and let $\mathbb{Z}_p^*$ denote the set $\{1, 2, \ldots, p-1\}$.
  - Since p is prime, we can solve equations modulo p.
  - The size of the universe of keys is greater than the number of slots in the hash table, whence $p > m$.
- Define the hash function h_{ab} for any $a \in \mathbb{Z}_p^*$ and any $b \in \mathbb{Z}_p$ using a linear transformation followed by reductions modulo p and then modulo m:

$$h_{ab}(k) = ((ak + b) \mod p) \mod m.$$

- The family of hash functions is $\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\}$.
- Each hash function h_{ab} maps $\mathbb{Z}_p$ to $\mathbb{Z}_m$.
- Since we have $p-1$ choices for a and p choices for b, the collection $\mathcal{H}_{pm}$ contains $p(p-1)$ hash functions.

Universality of $\mathcal{H}_{pm}$

Theorem (Universality of $\mathcal{H}_{pm}$)

The class $\mathcal{H}_{pm}$ of hash functions is universal.

- Consider two distinct keys k and ℓ from $\mathbb{Z}_p$, so that $k \neq \ell$. For a given h_{ab}, we let $r = (ak+b) \mod p$ and $s = (a\ell + b) \mod p$.
  Claim: $r \neq s$.
  Since $r - s \equiv a(k-\ell) \mod p$, p is prime and both a and $k-\ell$ are nonzero modulo p, their product must also be nonzero modulo p. Therefore, when computing any $h_{ab} \in \mathcal{H}_{pm}$, distinct inputs k and ℓ map to distinct values r and s modulo p. I.e., there are no collisions yet at the "mod p level". Each of the possible $p(p-1)$ choices for the pair (a,b), with $a \neq 0$, yields a different resulting pair (r,s), with $r \neq s$, since we can solve for a and b, given r and s: Indeed

$$\begin{aligned} a &= ((r-s)((k-\ell)^{-1} \mod p)) \mod p \\ b &= (r-ak) \mod p. \end{aligned}$$

Universality of $\mathcal{H}_{pm}$ (Cont'd)

- Since there are only $p(p-1)$ possible pairs (r,s), with $r \neq s$, there is a one-to-one correspondence between pairs (a,b), with $a \neq 0$, and pairs (r,s), with $r \neq s$.

  Thus, for any given pair of inputs k and ℓ, if we pick (a,b) uniformly at random from $\mathbb{Z}_p^* \times \mathbb{Z}_p$, the resulting pair (r,s) is equally likely to be any pair of distinct values modulo p.

  Therefore, the probability that distinct keys k and ℓ collide is equal to the probability that $r \equiv s \pmod{m}$, when r and s are randomly chosen as distinct values modulo p.

  For a given value of r, of the $p-1$ possible remaining values for s, the number of values s such that $s \neq r$ and $s \equiv r \pmod{m}$ is at most $\lceil \frac{p}{m} \rceil - 1 \leq \frac{p+m-1}{m} - 1 = \frac{p-1}{m}$. The probability that s collides with r when reduced modulo m is at most $\frac{\frac{p-1}{m}}{p-1} = \frac{1}{m}$. Therefore, for any pair of distinct values $k, \ell \in \mathbb{Z}_p$, $\Pr\{h_{ab}(k) = h_{ab}(\ell)\} \leq \frac{1}{m}$.

Subsection 4

Open Addressing

Idea Behind Open Addressing

- In **open addressing**, all elements occupy the hash table itself, i.e., each table entry contains either an element of the dynamic set or NIL.
- When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table.
- No lists and no elements are stored outside the table.
- In open addressing, the hash table can "fill up" so that no further insertions can be made, whence the load factor α can never exceed 1.
- Of course, we could store the linked lists for chaining inside the hash table, in the otherwise unused hash-table slots, but the advantage of open addressing is that it avoids pointers altogether.
  - Instead of following pointers, we compute the sequence of slots to be examined.
  - Avoiding pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval.

