Open Address Hash Tables

ESO207

Indian Institute of Technology, Kanpur

Open Addressing

- In ***open addressing***, there is no chaining. All elements occupy the hash table itself.
- Each entry of the table either contains an element of the dynamic set or is NIL.
- Searching for an item: systematically examine table slots until either we find the item or we have determined that the item is not in the table.
- No elements (lists, chains) are stored outside the table. The table is self-contained.
- Table can reach its capacity (i.e., become "full"). Implies that the load factor $\alpha = n/m$ will not exceed 1.

Open Address Table: insertion

- Choose a hash function that takes the key k and a *probe number* i and returns a slot of the table. That is,

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

- For the first *probe*, use hash value $h(k, 0)$ and consider the slot $T[h(k, 0)]$.
- If $T[h(k, 0)]$ is NIL(i.e., empty) then we insert the element in this slot.
- However, *it is possible that* $T[h(k, 0)]$ *is occupied*. Then, we try the slot $T[h(k, 1)]$, and if it is NIL, then we insert the element into this slot.

Insertion into Open Address Hash tables

- Otherwise, we try the slot $T[h(k, 2)]$ and so on. The process terminates if all the slots $T[h(k, 0)] \dots T[h(k, m-1)]$ are occupied.
- Open addressing requires that the probe sequence

$$h(k, 0), h(k, 1), \dots, h(k, m-1)$$

  is a permutation of $\{0, 1, 2, \dots, m-1\}$.
- Ensures that every hash table position is eventually considered as a slot for a new key as the table fills, and an insertion operation fails only when the table is completely full.

Insertion: pseudo-code

OPEN-ADDRESS-HASHING-INSERT(T, x)

$k = x.key$
$count = 0$
while $count < m$ and $T[h(k, count)] \neq$ NIL
    $count = count + 1$
if $count < m$
    $T[h(k, count)] = x$
else
    “Error: Hash table is full ”

Searching in open-address hash tables

- The algorithm for searching for key k *probes the same sequence of slots that was examined by the insertion algorithm*.
- This is necessary for correctness of the search algorithm. If during this probe sequence, any slot is found to be NIL, then the search stops and returns "not found".

Searching: Pseudo-code

OPEN-ADDRESS-HASHING-SEARCH(T, k)

    $count = 0$

    $slot = h(k, count)$

    while $count < m$ and $T[slot] \neq$ NIL and $T[slot] \neq x$

        $count = count + 1$

        $slot = h(k, count)$

    if $count == m$ or $T[slot] ==$ NIL

        return "Not Found"

    else

        return $slot$

Deletion:Open Addressing Hash tables

- Deletion operation requires some care.
- Suppose that at the time we inserted an element x with key k, it was placed in the slot numbered $h(k, 1)$.
- This means that the slot $h(k, 0)$ was occupied by some other y at the time x was inserted.
- Now suppose that y is deleted.
- If we replace the slot of y by NIL, the search algorithm searching for x will encounter NIL at slot number $h(k, 0)$.
- Would infer that the element with key k is not present in the table.
- But x (with key k) is present in the slot $h(k, 1)$, where it was inserted.

Deletion: Open Addressing

- The confusion arises because deletion of an element has replaced the element in that slot by NIL.
- Instead, we should replace it with some other constant such as DELETED, which signifies that there was an element here which was deleted.
- Slots marked DELETED are available for elements to be inserted into, that is, an insertion operation should treat a DELETED slot like a NIL and insert an element there.

Deletion: Pseudo-code

OPEN-ADDRESS-HASHING-DELETE($T, slot$)   //deletes element
  // at slot number *slot*

1. $T[slot] = \text{DELETED}$

The presence of the constant DELETED to indicate deleted element in a slot changes the pseudo-code for insertion: line 3 in the code for insertion changes to line 3'; the remaining code is unchanged.

3. **while** $count < m$ and $T[h(k, count)] \neq \text{NIL}$
        and $T[h(k, count)] \neq \text{DELETED}$
4. $count = count + 1$
5. **if** $count < m$
6. $T[h(k, count)] = x$
7. **else**
8.   "Error: Hash table is full "

Techniques for Open Addressing

- Three commonly used techniques to compute the probe sequences required for open addressing.
  1. *linear probing*,
  2. *quadratic probing*, and
  3. *double hashing*
- These techniques guarantee that the sequence $h(k, 0), h(k, 1), \ldots, h(k, m-1)$ is a permutation of $\{0, 1, \ldots, m-1\}$ for each key k.

