Chap 11 – Hash Tables

11.1 Direct-address tables

11.2 Hash tables

11.3 Hash functions

11.4 Open addressing

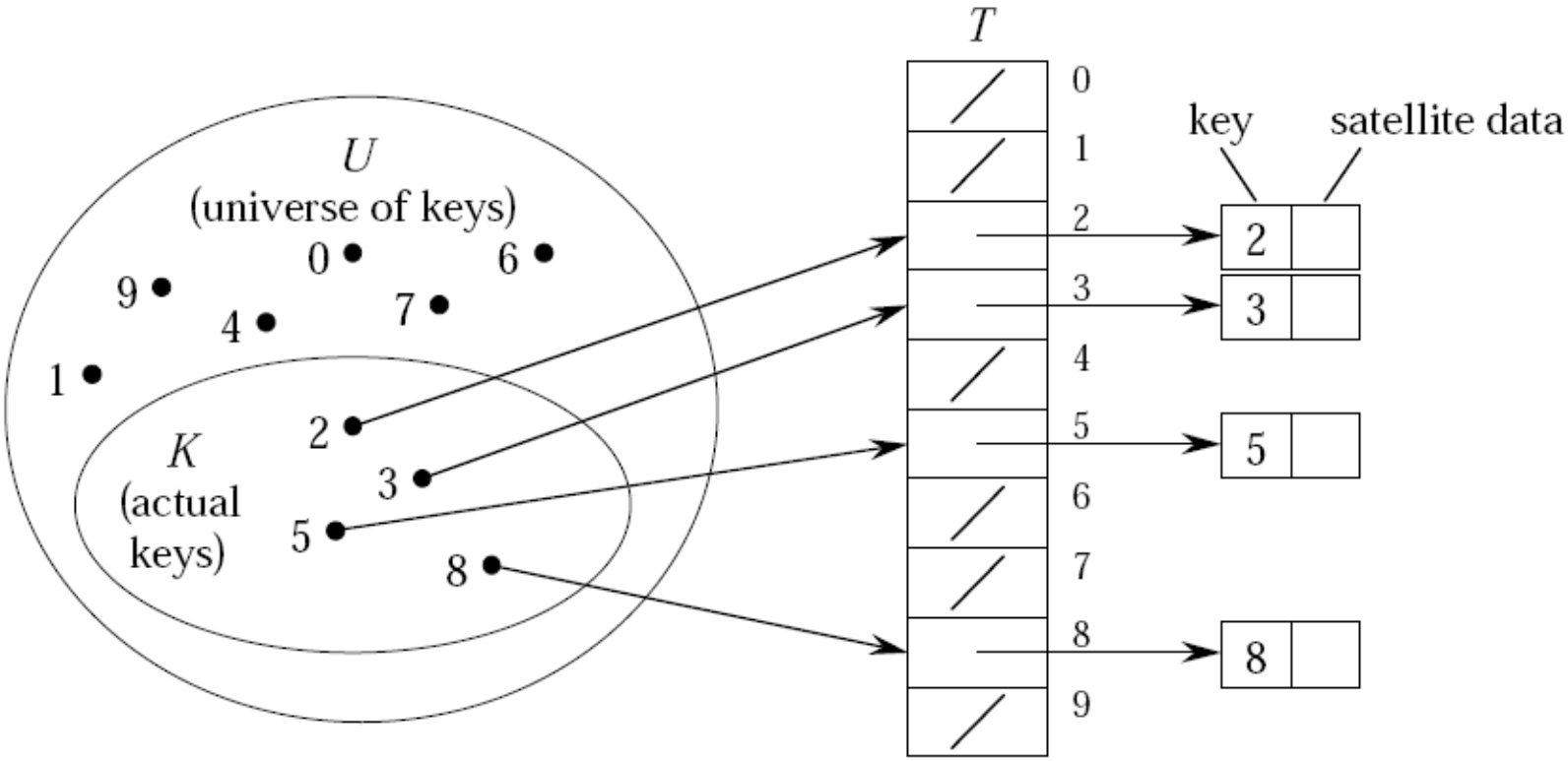
11.5 Perfect hashing

11.1 Direct-address tables

- *Scenario*
  - Maintain a dynamic set.
  - Each element has a key drawn from a universe $U = \{0,1,2, \ldots, m-1\}$ where m isn't too large.
  - No two elements have the same key.
- Direct-address tables
  - A direct-address table is an array $T[0..m-1]$
  - Store an element with key k in slot k

