Lecture 12: Hash Table

Read: Carrano, Chpt.12.

ADT ***Dictionary:***

A collection class with ***insert***, ***delete*** and ***search*** operations.

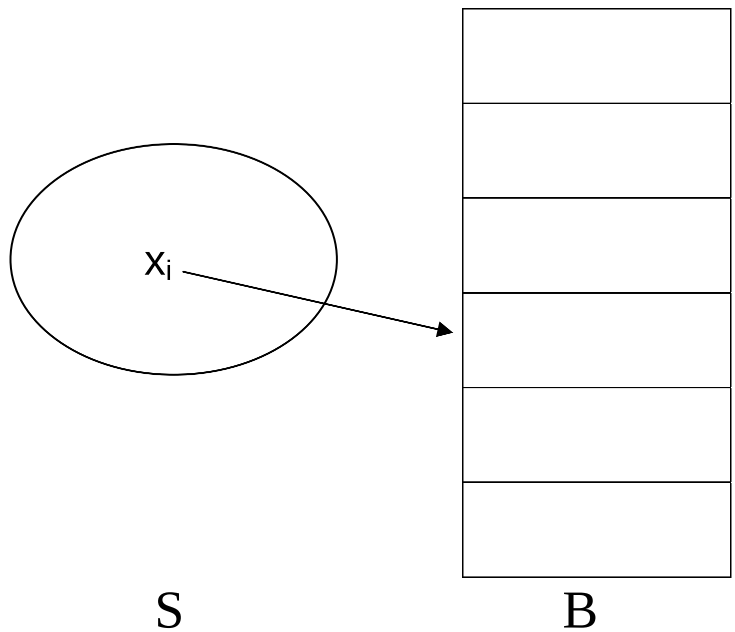
Hash Table: An implementation of dictionary consisting of

(1) *A set of m locations (buckets),* B[0..m–1]:
Used for storing a set of n objects with keys S = $\{x_1, x_2, \ldots, x_n\}$, $n \geq 0$.

(2) *A hash function* $h: S \rightarrow \{0, 1, \ldots, m-1\}$:
For any given data object with key $x_i \in S$, the location $B[h(x_i)]$ will be used to store the given object if it is not already occupied by another object.

(3) A collision resolution scheme:
Used to determine an alternate location for storing an object whenever it is hashed into a location that is already occupied with an existing object. A ***collision*** is said to occur whenever two or more objects are hashed into the same location.

Hashing: A process in finding location in B[0..m−1] for storing any given object with key in S.



Q: How do we store an object in B?

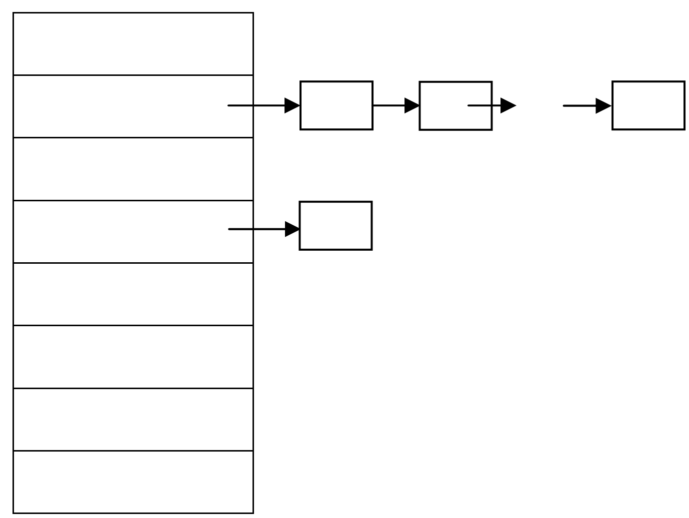
Organizing a Hash Table:

1. *Open (External) hashing*: Locations store pointers (reference locations) to objects.

2. *Closed (Internal) hashing*: Locations store actual objects.

Examples of Hashing Schemes:

1. (Open) Hashing with Separate Chaining:
   Objects hashed into the same address are simply linked (chained) together.



B

Consider the Search operation:

(1) Compute $h(x_i)$ to find location $B[h(x_i)]$.
(2) Search the linked structure at $B[h(x_i)]$ sequentially for object with key x_i.

$T(n)$ = cost in computing $h(x_i)$ +
cost in searching the linked list at $B[h(x_i)]$

Q: How should a hash table be designed so that it will have good performance?

Remark: In the worst-case, a chain may contain all n objects. To minimize searching time, each table location should contain a chain with roughly n/m objects.

A "*good*" hash function is a function that

(1) Can be computed in $\Theta(1)$ time, and

(2) Distributes the objects evenly over all locations with each location having roughly n/m items.

Define ***load factor*** = λ = n/m.

