**EECS560 Data Structures Kong**

## Topic 2: Dictionary and Hash Tables

什么时候用hash Tables？
选择什么hashTable来处理数据？eg.linear,qua, channing, and double hashing

**Read:** Chpt. 5, Weiss.

**Q:** Given a large data set S of size n. How do we store and maintain this given set of data objects such that some "useful" operations can be performed on them efficiently?

**Some Useful Operations:**

- **Static Operations:** Operations that will not modify/change the underlying data structure organization.
  Examples: size, isEmpty, find, findMin, findMax, …
- **Dynamic Operations:** Operations that will modify/change the underlying data structure organization.
  Example: insert, delete, sort, reverse, merge, …

**Simple Standard Classifications of ADTs:**

1. **Dictionary:** *find, insert*, *delete*, …
2. **Priority Queue**: *insert*, *findMax*, *deleteMax*, …
3. **Double-Ended Queue**: *insert*, *findMin*, *findMax*, *deleteMax*, *deleteMin,…*
4. **Concatenated Queue**: *insert*, *findMax*, *deleteMax*, *merge*, …

**Dictionary**:

A collection class of data objects supporting *find, insert*, and *delete* operations effectively.

**Hash Table:** A simple implementation of dictionary using a table, which is just a set of locations.

**Structure of a Hash Table:**

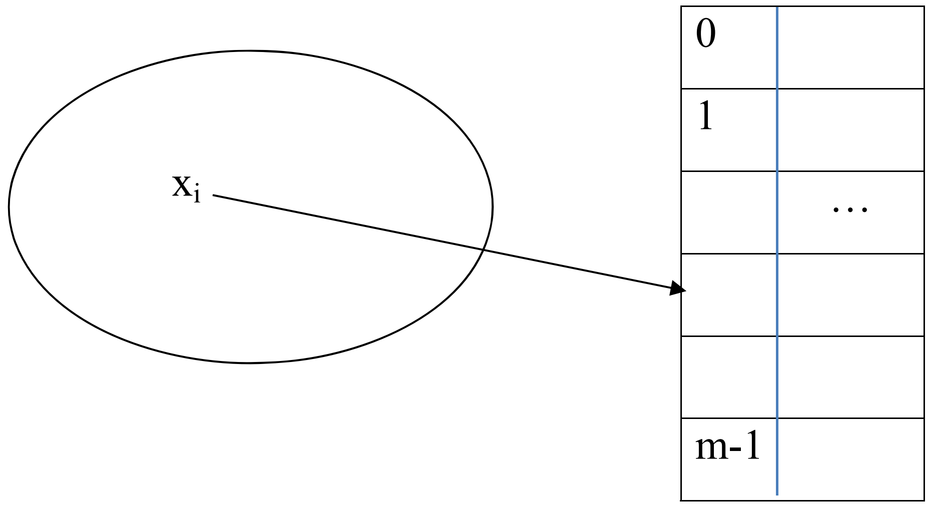
A hash table consists of

(1) A ***table***, which is an array B[0..m–1] of size m (tableSize), of m locations (buckets):
This table is used to store a set S of n data objects with keys $\{x_1, \ldots, x_n\}$, either internally or externally.

(2) A ***hash function*** $h: \{x_1, \ldots, x_n\} \rightarrow \{0, 1, \ldots, m-1\}$ with $h(x_i) = j$, $1 \leq i \leq n$, $0 \leq j \leq m-1$:
For any given data object with key $x_i$, the location $B[h(x_i)]$ will be used to store the given object $x_i$, or the address of $x_i$, if the location is not already occupied by another object.
**Ideal Case:** Distinct objects are hashed into distinct locations in B.
**Q:** What happens when two objects are being hashed into same location in B, resulting in a collision?
**A:** Need to implement a collision resolution scheme to store the new object that causes a collision.

(3) A ***collision resolution scheme***:
A method/operation used to store an item that is being hashed into an already occupied location, either by using an alternate location in the table, or by using an auxiliary data structure.