11.1 Direct-address tables

- Direct-address tables
  - Example



11.1 Direct-address tables

- Direct-address tables
  - Dictionary operations: Search, Insert, and Delete
  - DIRECT-ADDRESS-SEARCH(T, k)

    return $T[k]$
  - DIRECT-ADDRESS-INSERT(T, x)

    $T[x.key] = x$
  - DIRECT-ADDRESS-DELETE(T, x)

    $T[x.key] =$ NIL

11.1 Direct-address tables

- Direct-address tables
  - Pro

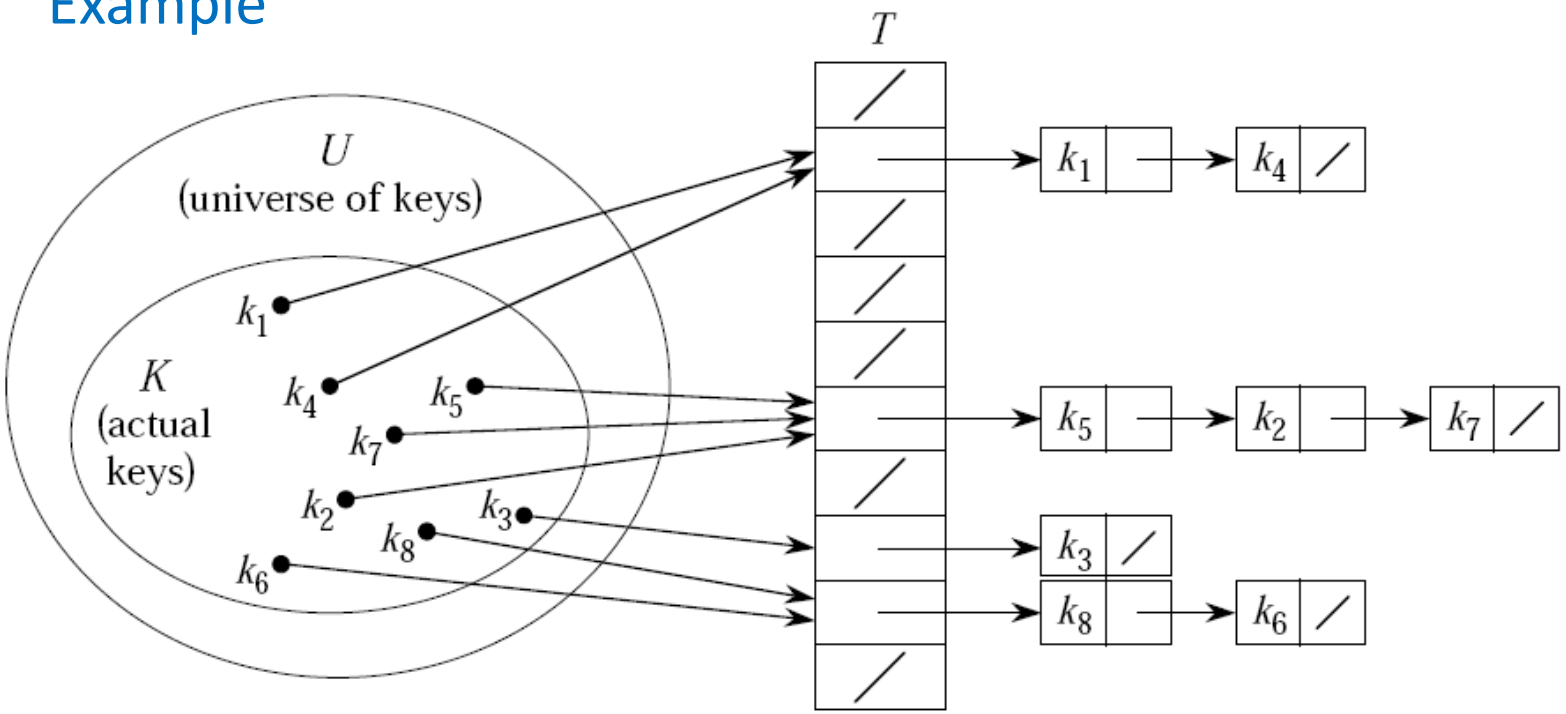
    Dictionary operations are trivial and take $O(1)$ time each.
  - Con
    - If the universe U is large, storing a table of size $|U|$ may be impractical or impossible.
    - Often, the set of keys actually stored is small, compared with U, so that most space allocate for T is waste.