Assuming that a good hash function h is used, we have

Unsuccessful search:

$$\begin{aligned} T_a(n) &= \Theta(1) + \Theta(1)(n/m) \\ &= \Theta(n/m) \\ &= \Theta(\lambda) \end{aligned}$$

Successful search:

$$\begin{aligned} T_a(n) &= \Theta(1) + \Theta(1)[(n/m)/2] \\ &= \Theta(n/m) \\ &= \Theta(\lambda) \end{aligned}$$

When m = $\Theta(n)$, we have

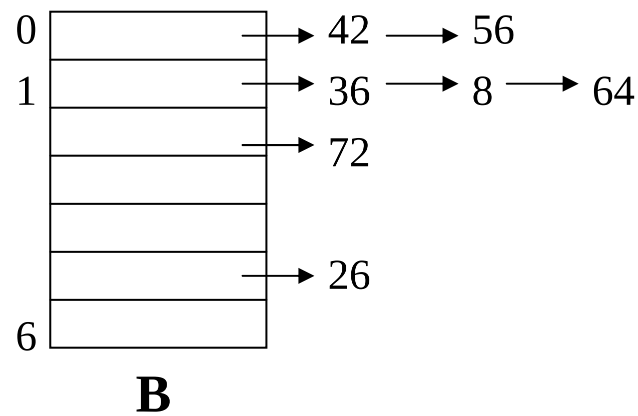
$$\begin{aligned} T_a(n) &= \Theta(n/m) \\ &= \Theta(1), \text{ which is the best possible!} \end{aligned}$$

Observe that $\mathbf{T_a(n) = \Theta(\lambda)}$**.** As n increases, λ also increases, and efficiency of operations decreases.

A Simple (but not too good) Hash Function:
Define $h(x_i) = x_i \bmod m$, where m is chosen to be a prime.

Example: Take m = 7. Insert 64, 26, 56, 72, 8, 36, and 42 into an initially empty hash table using separate chaining and hash function h(x) = x mod m

64 % 7 = 1,
26 % 7 = 5,
56 % 7 = 0,
72 % 7 = 2,
8 % 7 = 1,
36 % 7 = 1,
42 % 7 = 0.



Possible Extension

Singly linked list can be replaced with more advanced data structures so as to speed up searching once a location is found.

HW: Study other hash functions in text.

Advantages of Hashing with Chaining

1. Simplicity (in concept and implementation).
2. Insertion is always possible; hence, a small table can be used to store any number of data (efficiency will suffer).

Disadvantages of Hashing with Chaining

1. Can degenerate into a single chain with $T_w(n) = O(n)$.
2. Memory intensive: Need to implement/store pointers.
3. Slower speed: Indirect accessing data; need to follow pointers to data.

2. (Closed) Hashing with Open Addressing Scheme:

Given hash function h. For some fixed integer k, define a sequence of hash functions $h_i(x) = (h(x) + f_i) \bmod m$, with $0 \le i \le k$ and $f_0 = 0$.

The set of functions $\{f_0, f_1, \ldots, f_k\}$ is called ***collision resolution functions***.

For any given object with key x. Compute $h_0(x) = h(x)$, $h_1(x), \ldots, h_k(x)$ to find the first available location for inserting x.

Some Simple Open Addressing Schemes:

(i) Linear Probing:

Assume that $h(x) = j$ and $B[j]$ is occupied.
Search $B[j+1], B[j+2], \ldots, B[m-1], B[0], B[1], \ldots, B[j-1]$ sequentially to find the first available location to insert x. If no empty location is found, report overflow.

Recall that $h_i(x) = (h(x) + f_i) \bmod m$, $0 \le i \le k$, and $f_0 = 0$.

Define $f_i = i$, which is a *linear function*, we have

$$
\begin{aligned}
h_0(x) &= (h(x) + 0) \bmod m \\
&= h(x), \\
h_1(x) &= (h(x) + f_1) \bmod m \\
&= (h(x) + 1) \bmod m, \\
h_2(x) &= (h(x) + f_2) \bmod m \\
&= (h(x) + 2) \bmod m, \\
&\quad \ldots \\
h_k(x) &= (h(x) + f_k) \bmod m \\
&= (h(x) + k) \bmod m.
\end{aligned}
$$

Example: Take m = 7. Insert 64, 26, 56, 72, 8, 36, 42, using linear probing and hash function h(x) = x mod m, into an initially empty hash table.

