Topic 8: Hash Tables
Dictionary Operations

- Given a universe of elements $U$
- Need to store some keys
- Need to perform the following for keys
  - Insert
  - Search
  - Delete

Let’s call this a dictionary.
First Try:

- Use an array $T$ of the size of universe.

```plaintext
DIRECT-ADDRESS-SEARCH($T, k$)
    return $T[k]$

DIRECT-ADDRESS-INSERT($T, x$)
    $T[\text{key}[x]] = x$

DIRECT-ADDRESS-DELETE($T, x$)
    $T[\text{key}[x]] = \text{NIL}$
```

Each operation: $O(1)$.

Not great if universe size is much larger than the number of keys ever needed.
Hash Table

- $U$: universe
- $T[0 \ldots m-1]$: a hash table of size $m$
  - $m \ll |U|$

- Hash functions
  - $h: U \rightarrow \{0, 1, \ldots, m - 1\}$
  - $h(k)$ is called the **hash value** of key $k$.
    - Given a key $k$, we will store it in location $h(k)$ of hash table $T$. 
Collisions

- Since the size of hash table is smaller than the universe:
  - Multiple keys may hash to the same slot.

- How to handle collisions?
  - Chaining
  - Open addressing
Collision Resolved by Chaining

- \( T[j] \): a pointer to the head of the linked list of all stored elements that hash to \( j \)
- Nil otherwise
Dictionary Operations

- **Chained-Hash-Insert** \((T, x)\)
  - Insert \(x\) at the head of list \(T[h(key(x))]\)

- **Chained-Hash-Search** \((T, k)\)
  - Search for an element with key \(k\) in list \(T[h(k)]\)

- **Chained-Hash-Delete** \((T, x)\)
  - Delete \(x\) from the list \(T[h(key(x))]\)
Average-case Analysis

- \( n \): \# elements in the table
- \( m \): size of table (\# slots in the table)
- Load factor:
  - \( \alpha = \frac{n}{m} \): average number of elements per linked list
  - Intuitively the optimal time needed
- Individual operation can be slow (\( O(n) \) time)
  - Under certain assumption of the distribution of keys, analyze expected performance.
Simple Uniform Hashing

- Simple uniform hashing assumption:
  - any given element is equally likely to hash into any of the m slots in T

- Let $n_j$ be length of list $T[j]$
  - $n = n_0 + n_1 + \cdots + n_{m-1}$
  - Under simple uniform hashing assumption:
    - expected value $E[n_j] = \alpha = \frac{n}{m}$

Why?
Let \( \{k_1, k_2, \ldots, k_n\} \) be the set of keys

**Goal:** Estimate \( E(n_j) \)

Let \( X_i = 1 \) if \( h(k_i) = j \)

\( 0 \) otherwise

**Note:** \( n_j = \sum_{i=1}^{n} X_i \)

Hence

\[
E[n_j] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m}
\]

\[
E[X_i] = \Pr[h(k_i) = j] \times 1 = \frac{1}{m}
\]
Search Complexity – Case 1

- If search is unsuccessful
  - Based on simple uniform hashing, a new key is equally likely to be in any slot
  
  \[ ET[h(k)] = \sum_{j=1}^{m} \frac{1}{m} E[n_j] = \alpha = \frac{n}{m} \]

  - Expected search time: \( \Theta(1 + \alpha) \)
Search Complexity – Case 2

- If the search of $k$ is successful
  - Note, simple uniform hashing assumption does not necessarily implies that there is a equal chance for $k$ in any slot.
  - **Assume**: $k$ is equally likely to be any of the $n$ elements already stored in the hash table.

Theorem:
Under simple uniform hashing assumption, the search procedure takes $\Theta(1 + \alpha)$ expected time, when using collision resolution by chaining.
Hash Functions

- Ideally,
  - Hash function satisfies the assumption of simple uniform hashing
- Hard to achieve without knowledge of distribution where keys are drawn from
- Give a few heuristic examples
Division Method

- \( h(k) = k \mod m \)
  - e.g, \( h(k) = k \mod 701 \)

- Choice of \( m \) is important
  - Power of 2 not very good
    - Depends only on few least significant bits
    - Higher bits not used
  - A good choice is a prime number not too close to exact power of 2

- Related: \( h(k) = (k \cdot p) \mod m \)
Multiplication Method

- Choose some $0 < A < 1$
- $h(k) = \lfloor m (k A \mod 1) \rfloor$
- Slower than division method, but choice of $m$ not so critical
- One reasonable choice of $A$:
  - $A \approx \frac{\sqrt{5} - 1}{2} \approx 0.6180339887 \ldots$
Open-address Hashing

- All keys are stored in the table itself
  - No extra pointers
- Each slot is either a key or NIL
- To hash a key $k$:
  - In the $i$th iteration, compute $h(k, i)$
  - If $h(k, i)$ is taken (not NIL)
    - Go to next iteration
  - If $h(k, i)$ is free
    - Store $k$ here in this slot. Terminate.
procedure HashTable.Insert(K, D)
1    j ← 0;
2    repeat
3        i ← h(K, j);
4        if (T[i] is NIL) then
5            T[i] ← (K, D);
6            return;
7        else
8            j ← j + 1;
9        end
10    until (j = m);
11    error “hash table overflow”;
procedure HashTable.Search(K)
1    \( j \leftarrow 0; \)
2    repeat
3       \( i \leftarrow h(K, j); \)
4       if \( (T[i].key = K) \) then
5          return \( (T[i].data); \)
6       end
7       \( j \leftarrow j + 1; \)
8    until \( (j = m) \) or \( (T[i] \text{ is NIL}); \)
9    return \( (\emptyset); \)
Re-hashing Functions

- \( h: U \times \{0, 1, ..., m - 1\} \rightarrow \{0, 1, ..., m - 1\} \)

  **Possible choices**
  - **Linear probing:**
    \[ h(k, i) = (h'(k) + i) \mod m \]
  - **Quadratic probing:**
    \[ h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m \]
  - **Double hashing**
    \[ h(k, i) = (h_1(k) + i h_2(k)) \mod m \]

  Tend to cause primary clustering

  Secondary clustering
Remarks

- Advantages:
  - no pointer, no memory allocation during the course

- But:
  - Load factor $\alpha = \frac{n}{m} < 1$
  - Need resizing strategy when $n > m$
Summary

- Hash Table
  - Very practical data structure for dictionary operations
  - Especially when the number of keys necessary is much smaller than the size of universe
  - Need to choose hash functions properly
  - There exist more intelligent hashing schemes