11.2 Hash tables

- Hash tables
  - Let U be the universe and $T[0..m-1]$ be an array.
    Assume that $|U| > m$
  - Let $h: U \to \{0,1,2,\ldots,m-1\}$ be a hash function
  - Store an element with key k in slot $h(k)$.
  - *Collision*
    When two or more keys hash to the same slot.
  - *Collision resolution*
    Use two methods: chaining and open addressing.
  - Chaining is usually better than open addressing.
    We'll examine both.

11.2 Hash tables

- Hash tables with chaining
  - Example



This figure shows singly linked lists. If we want to delete elements, it's better to use doubly linked lists.

11.2 Hash tables

- Hash tables with chaining

  Dictionary operations

  - CHAINED-HASH-INSERT(T, x)

    insert x at the head of list $T[h(x.key)]$

    - Worst-case running time $= O(1)$

  - CHAINED-HASH-DELETE(T, x)

    delete x from the list $T[h(x.key)]$

    - Suppose x is a pointer to the element to delete

      For doubly-linked list, worst-case running time $= O(1)$

      For singly-linked list, deletion = searching

11.2 Hash tables

- Hash tables with chaining
  - CHAINED-HASH-SEARCH(T, k)

    search for an element with key k in list $T[h(k)]$
    - Running time $= O$(length of list in slot $h(k)$)
    - Worst-case running time

      Worst case is when all n keys hash to the same slot

      $\Rightarrow$ get a single list of length n

      $\Rightarrow$ worst-case running time to search is $\Theta(n)$
    - Average-case running time

      See below

11.2 Hash tables

- Average-case analysis of CHAINED-HASH-SEARCH
  - Analysis is in terms of the ***load factor*** $\alpha = n/m$
    - $n =$ # of elements in the table
    - $m =$ # of slots in the table
      $=$ # of (possibly empty) linked lists
    - Load factor is average # of elements per linked list.
    - Can have $\alpha < 1$, $\alpha = 1$, or $\alpha > 1$
  - Assume ***simple uniform hashing***: Any given element is equally likely to hash into any of the m slots.
    - Let n_i be the random variable denoting the length of list $T[i]$. Then, $\mathrm{E}[n_i] = n/m = \alpha$

11.2 Hash tables

- Average-case analysis of CHAINED-HASH-SEARCH
  - **THEOREM**

    An unsuccessful search takes expected time $\Theta(1+\alpha)$

    Proof

    Simple uniform hashing

    ⇒ Key not in the table is equally likely to hash to any slot

    Key k not in the table

    ⇒ Have to search the entire list $T[h(k)]$ of the expected length $\mathrm{E}[n_{h(k)}] = \alpha$

    Constant-time hashing

    ⇒ the expected total time is $\Theta(1+\alpha)$

11.2 Hash tables

- Average-case analysis of CHAINED-HASH-SEARCH
  - **THEOREM**

    A successful search takes expected time $\Theta(1+\alpha)$.