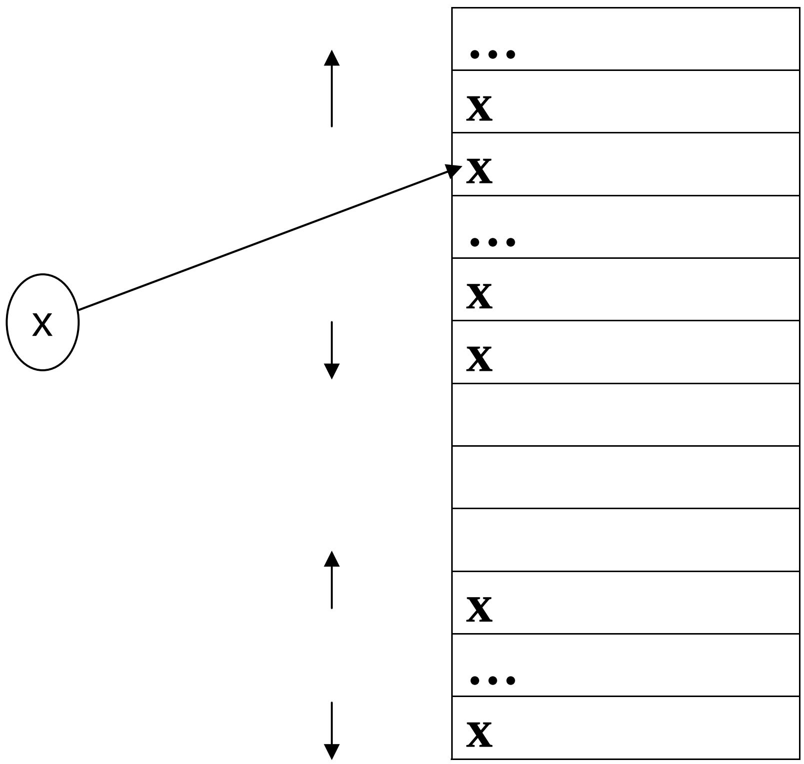
64 % 7 = 1,
26 % 7 = 5,
56 % 7 = 0,
72 % 7 = 2,
8 % 7 = 1 → 2 → 3,
36 % 7 = 1 → 2 → 3 → 4,
42 % 7 = 0 → 1 → 2 → 3 → 4 → 5 → 6.

Hash table using linear probing:

56
64
72
8
36
26
42

Remark: When blocks of locations are occupied, closed hashing with linear probing may result in ***primary clustering***, which are blocks of occupied locations.

Primary Clustering:



Remark: Primary clustering behaves like long chain and degrades the performance of the table!

Remedy: Use quadratic probing to eliminate primary clustering.

(ii) **Quadratic Probing**:

Recall that $h_i(x) = (h(x) + f_i) \bmod m$, $0 \le i \le k$, and $f_0 = 0$.

Define $f_i = i^2$, which is a *quadratic function*, we have

$$
\begin{aligned}
h_0(x) &= (h(x) + 0^2) \bmod m \\
&= h(x),
\end{aligned}
$$

$$
\begin{aligned}
h_1(x) &= (h(x) + f_1) \bmod m, \\
&= (h(x) + 1^2) \bmod m,
\end{aligned}
$$

$$
\begin{aligned}
h_2(x) &= (h(x) + f_2) \bmod m, \\
&= (h(x) + 2^2) \bmod m,
\end{aligned}
$$

. . .

$$
\begin{aligned}
h_k(x) &= (h(x) + f_k) \bmod m, \\
&= (h(x) + k^2) \bmod m.
\end{aligned}
$$

Example: Take m = 7. Insert 64, 26, 56, 72, 8, 36, 42, using quadratic probing and hash function h(x) = x mod m, into an initially empty hash table.

Addresses Computation:

64 % 7 = 1,
26 % 7 = 5,
56 % 7 = 0,
72 % 7 = 2,
8 % 7 = 1 $\rightarrow$ 2 $\rightarrow$ 5 $\rightarrow$ 3,
36 % 7 = 1 $\rightarrow$ 2 $\rightarrow$ 5 $\rightarrow$ 3 $\rightarrow$ 3 $\rightarrow$ 5 $\rightarrow$ 2 $\rightarrow$ …

Hash table using quadratic probing:

56
64
72
8
26

Problems with Closed Hashing:

1. Insertion may fail even though the table is not empty.
2. May form ***secondary clustering***.

More Problems with Closed Hash Table:

Consider the following example.

Example: Take m = 7. Insert 64, 56, 72, 8, followed by delete 64 and then delete 8, using linear probing and hash function h(x) = x mod m, into an initially empty hash table.

Addresses Computation:

64 % 7 = 1,
56 % 7 = 0,
72 % 7 = 2,
8 % 7 = 1 → 2 → 3,

Hash table after inserting 64, 56, 72 and 8:

56
64
72
8

Hash table after deleting 64:

56
72
8

Q: How do we delete 8?

Recall that 8 % 7 = 1 but B[1] is empty. Hence, we must continue searching for x even though an empty bucket is found!

Q: When can we stop searching?

Observation:

Two types of empty buckets:

1. A bucket that is always empty: Searching terminates.
2. A bucket that is emptied by deletion: Searching must continue.

