Hashing

- Consider the key-indexed search method that we studied earlier with symbol tables
  - Uses key value as array index rather than comparing the keys
  - Depends on the keys being distinct, and mappable to distinct integers to provide for the index within the array range
- Hashing is an extension of the key-indexed approach that handles general search application without any assumption on the properties of keys being able to be mapped onto distinct indices
  - The property of key being distinct still holds
- Search algorithms based on hashing consist of two parts:
  1. Compute a hash function
     - Transforms a key into an index
     - Hashing is also known as key transformation for this reason
  2. Collision-resolution process
     - Kicks in when two distinct keys get transformed to the same address
- Hashing is a good example of time-space tradeoff

Hash Tables

Effective data structure for implementing dictionaries
Symbol tables generated by a compiler - insert, search, delete
Worst case search time - \( \Theta(n) \)
Average case search time - \( O(1) \)
Effective when number of keys actually stored is small compared to the total number of possible keys

Direct-Address Tables

- Universe of keys \( U \) assumed to be reasonably small
  \[
  U = \{0, 1, \ldots, m - 1\}
  \]
- Assume that no two elements have the same key
- **Direct-Address Table** - An array \( T[0..m-1] \) in which each position, or slot corresponds to a key in the universe \( U \)
- If the set contains no element with key \( k \), then \( T[k] = \text{NIL} \)
- Dictionary operations
  - \texttt{direct_address_search} \( (T,k) \)
    \[
    \text{return}(T[k])
    \]
  - \texttt{direct_address_insert} \( (T,x) \)
    \[
    T[\text{key}[x]] \leftarrow x
    \]
  - \texttt{direct_address_delete} \( (T,x) \)
    \[
    T[\text{key}[x]] \leftarrow \text{NIL}
    \]
• Problems with direct addressing
  – If the universe $U$ is large, storing a table $T$ of size $|U|$ may be impractical, or even impossible
  – 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.
• Reduce the storage requirements to $\Theta(|K|)$, keeping the search for an element $O(1)$
• With direct addressing, an element with key $k$ goes in $T[k]$
• With hash addressing, an element with key $k$ goes in $T[h(k)]$
• Hash function $h$ is used to compute an address from key $k$
• $h$ maps the universe $U$ of keys into the slots of a hash table $T(0..m-1)$
  \[ h : U \rightarrow \{0, 1, \ldots, m-1\} \]
• An element with key $k$ hashes to slot $h(k)$
• $h(k)$ is the hash value of key $k$
• Result – reduction in the range of array indices that need to be handled
• Collision – Two keys hash to the same value
• Ideal hash function
  – Easy to compute
  – approximates a “random” function
• A simple hashing function
  – Consider a four character key called akey
  – Replace every character with its five bit representation (between 1 and 26)
    \[ akey \equiv 00001 \ 01011 \ 00101 \ 11001 \]
  – Decimal equivalent – 44217
  – Select a prime number of locations in the array – $m = 101$
  – Location corresponding to akey – $44217 \mod 101 = 80$
  – The key barh also hashes to location 80 – collision
  – Why prime number of locations for the hashing function
    * Arithmetic properties of the mod function
    * The number 44217 can be written as
      \[ 1 \cdot 32^3 + 11 \cdot 32^2 + 5 \cdot 32^1 + 25 \cdot 32^0 \]
    * If $m$ is chosen to be 32, the value of hash function is simply the value for the last character
• Collision resolution by chaining
  – Simplest collision resolution technique
  – Put all the elements that hash to the same address in a linked list
  – Address $j$ contains a pointer to the head of the list
  – If no elements hash to the address, the corresponding slot contains nil
  – New definition for dictionary operations
* **chained_hash_insert** \((T,x)\)
  insert \(x\) at the head of the list \(T[h(key[x])]\)
  Worst-case running time \(- O(1)\)

* **chained_hash_search** \((T,k)\)
  search for an element with key \(k\) in list \(T[h(k)]\)
  Worst-case running time proportional to length of list

* **chained_hash_delete** \((T,x)\)
  delete \(x\) from the list \(T[h(key[x])]\)
  \(O(1)\) if lists are doubly linked