    Proof

    Let x_i be the i^{th} element inserted into the table

    Then,

    the # of elements examined in a successful search for x_i

    $= 1 +$ # of elements appear before x_i in x_i's list

    Note that if x_j appears before x_i then $j > i$

    Assume that the element being searched for is equally likely to be any of the n elements stored in the table.

11.2 Hash tables

- Average-case analysis of CHAINED-HASH-SEARCH
  - **THEOREM** (Cont'd)

    So, averaging over the n elements in the table, the # of elements examined in a successful search is

    $$\frac{1}{n}\sum_{i=1}^{n}(1 + \#\text{ of elements appear before } x_i \text{ in } x_i\text{'s list})$$

    It remains to find its expected value, or equivalently, the **expected** # of elements appear before x_i in x_i's list.

    Let $k_i = x_i.key$

    Define indicator random variables

    $$X_{ij} = \mathrm{I}\{h(k_i) = h(k_j)\}, \quad i \neq j$$

11.2 Hash tables

- Average-case analysis of CHAINED-HASH-SEARCH
  - **THEOREM** (Cont'd)

    Simple uniform hashing

    $\Rightarrow \Pr\{h(k_i) = h(k_j)\} = m/m^2 = 1/m$

    $\Rightarrow \mathrm{E}[X_{ij}] = 1/m$

    Thus, the expected # of elements examined in a successful search is

$$\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}]\right)$$

$\because$ linearity of expectation

11.2 Hash tables

- Average-case analysis of CHAINED-HASH-SEARCH
  - **THEOREM** (Cont'd)

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}]\right)=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

$$=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)=1+\frac{1}{nm}\sum_{k=1}^{n-1}k$$

$$=1+\frac{n-1}{2m}=1+\frac{n}{2m}-\frac{n}{2nm}$$

$$=1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

11.2 Hash tables

- Average-case analysis of CHAINED-HASH-SEARCH
  - **THEOREM** (Cont'd)

    Adding 1 for constant-time hashing, the expected total time for a successful search is

    $$\Theta\left(2+\frac{\alpha}{2}-\frac{\alpha}{2n}\right)=\Theta(1+\alpha)$$

  - Conclusion

    If $n=O(m)$, then $\alpha=n/m=O(m)/m=O(1)$

    which means that searching takes $O(1)$ time on average

    Thus, all dictionary operations take $O(1)$ time on average.

11.3 Hash functions

- What makes a good hash function?
  - Ideally, simple uniform hashing
  - In practice, don't know the probability distribution
  - Often use heuristics, based on the domain of the keys
- Keys as natural numbers
  - Hash functions assume that the keys are natural numbers.
  - When they're not, must interpret them as natural numbers
  - Example

    Since there are 128 basic ASCII values, a string may be interpreted as a radix-128 integer.

11.3 Hash functions

- Keys as natural numbers
  - Example (Cont'd)

    E.g. ASCII values: C = 67, A = 65, T = 84.

    So, interprete the string CAT as

    $67 \cdot 128^2 + 65 \cdot 128^1 + 84 \cdot 128^0 = 1106132$

    Note: (Ex. 11.3-3)

    Consider the string ACT that is a permutation of CAT

    Then,

    $$\text{CAT} - \text{ACT} = (67 - 65) \cdot 128^2 + (65 - 67) \cdot 128^1$$
    $$= 2 \cdot 128 \cdot (128 - 1)$$
    $$\Rightarrow (\text{CAT} - \text{ACT}) \bmod (128 - 1) = 0$$

11.3 Hash functions

- Division method
  - $h(k) = k \bmod m$
  - Pro: Fast, since requires just one division operation.
  - Con: Have to avoid certain values of m
    - Powers of 2 are bad. If $m = 2^p$ for integer p, then $h(k)$ is just the least significant p bits of k.
    - If k is a character string interpreted in radix 2^p, then $m = 2^p - 1$ is bad: permuting characters in a string does not change its hash value (See previous example)
  - ***Good choice for m***

    A prime not too close to an exact power of 2.

