Problem: to construct a dynamic set that supports the dictionary operations: search, insert and delete.

Examples:
- dictionary: word key to definition
- compiler: symbol key to semantic data
- **Key types:**
  - Numerical
  - Alphabet
- **Key space:** the set of all possible keys.
  
  Recall that we can search a sorted array quickly, so the question is
  
  **Can we use array?**
  
  Case 1: if keys are integer, directly index into array.
  
  Case 2: if keys are string of alphabets, convert to case 1 by first transforming characters to integers (say ASCII).
Dictionary operations are easily supported in such direct-address model.
- Each operation takes $O(1)$ time.
- Problem: key space may be too huge.
  e.g., names of at most 20 letters $\Rightarrow$ size of key space
  $= 26^{20} \approx 2^{100} \approx 10^{28}$
In practice, while key space is huge, only a small portion is really used, say a few millions of names in our example.

Hashing

Hash function $h$
$$h: U \rightarrow \{0, 1, \ldots, m-1\}$$
where $U$ is the key space and typically $m << |U|$.

Since $m$ is smaller than $|U|$, $h$ can not be a one-to-one mapping.

Collisions: a collision occurs between keys $k_1$ and $k_2$ if $h(k_1) = h(k_2)$.
Closed address hashing (chained hashing)

- Each position in hash table is pointer to head of a linked list.
- To insert elements into the table, add to head of list.

Insert(T,x)
insert x at the head of list T[h(key[x])].
worst-case running time is O(1).

Search(T,k)
search for an element with key k in list T[h(k)].
worst-case running time is proportional to length of list T[h(k)].

Delete(T,x)
delete x from the list T[h(key[x])].
worst-case running time is the time for searching x plus O(1) time for removing it from the list.
Uniform hashing: each key is equally likely to be hashed into any integer [0, ..., m-1].

**Load factor** $\alpha$: $n/m$, where $n$ is the number of keys that will be actually stored in the table. That is, $\alpha$ is the average length of lists. Therefore, average time for search = $O(1 + \alpha)$.

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

All dictionary operations can be supported in $O(1)$ time on average.

---

Open address hashing

- all elements stored in the array of the hash table (no linked lists).
  - More space efficient
  - Less flexible: **load factor $\alpha$ can not be larger than 1.**
- Rehashing to resolve collisions.
  - If a key $K$ is hashed to position $i$, which is already occupied, $K$ is rehashed to an alternative location:
    \[
    \text{rehash}(i+d) = (i+d) \mod m
    \]
  - where $d$ is an increment computed from $K$.
  - Linear probing: $d = 1$
Example: $h(x) = 5x \mod 8$

keys: 1055, 1492, 1776, 1812, 1918, 1945.

- $h(1055) = 3$
- $h(1492) = 4$
- $h(1776) = 0$
- $h(1812) = 4$
- $h(1918) = 6$
- $h(1945) = 5$

Example: $h(x) = 5x \mod 8$, $\text{rehash}(i) = (i+1) \mod 8$.

keys: 1055, 1492, 1776, 1812, 1918, 1945.

- $h(1055) = 3$
- $h(1492) = 4$
- $h(1776) = 0$
- $h(1812) = 4$, but $T[4]$ is occupied. Rehash(4) = $(4+1) \mod 8 = 5$, which is empty, so 1812 is stored in $T[5]$.
- $h(1918) = 6$
- $h(1945) = 5$, but $T[5]$ is occupied. Rehash(5) = 6, $T[6]$ is also occupied, so 6 is rehashed to 7, which is empty.
- Search(T,key)
  1. i = h(key);
  2. inc = hashInc(key); // for a general increment scheme
  3. while (T[i] ≠ nil and i < m)
  4. if (T[i] = key)
  5. then return i;
  6. i = rehash(i, inc); // i = i+1 for linear probing
  7. return nil;

Average time: for load factor $\alpha =1$, time is $\sqrt{n}$.

- Choice of Hash Functions
  - Distribute keys uniformly into integer range [0, 1, ..., m].
  - Low collision rate.
  - Hashing method I: division
    \[ h(k) = k \mod m \]
    - Must avoid certain values of m.
      - Powers of 2. If $m = 2^p$, $h(k)$ is p lowest order bits of k.
        - e.g., $m = 8 = 2^3$, 0 ≤ k ≤ 128
          - k = (107) = 1101011, h(k) = 011 = 3
          - k = (43) = 0101011, h(k) = 011 = 3
          - ...xxxx011,
            - there are 16 collisions on h(k) = 3.
      - Powers of 10. similar argument.
  - Good values for m are primes not too close to exact power of 2.
Hashing method II: multiplication

\[ h(k) = \lfloor m(kA \mod 1) \rfloor \]

where \( A \) is a constant, \( 0 < A < 1 \).

e.g., \( A = (\sqrt{5} - 1) / 2 \approx 0.6180339887 \ldots \)

\( m = 10000 \)

\[ h(123456) = \lfloor 10000 \times (123456 \times 0.61803\ldots \mod 1) \rfloor \]

\[ = \lfloor 10000 \times (76300.0041151\ldots \mod 1) \rfloor \]

\[ = \lfloor 10000 \times 0.0041151\ldots \rfloor \]

\[ = 41.151\ldots \]

\[ = 41. \]

Optimal choice of \( A \) depends on characteristics of data (Knuth suggests the golden ratio)

Choose \( m \) as power of 2.

Summary

- Hash tables are an effective data structure for implementing dictionaries.
- Worst-case: search may take as long as \( \Theta(n) \) time.
- Average-case: \( O(1) \).