**Hashing:** A process in finding the locations in a hash table to store a given set of data objects.



Data set S with |S| = n                Hash Table B with m buckets

**Q:** How do we store the data objects in a hash table?

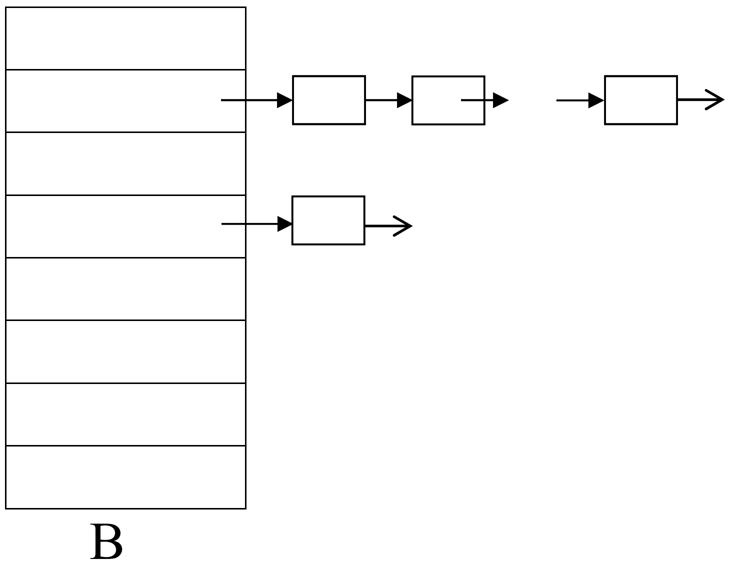
**Two Basic Hash Table Organizations:**

1. **Open (External) Hashing**:
   Each location/bucket can be used to store multiple data objects organized using an auxiliary data structure. Each bucket in the hash table will then hold a pointer pointing to one of these auxiliary data structures.
2. **Closed (Internal) Hashing**:
   Each location/bucket can be used to store at most one data object. Each bucket in the hash table will then store either an actual object or the address of the object. In closed hashing, in order to store all n objects in S using a hash table of size m, we must have $m \geq n$.

**Simple Hash Tables:**

I. **(Open) Hashing with (Separate) Chaining:**

Objects hashed into the same address/location in B are linked (chained) together using a linked structure. Hence, each location can contain a chain with many objects linked together.



**Basic Dictionary Operations:**

1. **Executing the *find(x)* Operation:**

   Given an object with key x. How do we search for x in B?

   ***Approach:***

   (i) Compute h(x) to find location B[h(x)] that may contain x.

   (ii) Traverse the linked structure at B[h(x)] sequentially, searching for object with key x.

**Factors Affecting the Performance of find(x) Operation:**

1. Evaluating the hash function h(x), and
2. Traversing the linked list; this depends on the size of the linked list.

**Characteristics of a "Good" Hash Function:**

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

(1) It can be computed in Θ(1) time, and

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

Define the ***load factor*** of a hash table $\lambda = \frac{n}{m}$.

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

**Unsuccessful search:**

1关于tablesize的哪一个链表，链表没有这个值，遍历完链表

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

**Successful search:**

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

**Remark:** Observe that for find operations in a hash table with chaining, $T_a(n) = \Theta(\lambda)$. As number of objects n increases, λ also increases; resulting in a decrease in efficiency of operations.

**2. Executing the *insert(x)* Operation:**

Given an object with key x. If the object is not already in the table B, how do we add the object with key x to B?

为什么增加在开头，而不是增加在结尾：节省时间，一开始find时候遍历所有元素，之后增加第一个在前面。

***Approach:***

(i) Compute h(x) to find location B[h(x)] that may contain x.

(ii) Traverse the linked structure at B[h(x)] sequentially, searching for object with key x. If not found, then insert the (new) object with key x at the beginning of the linked structure.

**3. Executing the *delete(x)* Operation:**

Given an object with key x. How do we delete the object with key x from B if exists?

***Approach:***

(i) Compute h(x) to find location B[h(x)] that may contain x.

(ii) Search the linked structure at B[h(x)] sequentially for the object with key x. If found, delete the object from the list.