11.3 Hash functions

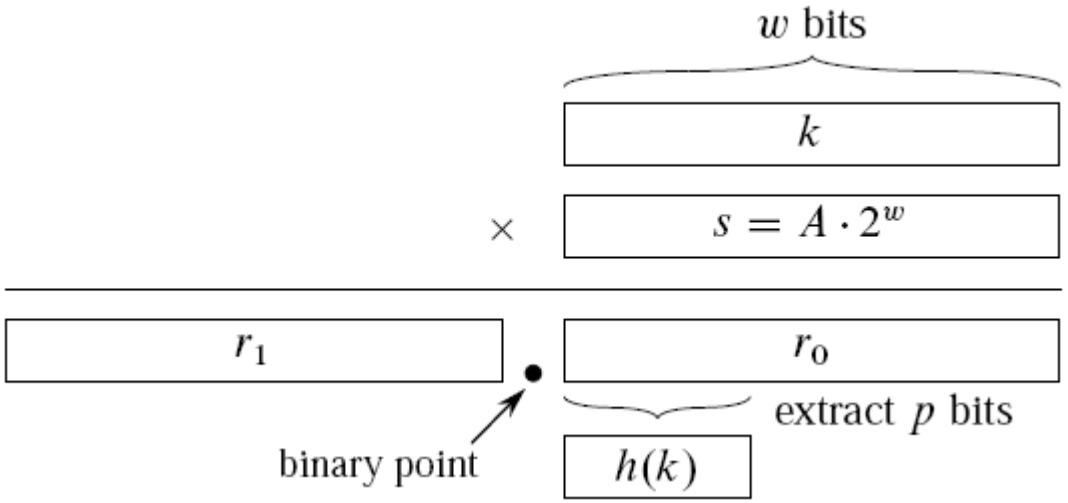
- Multiplication method
  - $h(k) = \lfloor m(kA \bmod 1) \rfloor, 0 < A < 1$
    where $kA \bmod 1 = kA - \lfloor kA \rfloor$ = fractional part of kA
  - Con: Slower than division method.
  - Pro: Value of m is not critical.
  - *(Relatively) easy implementation*
    - Assume that the keys occupy w bits
    - Choose $m = 2^p$ for some integer p
    - Choose $A = s/2^w$ for some integer s, $0 < s < 2^w$
    - Let $ks = r_1 2^w + r_0$
      $\Rightarrow kA = ks/2^w = r_1 + r_0 2^{-w} = r_1.r_0$ (in radix 2^w)

11.3 Hash functions

- Multiplication method
  - *(Relatively) easy implementation* (Cont'd)
    - $kA = r_1.r_0$

      $\Rightarrow 0.r_0 = kA - \lfloor kA \rfloor$

      $\Rightarrow h(k) = \lfloor 2^p(kA - \lfloor kA \rfloor) \rfloor$ = the leftmost p bits of r_0



11.3 Hash functions

- Multiplication method
  - Example

    Let $w = 5, k = 17$

    Choose $m = 8 \Rightarrow p = 3$

    Choose $A = 20/32 = 20/2^5 \Rightarrow s = 20 < 2^5$

    By formula

    $$kA = 17 \cdot \frac{20}{32} = 10\frac{20}{32} \Rightarrow h(k) = \left\lfloor 8 \cdot \left(10\frac{20}{32} \bmod 1 \right) \right\rfloor = 5$$

    By (relatively) easy implementation

    $$ks = 17 \cdot 20 = 340 = 10 \cdot 2^5 + 20$$

    $$\Rightarrow r_0 = 20 = \underline{10}100 \Rightarrow h(17) = 5$$

11.3 Hash functions

- Multiplication method
  - How to choose A?

