closed hashing (open addressing):

Hash table is an array of objects. Each entry of table can store a fixed number of elements, usually 1.

schemes for collision resolution:

Linear probing, Quadratic probing, Double hashing

Each scheme uses a sequence of probes: $s_0, s_1, \cdots s_k$, where $s_0 = h(x)$.

Linear probing: $s_i = s_0 + i \pmod n$ n = Table Size

s_0, insert(x) will try to find a vacant spot for x in Table[s_0], Table[s_0+1], ... until it succeeds.

Advantage: easy to implement Disadvantage: clustering

For a cluster of c (contiguous elements in the hash table), probability that a new element extends the cluster $\approx \frac{c}{n}$.

Therefore, bigger clusters have higher chances of getting bigger.

```
Insert(x):
    i ← Find(x)
    if Table[i] is free
        Table[i] ← x
```

```
Find(x):
    s0 ← h(x)
    Try probing sequence:
    for i ∈ {s0, s1, ...} do
        if Table[i] = x or
            Table[i] is available
        return i
```

```
Delete(x):
    i ← Find(x)
    if Table[i] = x
    then
        Mark Table[i]
        as deleted
```

Quadratic probing: $s_i = s_0 + i^2 \pmod n$

Double hashing: $s_i = s_0 + i \cdot h_2(x) \pmod n$ h_2 = secondary hash function

Advanced Hashing Algorithms

1. K-choice hashing (for separate chaining)

   Hash functions $h_1, h_2, \ldots, h_K$ (usually $K = 2$)

   Insert: K choices: Table $[h_1(x)]$, Table $[h_2(x)]$, ... Table $[h_K(x)]$
   choose smallest list and insert x into it
   (after verifying x is not already there).

   Find: check all K lists: Table $[h_i(x)]$ for $i = 1 .. K$

   Delete: Find x and delete it

   Advantages of K-choice hashing over separate chaining (1-choice):
   - Significant reduction in maximum chain length
   - Reduction in number of empty buckets

   Disadvantage:
   - Multiple hash functions are needed
   - More complex code
   - Find needs to check multiple lists

2. Improving closed hashing (open addressing):

   Load factor $\lambda = \frac{\text{Number of elements in table}}{\text{Table Size } n}$

   Traditional algorithms like Linear, Quadratic, Double probing sequences degrade in performance when $\lambda > 0.5$.
   Rehashing into a larger table (say, $2 * n$) is needed when load factor goes above 50%.

Cuckoo Hashing

Use 2 hash functions h_1, h_2 (generalizes to K hash functions for $K > 2$).

Idea: Insert(x): if Table$[h_1(x)]$ or Table$[h_2(x)]$ is vacant, insert x there. Otherwise, x replaces the element y in Table$[h_1(x)]$ and y is inserted back using h_2 by calling reinsert(y, 2, 1):

```
reinsert(y, i, t)   // Element y is reinserted into
                    // table using hash function h_i
                    // t = number of trials
  if Table[h_i(y)] is
  free then
      Table[h_i(y)] ← y
      return
  else if t+1 > retryLimit then
      resize table if load factor is high, or
      insert y into overflow table that uses
          another hashing scheme

  j ← i+1
  if j > k then j = 1
  z ← Table[h_i(y)]
  Table[h_i(y)] ← y
  reinsert(z, j, t+1)
```

Advantage: Find and Delete run in $O(k)$ time.

Find(x): Look for x in Table$[h_i(x)]$, for $i = 1..k$

Delete(x): find x and mark it as deleted

Robin Hood Hashing

Goal: Reduce worst-case search time.

Each element stores its probe sequence length.

Insert(x): probe sequence $s_0, s_1, \ldots, s_k$, where $s_0 = h(x)$.

if Table$[s_i]$ is free, Table$[s_i] \leftarrow x$ with probe length i

else if Table$[s_i]$ is occupied by y with probe length j:

if $i > j$ then

replace y by x:

Table$[s_i] \leftarrow x$ with probe length i

Reinsert y into Table.

Algorithm also needs to store the maximum probe length of any element in the table.

Find(x):

Use probe sequence $s_0, s_1, \ldots s_k$, where $s_0 = h(x)$ and k = max probe sequence length

Look for x in Table$[s_i]$, $i = 0 \ldots k$

Delete(x):

Find x and mark it as deleted

Reorganization:

When load factor gets high, or when maximum probe length is too large, table is reorganized by reinserting elements into a new table, possibly with a new hash function.

Hopscotch Hashing

Parameter K specifies how far an element is allowed to deviate from $h(x)$; i.e., x is stored in $\text{Table}[s_0 .. s_0+K]$, where $s_0 = h(x)$.

Idea: Insert(x): If a vacant spot is available in $\text{Table}[h(x)..h(x)+k]$, insert x closest spot to $h(x)$.

If all spots are filled, find closest free spot $T[f]$.

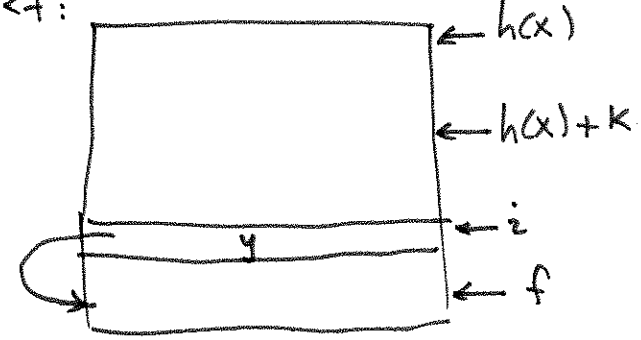
Try to move f closer to $h(x)$ by moving some element down to f:

For some y in $\text{Table}[i]$ with $i < f$:

if $f \leq h(y) + K$ then

$\text{Table}[f] \leftarrow y$

$f \leftarrow i$



By repeating such relocation operations, if f gets close enough to $h(x)$, then x is inserted into $\text{Table}[f]$.

Otherwise resize table, or increase k, or insert x into an overflow table.

Find(x): Look for x in $\text{Table}[h(x) .. h(x)+k]$.

Delete(x): Find x and mark it as deleted