**Complexity Analysis for insert(x) and delete(x):**

$T_a(n) = \Theta(\lambda)$. **(HW)**

**Observation:** When designing a hash table with chaining, if we choose the tableSize $m = \Theta(n)$, we have

$$T_a(n) = \Theta(\tfrac{n}{m})$$
$$= \Theta(1), \text{ which is the best possible!}$$

**Design a "Good" Hash Function:**

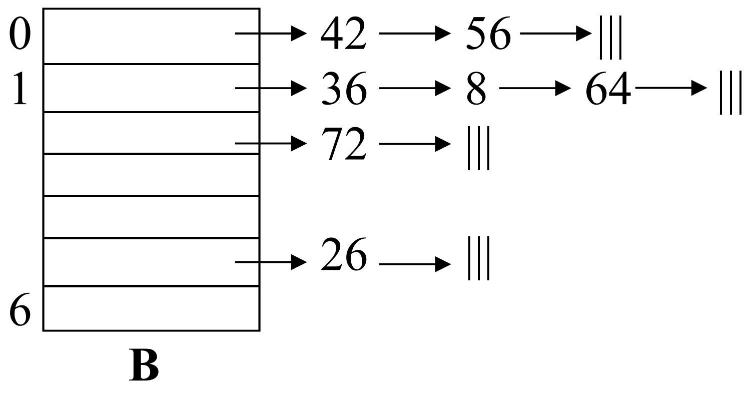
Very difficult, depending on the characteristics, structures of the keys and the table size m.

**A Simple Hash Function:**

Use division (mod m) function. If $h(x_i)$ has int value, define $h(x_i) = x_i \bmod m$, where m is chosen to be a prime. Observe that $0 \le h(x_i) \le m-1$.

**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 $h(x) = x \bmod m$.

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



**Remark:**

Insertions are done at the beginning of a chain.

**Possible Extension:**

Simple linked structure can be replaced by other linked structures such as doubly linked list. Similarly, the linked structure can be replaced by more advanced data structures such as a search tree so as to speed up the search.

**Advantages of Hashing with Chaining**

1. Simplicity (both conceptually and implementations).
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. Linked structure can degenerate into a single chain, resulting in $T_w(n) = \Theta(n)$.
2. Memory intensive: Need to implement/store pointers.
3. Slower speed: Indirect accessing data; need to follow pointers to data. Also, need to allocate and de-allocate dynamic memory.

## II. (Closed) Hashing with Open Addressing:

If each location/bucket is used to store at most one data object, then each bucket in the hash table will then store either an actual object or the address of the object.

**Q:** What if during insertion of x, we have h(x) = j and B[j] is already occupied by another object.

**A Simple Collision Resolution Scheme:**

Sequentially search B[j+1], B[j+2], …, B[m-1], B[0], B[1], …, B[j-1] to find the first available location to insert x. If no vacant location is found, report overflow. This collision resolution scheme is called ***Linear Probing***, which is a special case of Open Addressing Collision Resolution Scheme.

**Q:** What is Hashing with Open Addressing?

Given a hash function h, for some fixed integer k, define a sequence of k hash functions $\{h_0, h_1, \ldots, h_{k-1}\}$ such that
$h_i(x) = (h(x) + f_i) \bmod m$, with $0 \le i \le k-1$, $f_0 = 0$.

This set of k functions $\{f_0, f_1, \ldots, f_{k-1}\}$ is called ***collision resolution functions***.

For any given object with key x, we compute

$$\begin{aligned}
h_0(x) &= (h(x)+f_0) \bmod m, \ (h_0(x) = h(x) \bmod m)\\
h_1(x) &= (h(x)+f_1) \bmod m,\\
h_2(x) &= (h(x)+f_2) \bmod m,\\
&\ldots\\
h_{k-1}(x) &= (h(x)+f_{k-1}) \bmod m.
\end{aligned}$$

When a collison occurs, $B[h_0(x)]$ is occupied by another object, one would sequentially search $B[h_1(x)]$, $B[h_2(x)]$, …, until $B[h_{k-1}(x)]$ to find the first available empty bucket for inserting x. If all these k locations are occupied, x will not be inserted and an error message will be generated.

**Some Simple Open Addressing Schemes:**

**1. Hashing with Linear Probing:**

Define $f_i = i$, $\forall i$, $1 \leq i \leq k-1$, which is a family of *linear functions*, we have