Idea Behind Insertion

- To perform insertion using open addressing, we successively examine, or probe, the hash table until we find an empty slot in which to put the key.
- Instead of being fixed in the order $0, 1, \ldots, m-1$ ($\Theta(n)$ search time), the sequence of positions probed depends upon the key being inserted.
- To determine which slots to probe, we extend the hash function to include the probe number (starting from 0) as a second input.

  Thus, the hash function becomes

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

- With open addressing, we require that for every key k, the probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ be a permutation of $\langle 0, 1, \ldots, m-1\rangle$, so that every hash-table position is eventually considered as a slot for a new key as the table fills up.

The Hash Insert Procedure

- Suppose the elements in the hash table T are keys with no satellite information and key k is identical to the element containing key k.
- Each slot contains either a key or NIL (if the slot is empty).
- The input is a hash table T and a key k.
- The output is the slot number where k is stored or an "overflow".

HashInsert(T, k)

```
1. $i = 0$
2. repeat
3.    $j = h(k, i)$
4.    if $T[j] ==$ NIL
5.       $T[j] = k$
6.       return $j$
7.    else $i = i + 1$
8. until $i == m$
9. error "hash table overflow"
```

The Hash Search Procedure

- The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted.
- The search can terminate (unsuccessfully) when it finds an empty slot, since k would have been inserted there and not later in its probe sequence (assuming that keys are not deleted from the table).
- HASHSEARCH takes as input a hash table T and a key k.
- It returns j if slot j contains key k, or NIL if key k is not in T.

HASHSEARCH(T, k)

```
1. $i = 0$
2. repeat
3.    $j = h(k, i)$
4.    if $T[j] == k$
5.       return $j$
6.    $i = i + 1$
7. until $T[j] ==$ NIL or $i == m$
8. return NIL
```

Deletion in Open Addressing

- Deletion from an open-address hash table is difficult:

  When we delete a key from slot *i*, we cannot simply mark that slot as empty by storing NIL in it.

  If we did, we might be unable to retrieve any key *k* during whose insertion we had probed slot *i* and found it occupied.
- We can solve this problem by marking the slot, storing in it the special value DELETED instead of NIL.
- We would then modify the procedure HASHINSERT to treat such a slot as if it were empty so that we can insert a new key there.
- We do not need to modify HASHSEARCH, since it will pass over DELETED values while searching.
- When we use DELETED, search times no longer depend on the load factor α, and for this reason chaining is more commonly selected as a collision resolution technique when keys must be deleted.

Techniques for Approximating Uniform Hashing

- We assume uniform hashing: the probe sequence of each key is equally likely to be any of the m permutations of $\langle 0, 1, \ldots, m-1 \rangle$.
- Uniform hashing generalizes the notion of simple uniform hashing defined earlier to a hash function that produces not just a single number, but a whole probe sequence.
- We examine three techniques to compute the probe sequences for open addressing:
  - linear probing;
  - quadratic probing;
  - double hashing.

  These techniques all guarantee that $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$, for each key k.
- None of these techniques fulfills the assumption of uniform hashing, since none of them is capable of generating more than m^2 different probe sequences (uniform hashing requires $m!$).
- Double hashing outputs the greatest number of probe sequences.

Linear Probing

- Given an ordinary hash function $h' : U \to \{0, 1, \dots, m-1\}$, which we refer to as an **auxiliary hash function**, the method of **linear probing** uses the hash function

$$h(k,i) = (h'(k) + i) \mod m, \quad i = 0, 1, \dots, m-1.$$

- Given key k:
  - We probe $T[h'(k)], T[h'(k)+1], \dots, T[m-1]$.
  - Then we wrap around to slots $T[0], T[1], \dots, T[h'(k)-1]$.
- Because the initial probe determines the entire probe sequence, there are only m distinct probe sequences.
- Linear probing is easy to implement, but it suffers from a problem known as primary clustering: Long runs of occupied slots build up, increasing the average search time.