    In The Art of Computer Programming, Vol. 3, Knuth suggests

    $$A \approx \frac{\sqrt{5}-1}{2} = 0.618\ldots$$

    In the preceding example,

    $$A = 20/32 = 0.625$$

11.3 Hash functions

- Universal hashing
  - For any choice of fixed hash function, there exists a bad set of keys that collide together.
  - Idea

    Randomize – choose a hash function at random from a "well-designed" set of hash functions on each run
  - **DEFINITION**

    A set of hash functions

    $\mathcal{H} = \{h \mid h: U \to \{0,1,\ldots,m-1\}\}$

    is universal, if

    $|\{h \in \mathcal{H} \mid h(x) = h(y)\}| \leq |\mathcal{H}|/m, \forall x \neq y$

11.3 Hash functions

- Universal hashing
  - Let h be a hash function chosen at random from $\mathcal{H}$, then $\Pr\{h(x) = h(y)\} \leq 1/m, \forall x \neq y$
  - **THEOREM**

    Using chaining and universal hashing on key k,

    k is not in the table $T \Rightarrow \mathrm{E}[n_{h(k)}] \leq \alpha$

    k is in the table $T \Rightarrow \mathrm{E}[n_{h(k)}] \leq 1 + \alpha$

    Proof

    Let $Z_{kl} = \mathrm{I}\{h(k) = h(l)\} \quad k \neq l$

    Then,

    $\mathrm{E}[Z_{kl}] = \Pr\{h(k) = h(l)\} \leq 1/m$

11.3 Hash functions

- Universal hashing
  - **THEOREM** (Cont'd)

$$k \notin T \Rightarrow n_{h(k)} = \sum_{l \in T} Z_{kl}$$

$$\Rightarrow \mathrm{E}[n_{h(k)}] = \mathrm{E}\left[\sum_{l \in T} Z_{kl}\right] = \sum_{l \in T} \mathrm{E}[Z_{kl}] \leq n/m = \alpha$$

$$k \in T \Rightarrow n_{h(k)} = 1 + \sum_{l \in T, l \neq k} Z_{kl}$$

$$\Rightarrow \mathrm{E}[n_{h(k)}] = 1 + \sum_{l \in T, l \neq k} \mathrm{E}[Z_{kl}]$$

$$\leq 1 + (n-1)/m < 1 + \alpha$$

11.3 Hash functions

- Universal hashing
  - **COROLLARY**

    Using chaining and universal hashing

    A (successful or unsuccessful) search takes expected time $O(1+\alpha)$.

    Proof

    Adding 1 for constant-time hashing

    Expected total time for an unsuccessful search $\leq 1+\alpha$

    Expected total time for a successful search $\leq 2+\alpha$

11.3 Hash functions

- Universal hashing

  Design a universal class of hash functions

  - Pick up a prime p such that $0 \leq k \leq p-1 \; \forall k \in U$
  - m is arbitrary as long as $n = O(m)$
  - Since we assume $|U| > m$, we have $p \geq |U| > m$
  - Let $\mathbb{Z}_p = \{0,1,\ldots,p-1\}$, $\mathbb{Z}_p^* = \{1,2,\ldots,p-1\}$

    N.B. $U \subseteq \mathbb{Z}_p$

    Define

    $h_{a,b} : \mathbb{Z}_p \rightarrow \mathbb{Z}_m \quad \forall a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p$

    by

    $h_{a,b}(k) = ((ak+b) \bmod p) \bmod m$

11.3 Hash functions

- Universal hashing

  Design a universal class of hash functions

  - Define

    $\mathcal{H}_{p,m} = \{h_{a,b} | a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$

    Note that $|\mathcal{H}_{p,m}| = p(p-1)$

  - **THEOREM**

    $\mathcal{H}_{p,m}$ is universal.

11.4 Open addressing

- Open addressing
  - Store all keys in the hash table itself.
  - Each slot contains either a key or NIL.
  - Examining a slot is known as a ***probe***.
  - To search for key k:
    - If slot $h(k)$ contains k, the search is successful.
    - If slot $h(k)$ contains NIL, the search is unsuccessful.
    - If slot $h(k)$ contains a key $\neq k$, probe the "next" slot
    - Keep probing until we find a slot containing k or NIL
  - To insert, act as though we're searching, and insert at the first NIL slot we find.

