# CH-231-A

# **Algorithms and Data Structures**

# ADS

## **Lecture 27**

Dr. Kinga Lipskoch

Spring 2020

# Direct Access Table

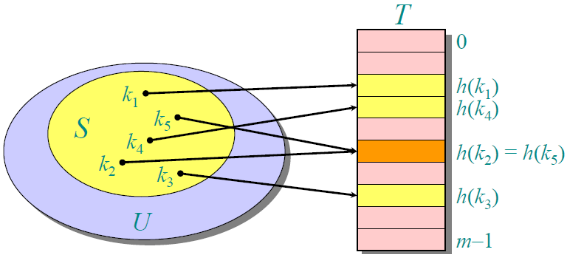
- The idea of a direct access table is that objects are directly accessed via their key.
- Assuming that keys are out of $U = \{0, 1, ..., m - 1\}$.
- Moreover, assume that keys are distinct.
- Then, we can set up an array $T[0..m - 1]$ with

$$T[k] = \begin{cases} x & \text{if } x \in K \text{ and } key[x] = k \\ \text{NIL} & \text{otherwise.} \end{cases}$$

- Time complexity: With this set-up, we can have the dynamic-set operations (Search, Insert, Delete, ...) in $\Theta(1)$.
- Problem: $m$ is often large. For example, for 64-bit numbers we have $18, 446, 744, 073, 709, 551, 616$ different keys.

# Hash Function

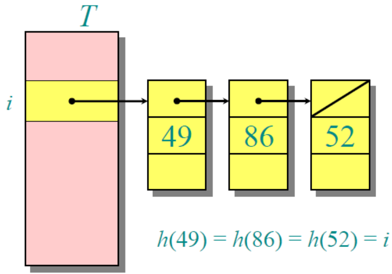
- Use a function $h$ that maps $U$ to a smaller set $\{0, 1, \ldots, m-1\}$.




- Such a function is called a hash function.
- The table $T$ is called a hash table.
- If two keys are mapped to the same location, we have a collision.

# Resolving Collisions

- Collisions can be resolved by storing the colliding mappings in a (singly-)linked list.




- Worst case: All keys are mapped to the same location. Then, access time is $\Theta(n)$.

# Average Case Analysis (1)

- Assumption (simple uniform hashing): Each key is equally likely to be hashed to any slot of the table, independent of where other keys are hashed.
- Let $n$ be the number of keys.
- Let $m$ be the number of slots.
- The load factor $\alpha = n/m$ represents the average number of keys per slot.

# Average Case Analysis (2)

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

- 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 $T[h(k)]$.
- Expected length of the list is $E[n_{h(k)}] = \alpha$.
- Time for computing $h(k) = O(1) \Rightarrow$ overall time $\Theta(1+\alpha)$.

# Average Case Analysis (3)

- Runtime for unsuccessful search:
  The expected time for an unsuccessful search is $\Theta(1+\alpha)$ including applying the hash function and accessing the slot and searching the list.
- What does this mean?
  - $m \sim n$, i.e., if $n = O(m) \Rightarrow \alpha = n/m = O(m)/m = O(1)$
  - Thus, search time is $O(1)$
- A successful search has the same asymptotic bound.

# Choosing a Hash Function (1)

- What makes a good hash function?
  - The goal for creating a hash function is to distribute the keys as uniformly as possible to the slots.
- Division method
  - Define hashing function $h(k) = k \bmod m$.
  - Deficiency: Do not pick an $m$ that has a small divisor $d$, as a prevalence of keys with the same modulo $d$ can negatively effect uniformity.
  - Example: if $m$ is a power of 2, the hash function only depends on a few bits: If $k = 1011000111011010$ and $m = 2^6$, then $h(k) = 011010$.

# Choosing a Hash Function (2)

- Division method (continue)
  - Common choice: Pick $m$ to be a prime not too close to a power of 2 or 10 and not otherwise prominently used in computing environments.
  - Example: $n = 2000$; we are ok with average 3 elements in our collision chain $\Rightarrow m = 701$ (a prime number close to $2000/3$), $h(k) = k \bmod 701$.

# Choosing a Hash Function (3)

- Multiplication method
  - On advantage of the multiplication method is that the value of $m$ is not critical
  - Knuth suggests that $A \approx (\sqrt{5} - 1)/2$ works well
  - Assume all keys are integers, $m = 2^r$, and the computer uses $w$-bit words.
  - Define hash function $h(k) = (A \cdot k \bmod 2^w) >> (w - r)$, where "$>>$" is the right bit-shift operator and $A$ is an odd integer with $2^{w-1} < A < 2^w$.
  - Example: $m = 2^3 = 8$ and $w = 7$.

$$
\begin{array}{rr}
 & 1011001 = A \\
\times & 1101011 = k \\
\hline
 & 1001010\underbrace{011}_{h(k)}0011
\end{array}
$$

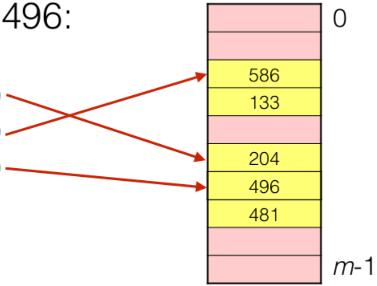
# Resolving Collisions by Open Addressing

- No additional storage is used.
- Only store one element per slot.
- Insertion probes the table systematically until an empty slot is found.
- The hash function depends on the key and the probe number, i.e., $h : U \times \{0, 1, ..., m - 1\} \rightarrow \{0, 1, ..., m - 1\}$.
- The probe sequence $< h(k, 0), h(k, 1), ..., h(k, m - 1) >$ should be a permutation of $\{0, 1, ..., m - 1\}$.

# Insert Example

- Insert key k = 496:

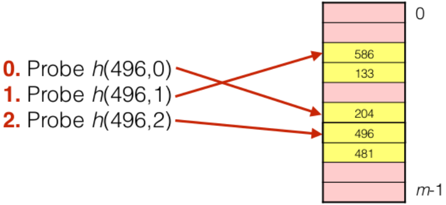
**0.** Probe $h(496,0)$
**1.** Probe $h(496,1)$
**2.** Probe $h(496,2)$




```
HASH-INSERT(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL
        T[j] = k
        return j
    else i = i + 1
until i == m
error "hash table overflow"
```

# Search Example

```
HASH-SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL or i == m
return NIL
```




- Search key $k = 496$
  - Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot (or made it all the way through the list)
  - Search times no longer depend on load factor $\alpha$
- What about delete?
  - Have a special node type: DELETED
  - Chaining more commonly used when keys must also be deleted

# Probing Strategies (1)

Linear probing:

- Given an ordinary hash function $h'(k)$, linear probing uses the hash function $h(k, i) = (h'(k) + i) \bmod m$.
- This is a simple computation.
- However, it may suffer from primary clustering, where long runs of occupied slots build up and tend to get longer.
  - empty slot preceded by $i$ full slots gets filled next with probability $(i + 1)/m$

# Probing Strategies (2)

Quadratic probing:

- Quadratic probing uses the hash function $h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$.
- Offset by amount that depends on quadratic manner, works much better than linear probing
- But, it may still suffer from secondary clustering: If two keys have initially the same value, then they also have the same probe sequence
- In addition $c_1$, $c_2$, and $m$ need to be constrained to make full use of the hash table