  Clusters arise because an empty slot preceded by i full slots gets filled next with probability $\frac{i+1}{m}$. Long runs of occupied slots tend to get longer, and the average search time increases.

Quadratic Probing

- **Quadratic probing** uses a hash function of the form

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m,$$

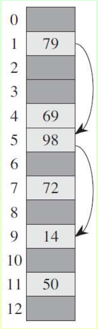
  where h' is an **auxiliary hash function**, c_1 and c_2 are positive auxiliary constants, and $i = 0, 1, \ldots, m-1$.
- The initial position probed is $T[h'(k)]$. Later positions probed are offset by amounts depending quadratically on i.
- This method works much better than linear probing, but to make full use of the hash table, the values of c_1, c_2 and m are constrained.
- If two keys have the same initial probe position, their probe sequences are the same, since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$.
- This property leads to a milder form of clustering, called **secondary clustering**.
- As in linear probing, the initial probe determines the entire sequence, and so only m distinct probe sequences are used.

Double Hashing

- **Double hashing** uses a hash function of the form

$$h(k,i) = (h_1(k) + ih_2(k)) \mod m,$$

where both h_1 and h_2 are auxiliary hash functions.
- The initial probe goes to position $T[h_1(k)]$.
Successive probe positions are offset from previous positions by the amount $h_2(k)$, modulo m. Unlike the case of linear or quadratic probing, the probe sequence depends in two ways upon the key k, since the initial probe position, the offset, or both, may vary.



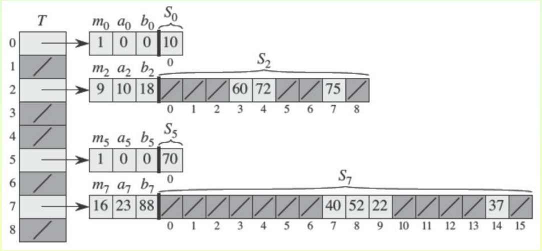
- The value $h_2(k)$ must be relatively prime to the hash-table size m for the entire hash table to be searched.
  - A convenient way to ensure this condition is to let m be a power of 2 and to design h_2 so that it always produces an odd number.
  - Another way is to let m be prime and to design h_2 so that it always returns a positive integer less than m.

Subsection 5

Perfect Hashing

Perfect Hashing

- We call a hashing technique **perfect hashing** if the worst-case number of memory accesses required to perform a search is $O(1)$.
- The basic idea to create a perfect hashing scheme is simple: We use a two-level hashing scheme with universal hashing at each level.



  - In the first level the n keys are hashed into m slots using a hash function h carefully selected from a family of universal hash functions.
  - Instead of making a list of the keys hashing to slot j, we use a small secondary hash table S_j with an associated hash function h_j. By choosing the h_j's carefully, we can avoid collisions at the second level.

Second Level Hashing

- To guarantee that there are no collisions at the secondary level we need to let the size m_j of hash table S_j be the square of the number n_j of keys hashing to slot j.
  - This quadratic dependence of m_j on n_j may seem to require excessive storage, but we show that a good choice of the first level hash function keeps the expected total amount of space used at $\mathrm{O}(n)$.
- We use hash functions chosen from the universal classes of hash functions $\mathcal{H}_{p,m}$, where p is a prime greater than any key value.
- Those keys hashing to slot j are re-hashed into a secondary hash table S_j of size m_j using a hash function h_j chosen from the class $\mathcal{H}_{p,m_j}$.
  - First, we determine how to ensure that the secondary tables have no collisions.
  - Second, we show that the expected amount of memory used overall - for the primary hash table and all the secondary hash tables - is $\mathrm{O}(n)$.

Secondary Hashing Without Collisions

Theorem

Suppose that we store n keys in a hash table of size $m = n^2$ using a hash function h randomly chosen from a universal class of hash functions. Then, the probability is less than $\frac{1}{2}$ that there are any collisions.