11.4 Open addressing

- Open addressing
  - Hash function

    $$h : U \times \underbrace{\{0,1,\ldots,m-1\}}_{\text{probe number}} \rightarrow \underbrace{\{0,1,\ldots,m-1\}}_{\text{slot number}}$$

  - For every key k, the probe sequence

    $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$

    shall be a permutation of $\langle 0,1,\ldots,m-1 \rangle$

11.4 Open addressing

- Open addressing
  - HASH-SEARCH and HASH-INSERT

HASH-SEARCH(T, k)

$i = 0$

repeat

  $j = h(k, i)$

  if $T[j] == k$

    return j

  $i = i + 1$

until $T[j] == \text{NIL}$ or $i == m$

return NIL

HASH-INSERT(T, k)

$i = 0$

repeat

  $j = h(k, i)$

  if $T[j] == \text{NIL}$

    $T[j] = k$; **return**

  $i = i + 1$

until $i == m$

error "hash table overflow"

11.4 Open addressing

- Open addressing

  How to handle HASH-DELETE(T, k)?

  - We can't just put NIL into the slot containing key k.

    ∵ Subsequent search for key k' would be unsuccessful, if k' were inserted after k and probed the slot occupied by k during its insertion.
  - Solution: Use a special value **DELETED** instead of NIL
    - Search treats **DELETED** as if the slot holds a key that does not match the one being searched for.
    - Insertion treats **DELETED** as if the slot were empty and so can be reused.
  - Con: Search time no longer depends on the load factor α

11.4 Open addressing

- How to compute probe sequences
  - Ideal situation

    Uniform hashing – Each key is equally likely to have any of the $m!$ permutations of $\langle 0,1,\ldots,m-1\rangle$ as its probe sequence.
  - This generalizes ***simple uniform hashing*** – Each key is equally likely to have any of the m slots to hash to.
  - It's hard to implement true uniform hashing.
  - Approximation

    Use hash functions that at least guarantee that the probe sequence is a permutation of $\langle 0,1,\ldots,m-1\rangle$

11.4 Open addressing

- Linear probing
  - $h(k,i) = (h'(k) + i) \bmod m$

    where h' is an auxiliary hash function
  - Probe sequence

    $\langle h'(k), h'(k)+1, \ldots, m-1, 0, 1, \ldots, h'(k)-1 \rangle$
  - The initial probe $h'(k)$ determines the entire sequence

    $\Rightarrow$ only m possible sequences
  - Suffer from ***primary clustering***

    Long runs of occupied sequences build up

    $\because \Pr\{\text{occupied seq. of length } i \Rightarrow \text{length } i+1\} = \frac{i+2}{m}$

    Thus, the average search and insertion times increase.

11.4 Open addressing

- Quadratic probing
  - $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
  - To get a permutation of $\langle 0,1,\ldots,m-1 \rangle$, the constants c_1, c_2 and m must be chosen properly (Problem 11-3).
  - Again, initial probe $h'(k)$ determines the entire sequence
    $\Rightarrow$ only m possible sequences
  - Can get ***secondary clustering***
    Keys with the same initial probe position have the same probe sequence.