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

**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.

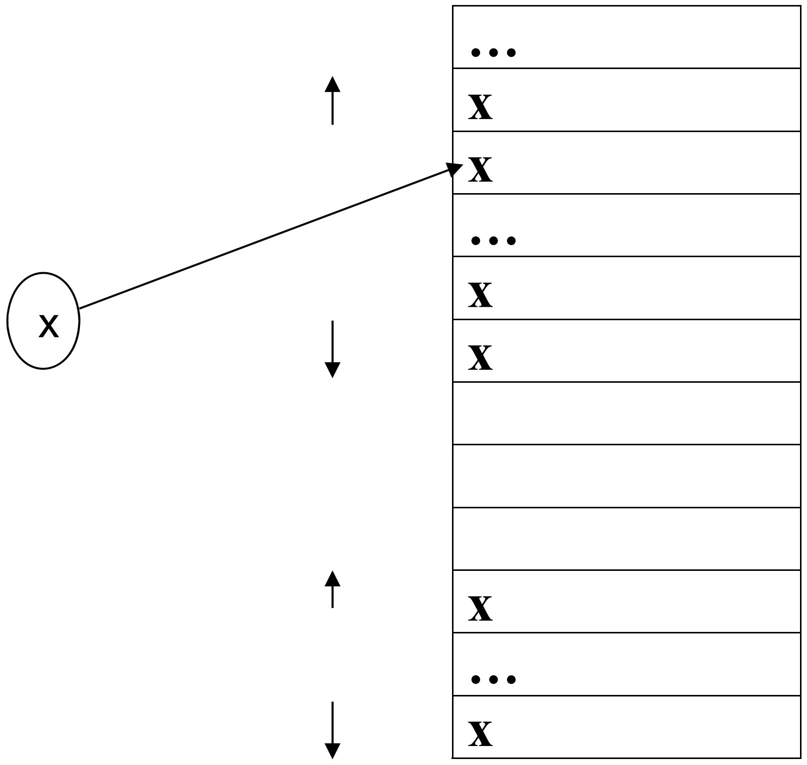
$$\begin{aligned} 64 \% 7 &= 1, \\ 26 \% 7 &= 5, \\ 56 \% 7 &= 0, \\ 72 \% 7 &= 2, \\ 8 \% 7 &= 1 \rightarrow 2 \rightarrow 3, \\ 36 \% 7 &= 1 \rightarrow 2 \rightarrow 3 \rightarrow 4, \\ 42 \% 7 &= 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6. \end{aligned}$$

**Hash table using linear probing**:

|  |
|---|
| 56 |
| 64 |
| 72 |
| 8 |
| 36 |
| 26 |
| 42 |

**Disadvantages in using Linear Probing:**

Closed hashing with linear probing may result in ***primary clustering***, which are blocks of occupied locations in B.



**Remedy:** Use quadratic probing to eliminate primary clustering.

2. **Hashing with Quadratic Probing:**
   Define $f_i = i^2$, $\forall i$, $1 \leq i \leq k\text{-}1$, which is a family of *quadratic functions*, we have

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

**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:**

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

**Hash table using quadratic probing:**

| |
|---|
| 56 |
| 64 |
| 72 |
| 8 |
| |
| 26 |
| |

**Problems with Closed Hashing:**

(1) Insertion may fail even though the table is not full.
(2) It may form secondary clustering.
(3) Problem in searching. Why?

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 $\rightarrow$ 2 $\rightarrow$ 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 | $\Rightarrow$ | Bucket is emptied by deletion; searching must continues. |
| flag = false | $\Rightarrow$ | 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 Quadratic Probing:**

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

**An Important Result in Hashing with Quadratic Probing:**

**Theorem:** Let m be a prime number > 3. If quadratic probing is used in a hash table of size m and, if 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 $\lambda$ 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*.

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

tableSize m (prime) $\leftrightarrow$ tableSize m* (prime $\approx$ 2m)

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

linear probe，当聚焦问题严重或者表接近满时，要搜索一个关键字，往往要逐个检查很多个无关项（先于和搜索关键字匹配的元素插入）。为了解决聚焦问题，提出了double hashing，其基本策略和线性探测法一项，唯一不同是：它不是检查冲突位置后的每一个位置，而是采用另一个散列函数产生一个固定的增量。（跳跃式检查和插入，减小聚焦大小）

**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 $\lambda$ to determine when to rehash.
3. When insertion becomes increasingly difficult or fails.
4. When deletion becomes increasingly difficult.

## 3. 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)

好处：插入的值不集中；不像linear和quad每一个重复的值间隔很短；需要花更多时间搜索无关项
坏处：很难delete

Observe that

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

**A Simple $h^+$ Function:**

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

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

**Addresses Computation:**

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

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

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

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

## Hash table using double hashing:

| |
|---|
| 56 |
| 64 |
| 72 |
| 8 |
| 42 |
| 26 |
| 36 |