- There are $\binom{n}{2}$ pairs of keys that may collide. Each pair collides with probability $\frac{1}{m}$ if h is chosen at random from a universal family $\mathcal{H}$ of hash functions. Let X be a random variable that counts the number of collisions. When $m = n^2$, the expected number of collisions is $E[X] = \binom{n}{2}\frac{1}{n^2} = \frac{n^2-n}{2}\frac{1}{n^2} < \frac{1}{2}$. Applying Markov's inequality, $\Pr\{X \geq t\} \leq \frac{E[X]}{t}$. Taking $t = 1$, completes the proof.
- Given a static set K of n keys to be hashed, it is easy to find a collision-free hash function h with a few random trials.
- To deal with excessive table size $m = n^2$, we use the theorem only to hash the entries at the second level (within each slot).

Towards Achieving Linearity of Space

- We now show that the overall memory used is $O(n)$.
- Since the size m_j of the jth secondary hash table grows quadratically with the number n_j of keys stored, we run the risk that the overall amount of storage could be excessive.
- If the first-level table size is $m = n$, then the amount of memory used is $O(n)$ for
  - the primary hash table,
  - the storage of the sizes m_j of the secondary hash tables,
  - the storage of the parameters a_j and b_j defining the secondary hash functions h_j drawn from the class $\mathcal{H}_{p,m_j}$.
- The following results provide a bound on:
  - the expected combined sizes of all the secondary hash tables;
  - the probability that the combined size of all the secondary hash tables is superlinear (that it equals or exceeds $4n$).

Expected Combined Size of Secondary Hash Tables

Theorem

Suppose that we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions. Then, we have $E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n$, where n_j is the number of keys hashing to slot j.

- We start with $a^2 = a + 2\binom{a}{2}$. We have $E\left[\sum_{j=0}^{m-1} n_j^2\right] = E\left[\sum_{j=0}^{m-1} \left(n_j + 2\binom{n_j}{2}\right)\right] = E\left[\sum_{j=0}^{m-1} n_j\right] + 2E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] = E[n] + 2E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right] = n + 2E\left[\sum_{j=0}^{m-1} \binom{n_j}{2}\right]$. The sum $\sum_{j=0}^{m-1} \binom{n_j}{2}$ is just the total number of pairs of keys that collide. By the properties of universal hashing, the expected value of this summation is at most $\binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m} = \frac{n-1}{2}$. Thus, $E\left[\sum_{j=0}^{m-1} n_j^2\right] \leq n + 2\frac{n-1}{2} = 2n - 1 < 2n$.

Expected Space for Secondary Hash Tables

Corollary

Suppose that we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions, and we set the size of each secondary hash table to $m_j = n_j^2$, for $j = 0, 1, \ldots, m-1$. Then, the expected amount of storage required for all secondary hash tables in a perfect hashing scheme is less than $2n$.

- Since $m_j = n_j^2$, for $j = 0, 1, \ldots, m-1$, the theorem gives

$$E\left[\sum_{j=0}^{m-1} m_j\right] = E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n.$$

This completes the proof.

Probability of Large Secondary Hash Table Storage Space

Corollary

Suppose that we store n keys in a hash table of size $m = n$ using a hash function h randomly chosen from a universal class of hash functions, and we set the size of each secondary hash table to $m_j = n_j^2$, for $j = 0, 1, \ldots, m-1$. Then, the probability is less than $\frac{1}{2}$ that the total storage used for secondary hash tables equals or exceeds $4n$.

- We apply Markov's inequality $\Pr\{X \geq t\} \leq \frac{E[X]}{t}$, with $X = \sum_{j=0}^{m-1} m_j$ and $t = 4n$.

  Recalling that $E\left[\sum_{j=0}^{m-1} m_j\right] < 2n$, we get

$$\Pr\left\{\sum_{j=0}^{m-1} m_j \geq 4n\right\} \leq \frac{E\left[\sum_{j=0}^{m-1} m_j\right]}{4n} < \frac{2n}{4n} = \frac{1}{2}.$$