Linear Probing

- Let $h' : U \to \{0, 1, \ldots, m-1\}$ be a hash function.
- Linear probing method uses the hash function

$$h(k, i) = (h_1(k) + i) \mod m, \quad i = 0, 1, \ldots, m-1$$

- h_1 is the ***auxiliary hash function***.
- The ith probe results in the slot $h_1(k) + i \mod m$. Given a key k, the probe sequence is

$$T[h_1(k)], T[h_1(k) + 1 \mod m], T[h_1(k) + 2 \mod m], \ldots$$

- That is, probe the slot $h_1(k)$, then the next slot, and then the slot next to it and so on wrapping around the table when we come to slot $m - 1$.

Operation:

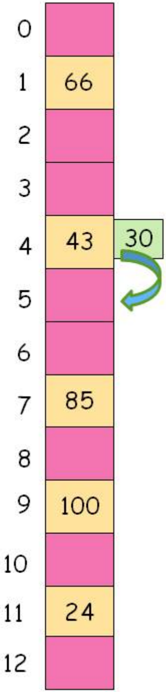
insert(30)

$30 = 4 \bmod 13$

$h_1(k) = k \bmod 13$

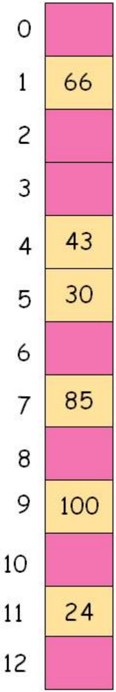
$h(k) = (h_1(k) + i) \bmod 13$

Linear Probing



Collision at slot 4!
So move to next slot,
which is empty.
Insert at slot 5

State after inserting 30



$h_1(k) = k \bmod 13$

$h(k) = (h_1(k) + i) \bmod 13$

Linear Probing

Operation:

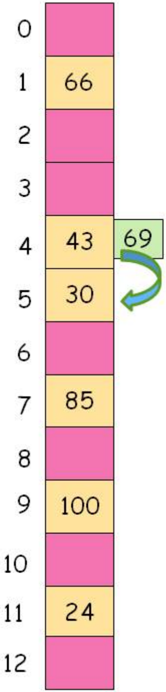
insert(69)

69 = 4 mod 13

$h_1(k) = k \bmod 13$

$h(k) = (h_1(k) + i) \bmod 13$

Linear Probing



Collision at slot 4!
So move to next slot,
which is also full.

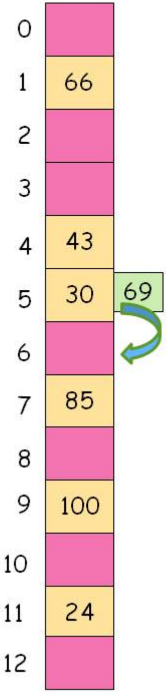
Operation:
insert(69)

69 = 4 mod 13

$h_1(k) = k \bmod 13$

$h(k) = (h_1(k) + i) \bmod 13$

Linear Probing



Collision at slot 5!
So move to next slot,
which is empty,

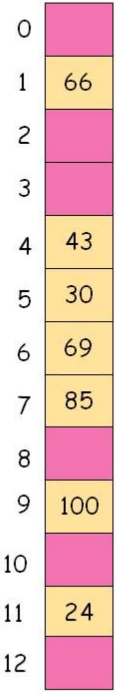
Operation:
insert(69)

69 = 4 mod 13

$h_1(k) = k \bmod 13$

$h(k) = (h_1(k) + i) \bmod 13$

Linear Probing



State after inserting 69

Linear Probing

- Probe sequence is completely determined by the first probe, hence there are only m distinct probe sequences.
- Linear probing is easy to implement.
- Suffers from a problem known as *primary clustering*.
- Long runs of occupied slots build up, increasing the average search time.
- Why? An empty slot preceded by i full slots gets filled next with probability $(i+1)/m$, where, it is assumed that insertions are uniformly distributed over the key space $\{0, 1, \ldots, m-1\}$).
- Thus long runs of occupied slots tend to get longer.

Quadratic Probing

- ***Quadratic Probing*** uses a hash function of the form

  $$h(k,i) = (h_1(k) + c_1 i + c_2 i^2) \mod m, \quad i = 0, 1, \ldots, m-1$$

  where h_1 is an auxiliary function and c_1 and c_2 are constants in $\{0, 1, \ldots, m-1\}$.
- The initial position probed is $T[h_1(k)]$.
- Later positions probed are offset by amounts that depend on the function $c_1 i + c_2 i^2$.
- Combinations of c_1 and c_2 are constrained to ensure that $h(k,0), \ldots, h(k, m-1)$ is a permutation of $\{0, 1, \ldots, m-1\}$, so that the entire hash table is used. (Discussed later).