11.4 Open addressing

- Double probing
  - $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$
  - Use two auxiliary hash functions h_1 and h_2
  - To get a permutation of $\langle 0,1,\ldots,m-1\rangle$, the value $h_2(k)$ must be relatively prime to m (Exercise 11.4-4).
    - Could let m be a prime and $1 \le h_2(k) < m$
    - Could let m be a power of 2 and $h_2(k)$ be odd
  - Each pair $\langle h_1(k), h_1(k)\rangle$ yields a distinct probe sequence $\Rightarrow \Theta(m^2)$ possible probe sequences

11.4 Open addressing

- Analysis of open-address hashing
  - Assumptions
    - Assume that the table never completely fills
      i.e. $0 \leq n < m \Rightarrow 0 \leq \alpha < 1$
    - Assume uniform hashing
    - No deletion
  - **THEOREM**
    The expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.
    Proof
    Define the random variable
    X = #of probes made in an unsuccessful search

11.4 Open addressing

- Analysis of open-address hashing
  - **THEOREM** (Cont'd)

$$\mathrm{E}[X] = \sum_{i=1}^{n+1} i\Pr\{X = i\}$$

$$= \sum_{i=1}^{\infty} i\Pr\{X = i\} \quad \because \Pr\{X = j\} = 0 \quad \forall j > n+1$$

$$= \sum_{i=1}^{\infty} i(\Pr\{X \geq i\} - \Pr\{X \geq i+1\})$$

$$= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \quad \because \begin{matrix} 1(\geq 1 - \geq 2) \\ 2(\geq 2 - \geq 3) \\ 3(\geq 3 - \geq 4) \end{matrix}$$

11.4 Open addressing

- Analysis of open-address hashing
  - **THEOREM** (Cont'd)

    Let A_i = the event that the i^{th} probe is to an occupied slot

    $$\Pr\{X \geq i\}$$
    $$= \Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$$
    $$= \Pr\{A_1\} \cdot \Pr\{A_2|A_1\} \cdot \cdots \Pr\{A_{i-1}|A_1 \cap A_2 \cap \cdots \cap A_{i-2}\}$$
    $$= \underbrace{\frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \cdots \cdot \frac{n-i+2}{m-i+2}}_{i-1 \text{ factors}} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$

    CLAIM

    $$\Pr\{A_j | A_1 \cap A_2 \cap \cdots \cap A_{j-1}\} = (n-j+1)/(m-j+1)$$

11.4 Open addressing

- Analysis of open-address hashing
  - **THEOREM** (Cont'd)

    Therefore,

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\} \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha}$$

    Finally, the **CLAIM**:

$$\Pr\{A_j \mid A_1 \cap A_2 \cap \cdots \cap A_{j-1}\} = (n-j+1)/(m-j+1)$$

    $\because n-j+1$ of the $m-j+1$ remaining slots are occupied

    So, by uniform hashing, the next probe is to the occupied slot with probability $(n-j+1)/(m-j+1)$

11.4 Open addressing

- Analysis of open-address hashing
  - ***Interpretation***

    If α is constant, an unsuccessful search takes $O(1)$ time.
    - If $\alpha = 0.5$, an unsuccessful search takes an average of $1/(1-0.5) = 2$ probes.
    - If $\alpha = 0.9$, an unsuccessful search takes an average of $1/(1-0.9) = 10$ probes.
  - **COROLLARY** The expected number of probes to insert is at most $1/(1-\alpha)$.

    Proof Since there is no deletion, insertion uses the same probe sequence as an unsuccessful search.

11.4 Open addressing

- Analysis of open-address hashing
  - **THEOREM**

    The expected number of probes in a successful search is at most $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$.

    Proof

    A successful search for a key uses the same probe sequence as when the key was inserted

    $\Rightarrow$ If key k was the $(i+1)^{st}$ key inserted, by the corollary, to search for key k, the **expected** # of probes made $\leq 1/(1-i/m) = m/(m-i)$

    since $\alpha = i/m$ at the time k was inserted

11.4 Open addressing

- Analysis of open-address hashing
  - **THEOREM** (Cont'd)

    Assume that the element being searched for is equally likely to be any of the n elements stored in the table.

    Taking average over all keys, we have

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i}=\frac{m}{n}\sum_{i=0}^{n-1}\frac{m}{m-i}$$

$$=\frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k}\leq\int_{m-n}^{m}\frac{1}{x}dx \qquad (A.12)$$

$$=\frac{1}{\alpha}\ln\frac{m}{m-n}=\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$