Data Structure for Bucket:

Using an extra flag/Boolean field:

flag = true ⇒ Bucket is emptied by deletion; searching must continues.

flag = false ⇒ Bucket is always empty; searching terminates.

Advantages of Closed Hashing with Open Addressing:

1. Faster speed: No need to follow pointers.
2. Less memory consumption: No need to implement pointers.

Disadvantages of Hashing with Open Addressing

1. Much more complex.
2. Can degenerate into primary or secondary clustering.
3. Deletion/Find operations are much more complex.
4. Insertion is not always possible even though the table is not empty.

Theorem: When m is prime and the table is at least half-empty; i.e., $\lambda < 1/2$, we can always insert a new item into a closed hash table using quadratic probing.

Conclusions:

1. To guarantee good performance in a hash table, a prime number should be chosen for m such that $\lambda < 1$ for open hashing and $\lambda < 1/2$ for closed hashing.
2. Must monitor λ during the lifetime of your hash table.
3. Hashing with open addressing (eg. quadratic probing) outperforms hashing with chaining only if implemented correctly!

Q: What happens when insertion/deletion become increasingly difficult?

Need a new hash table with larger/smaller size!

Rehashing:

A process in hashing all the elements of an existing hash table H into a new hash table H*.

$$\mathbf{H} \quad \leftrightarrow \quad \mathbf{H^*}$$

tableSize m (prime) $\leftrightarrow$ tableSize ~2m (prime)

Remark: Rehashing is a very expensive process and should only be performed infrequently.

Q: When do we rehash?

1. When $\lambda \to 1$ for open hashing and $\lambda \to 1/2$ for closed hashing.
2. Use a pre-specified λ to determine when to rehash.
3. When insertion becomes increasingly difficult or fails.
4. When deletion becomes increasingly difficult.

(iii). **Another Open Addressing Scheme: Double Hashing**:

Use two hash functions h and h^+ such that the collision functions f_i's are functions of i and h^+.

Now, define $f_i = ih^+$. (or i^2h^+, or others)

Observe that

$$h_0(x) = (h(x) + 0h^+(x)) \bmod m$$
$$= h(x),$$

$$h_1(x) = (h(x) + f_1) \bmod m,$$
$$= (h(x) + 1h^+(x)) \bmod m,$$

$$h_2(x) = (h(x) + f_2) \bmod m,$$
$$= (h(x) + 2h^+(x)) \bmod m,$$

. . .

$$h_k(x) = (h(x) + f_k) \bmod m,$$
$$= (h(x) + kh^+(x)) \bmod m.$$

A Simple h^+ Function:

Define $h^+(x) = R - (x \bmod R)$, where $R < m$ is a prime.

Example: Take m = 7, R = 5. Insert 64, 26, 56, 72, 8, 36, 42, using double hashing with hash functions $h(x) = x \bmod m$, $h^+(x) = R - (x \bmod R)$, and $f_i = ih^+$, into an initially empty hash table.

Addresses Computation:

64 % 7 = 1,
26 % 7 = 5,
56 % 7 = 0,
72 % 7 = 2,

8 % 7 = 1,
$h^+(x) = R - (x \bmod R) = 5 - (8 \bmod 5) = 2$,
$h_1(x) = (h(x) + 1h^+(x)) \bmod m = (1 + 2) \bmod 7 = 3$,

36 % 7 = 1,
$h^+(x) = R - (x \bmod R) = 5 - (36 \bmod 5) = 4$,
$h_1(x) = (h(x) + 1h^+(x)) \bmod m = (1 + 4) \bmod 7 = 5$,
$h_2(x) = (h(x) + 2h^+(x)) \bmod m = (1 + 8) \bmod 7 = 2$,
$h_3(x) = (h(x) + 3h^+(x)) \bmod m = (1 + 12) \bmod 7 = 6$,

$42 \% 7 = 0$,
$h^+(x) = R - (x \bmod R) = 5 - (42 \bmod 5) = 3$,
$h_1(x) = (h(x) + 1h^+(x)) \bmod m = (0 + 3) \bmod 7 = 3$,
$h_2(x) = (h(x) + 2h^+(x)) \bmod m = (0 + 6) \bmod 7 = 6$,
$h_3(x) = (h(x) + 3h^+(x)) \bmod m = (0 + 9) \bmod 7 = 2$,
$h_4(x) = (h(x) + 4h^+(x)) \bmod m = (0 + 12) \bmod 7 = 5$,
$h_5(x) = (h(x) + 5h^+(x)) \bmod m = (0 + 15) \bmod 7 = 1$,
$h_6(x) = (h(x) + 6h^+(x)) \bmod m = (0 + 18) \bmod 7 = 4$.

Hash table using double hashing:

56
64
72
8
42
26
36