- Analysis of hashing with chaining
  - Given – Hash table \(T\) with \(m\) slots to store \(n\) elements
  - Load factor \(- \alpha\) for \(T = n/m\)
  - Assume that \(\alpha\) stays constant as \(m\) and \(n\) approach infinity
  - No other restriction on \(\alpha\); can be \(< 1\), \(= 1\), or \(> 1\)
  - Worst case behavior
    - All \(n\) keys hash to the same address
    - A list of length \(n\)
    - Worst case time for search \(- \Theta(n) + \) Time to compute hash function
  - Average case performance
    - Depends upon the distribution of keys among \(m\) addresses by \(h\)
    - Simple uniform hashing
  - Assume that \(h(k)\) can be computed in \(O(1)\) time
  - Time for search depends linearly upon the length of the list \(T[h(k)]\)

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

**Theorem 2** 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.

- by above theorems, if the number of hash addresses is at least proportional to the number of elements in the table, \(n = O(m)\)
  Consequently, \(\alpha = n/m = O(m)/m = O(1)\)

**Hash Functions**

- What is a good hash function?
  - Each key is equally likely to hash to any of the \(m\) addresses
  - Compute the hash value as independent of any patterns in data

- Interpreting keys as natural numbers

- **The division method**
  - Hash function \(- h(k) = k \mod m\)
  - Good values for \(m\) are primes not too close to a power of 2

- **The multiplication method**
  - Two steps
Hashing

Multiply the key \( k \) by a constant \( A \), \( 0 < A < 1 \), and extract the fractional part of \( kA \)

Multiply this value by \( m \) and take the floor of the result

- Also given by \(- h(k) = \lfloor m(kA \mod 1) \rfloor\)
- Value of \( m \) is not critical any more
- Typically, \( m \) is chosen to be \( 2^p \) for some integer \( p \)

- Universal hashing
  - Choose the hash function randomly in a way that is independent of the keys to be stored from a set of hash functions

Open Addressing

- All elements stored in the hash table itself
- Possible to “fill up” the table so that no more insertions can be made
- Load factor \( \alpha \) can never exceed 1
- No need for pointers – the space used by pointers can be added to hash table address space to yield fewer collisions and faster retrieval
- “probing” for insertion
- Possible to probe after a fixed number of keys rather than successive keys
- New hash function
  \[ h : U \times \{0, 1, \ldots, m - 1\} \rightarrow \{0, 1, \ldots, m - 1\} \]

- Probe sequence
  \[ \langle h(k, 0), h(k, 1), \ldots, h(k, m - 1) \rangle \]
  must be a permutation of \( \langle 0, 1, \ldots, m - 1 \rangle \) so that every hash table position can be eventually considered

- Procedure to insert in a hash table

  ```
  hash_insert (T, k)
  i ← 0
  repeat
    j ← h(k, i)
    if T[j] = nil then
      T[j] ← k
      return j
    else
      i ← i + 1
  until i = m
  error "hash table overflow"
  ```

- Procedure to search in a hash table

  ```
  hash_search (T, k)
  i ← 0
  repeat
    j ← h(k, i)
    if T[j] = k then
      return j
    i ← i + 1
  until T[j] = nil or i = m
  return nil
  ```
Hashing

- Procedure to delete from hash table
- Probing sequences
  - Linear probing
    * Easy to implement
    * Given an ordinary hash function $h' : U \rightarrow \{0, 1, \ldots, m - 1\}$
      
      the method of linear probing uses the hash function
      
      $$h(k, i) = (h'(k) + i) \mod m$$

      for $i = 0, 1, \ldots, m - 1$.
    * Suffers from the problem of primary clustering
  - Quadratic probing
    * Better than linear probing
    * Hash function is 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 $c_2 \neq 0$ are auxiliary constants, and $i = 0, 1, \ldots, m - 1$.
    * Leads to a milder form of clustering known as secondary clustering
  - Double hashing
    * Uses a hash function of the form
      
      $$h(k, i) = (h_1(k) + ih_2(k)) \mod m$$

      where $h_1$ and $h_2$ are auxiliary hash functions
    * First position to be probed is $T[h_1(k)]$
    * Successive probe positions are offset from previous position by $h_2(k) \mod m$