Quadratic Probing

- If two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$.
- This leads to a milder form of clustering, called *secondary clustering.*
- As with linear probing, the initial probe determines the sequence, hence there are only m distinct probe sequences.
- Quadratic probing works better than linear probing in practice.

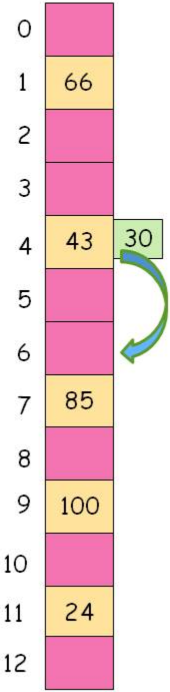
Operation:

insert(30)

30 = 4 mod 13

$h_1(k) = k \bmod 13$

$h(k) = (h_1(k) + i + i^2 \bmod 13$



Collisionat slot 4.
Next slot is
4 + 1 + 1 mod 13 = 6
Slot 6 is empty

Quadratic Probing

Operation:

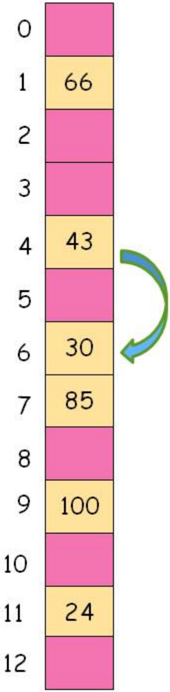
insert(30)

$30 = 4 \bmod 13$

$h_1(k) = k \bmod 13$

$h(k) = (h_1(k) + i + i^2$
$\bmod 13$

Quadratic Probing



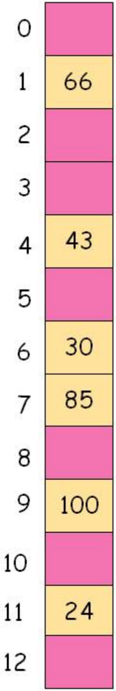
Collision at slot 4.
Next slot is
$4 + 1 + 1 \bmod 13 = 6$
Slot 6 is empty

Operation:
insert(30)

$h_1(k)$ = k mod 13

$h(k) = (h_1(k) + i + i^2$
mod 13

Quadratic Probing



State after inserting 30

Operation:

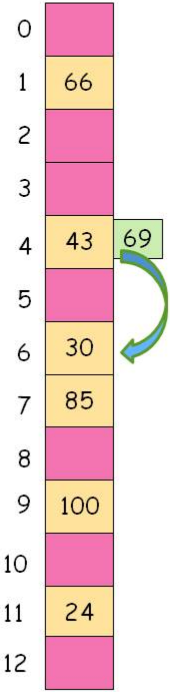
insert(69)

$69 = 4 \bmod 13$

$h_1(k) = k \bmod 13$

$h(k) = (h_1(k) + i + i^2) \bmod 13$

Quadratic Probing



Collisionat slot 4.
Next slot is
$4 + 1 + 1 \bmod 13 = 6$
This is also occupied.

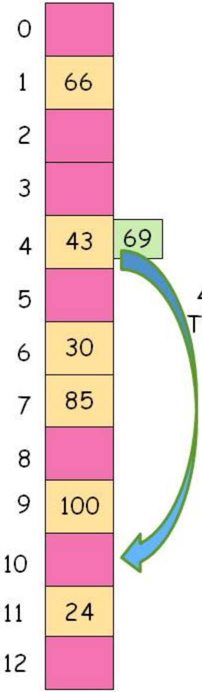
Operation:
insert(69)

69 = 4 mod 13

$h_1(k) = k \bmod 13$

$h(k) = (h_1(k) + i + i^2) \bmod 13$

Quadratic Probing



Collision at slot 4.
Next slot is
$4 + 2 + 2^2 \bmod 13 = 10$
This is empty, so insert
69 in slot 10.

Operation:
insert(69)

State after inserting 69

0	
1	66
2	
3	
4	43
5	
6	30
7	85
8	
9	100
10	69
11	24
12	

$h_1(k) = k \bmod 13$

$h(k) = (h_1(k) + i + i^2) \bmod 13$

Quadratic Probing

Double Hashing

- ***Double hashing*** uses two auxiliary hash functions, h_1 and h_2 and constructs a hash function of the form

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m, \quad i = 0, 1, \ldots, m-1$$

- The initial probe ($i = 0$) goes to slot $T[h_1(k)]$, the successive probes are offset from previous positions by the amount $h_2(k) \mod m$.
- The value $h_2(k)$ must be relatively prime to the hash table size m for the entire hash table to be searched.
- ***Why?*** Let us see an example first.

Example

- Suppose T has $m = 6$ slots $\{0, 1, \ldots, 5\}$.
- Let $h_1(k) = 0$ and $h_2(k) = 3$.
- Then, the sequence is

$$0, 0+3, 0+3\cdot 2, 0+3\cdot 3, 0+3\cdot 4, 0+3\cdot 5 \mod 6$$

.

- Taking modulo m, the sequence is

$$0, 3, 0, 3, 0, 3 \quad .$$

giving a repetitive sequence.

- Slots $1, 2, 4$ are not probed.
- Reason: $\gcd(6, 3) = 3$.

Example

- Let $h_2(k) = 4$ then, the sequence is $0, 0 + 4, 0 + 2 \cdot 4, 0 + 3 \cdot 4, 0 + 4 \cdot 4, 0 + 5 \cdot 4$ modulo 6.
- This is the same as

$$0, 4, 2, 0, 4, 2$$

  once again giving a repetitive sequence.
- The table slots $1, 3$ are not probed. This is because $\gcd(6, 4) = 2$.

Example

- However, if $h_2(k) = 5$ then the sequence is $0, 5, 10, 15, 20, 25$ modulo 6 which is

$$0, 5, 4, 3, 2, 1$$

- This covers the whole table (in some order).
- This is because 5 and 6 are relatively prime.

Double hashing

- Let $h_1(k) = a$ and $h_2(k) = b$.
- The sequence obtained is

$$a, a+b, a+2b, \ldots, a+(m-1)b \mod m$$

- If the entire table has to be searched then the above sequence must have be all the elements of the set $\{0, 1, \ldots .m-1\}$ in some order.
- Or, no two members of the sequence must repeat, that is, for any $i, j \in \{0, 1, \ldots, m-1\}$ such that $i \neq j$

$$a + ib = a + jb \mod m$$

must have no solution.
- Now $a + ib = a + jb \mod m$ is equivalent to

$$(i-j)b = 0 \mod m$$

that is, m divides $(i-j)b$.

Double Hashing

- If b is relatively prime to m, then, m divides $(i-j)b$ iff m divides $i-j$, which is not possible, since, $i, j \in \{0, 1, \ldots, m-1\}$ and $i \neq j$ and therefore,

$$i - j \mod m \in \{1, 2, \ldots, m-1\} \ .$$

- *Implies* if b is relatively prime to m then, $a + ib = a + jb \mod m$ has no solution for $i \neq j$.
- Or, for any fixed a, b, the set $\{a + ib \mod m\}$ is some reordering of $\{0, 1, \ldots, m-1\}$.
- Hence the entire table is searched.

Double Hashing

Let $g = gcd(m, b)$. Then, $a + ib = a + jb \mod m$ iff m/g divides $i - j$.

Proof:

- Let g be the greatest common divisor of m and b.
- Suppose that $g > 1$.
- Then, $m = g \cdot m'$ and $b = g \cdot b'$.
- Suppose $a + ib = a + jb \mod m$, or, m divides $(i - j)b$.
- iff $g \cdot m'$ divides $(i - j)g \cdot b'$, or, equivalently
- m' divides $(i - j)$, or,

$$i = j \mod m' .$$

Double Hashing

- So $a = a + m'b \mod m$.
- Hence, the sequence $a, a+b, \ldots, a+(m-1)b$ is the sequence $a, a+b, \ldots a+(m'-1)b$ repeated $g = m/m'$ times.
- Thus, only a fraction $m'/m = 1/g$ of the table entries are probed.

Double Hashing: Choice of second hash function

- **How can we ensure that $h_2(k)$ is relatively prime to m.**
- Example 1:
  1. Let m be a power of 2.
  2. Design h_2 so that it always returns an odd number.

Double Hashing: choice of second hash function

- Example 2:
  1. Let m be prime.
  2. Design h_2 so that it always returns a positive integer less than m.
  3. For example, let m be prime and let

$$h_1(k) = k \mod m$$
$$h_2(k) = 1 + (k \mod m')$$

     where $m' = m - 1$ (or any number less than m).
- E.g.: $k = 123456, m = 701, m' = 700$
  1. Then, $h_1(k) = 80$ and $h_2(k) = 257$.
  2. First probe is to slot 80, then every 257th slot (modulo m) until we find the key or have examined all slots.

Double Hashing: Benefits

- For m prime or power of 2, double hashing improves over linear or quadratic probing, since,
- $\Theta(m^2)$ possible hash sequences are used.
- *Why?* Each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence.
- In practice, double hashing with prime m or power of 2 is superior to linear and quadratic probing.