Hash Table

A hash table is a **data structure** that allows efficient lookup, insertion, and deletion of key-value pairs. It works by mapping each key to a unique index in it.

Why we need a Hash Table?

Suppose we have an array/linked list. If we want to get a certain value, we must traverse the it.

0	1	2		…	500,000	…	…	1M

If the array/linked list is very long, the traversal will generate huge overhead.

Can we find this value in constant time without traversal?

Hash Table!

Hash Table

A hash table is a **data structure** that allows efficient lookup, insertion, and deletion of key-value pairs. It works by mapping each key to a unique index in it.

Direct Address Table

The keys are used as their indexes in the array to store the record by Direct Addressing

If we want to store the values {0, 1, 2, 3, 100001}

Create an array with the size of 100002, each value is stored under its corresponding index

0	1	2	3	…	100001
0	1	2	3		100001

Now, to find the value 1001, we don't need to traverse the array. Since arrays support random access, we can simply retrieve it using its index: A[1001] = 1001.

Operations take O(1) time, but waste too much space → **Hash Function** Map the universe of all keys into a hash table slots space efficiently

Hash Table

Hash Function

Determine how the table should store the data ---- Requires both key and the value

Key → Hash Function → Index

Contains the logic that determines what *index* the value will live at within the hash table

- Modulo Arithmetic: $H(k) = k \ mod \ h$
- Folding: $H(abc) \rightarrow (a + b + c) mod \ h$
- Mid-square: $H(k) = k^2 \ mod \ h$, the middle part of the result is used as the hash address
- Multiplicative Congruential Method: $H(k) = (a \times k) \ mod \ h$, a is a pseudo-random number

Collision

Ensure each possible key can find its index in the table, but multiple keys may be mapped to the same slots

- Closed Address Hashing
- Open Address Hashing

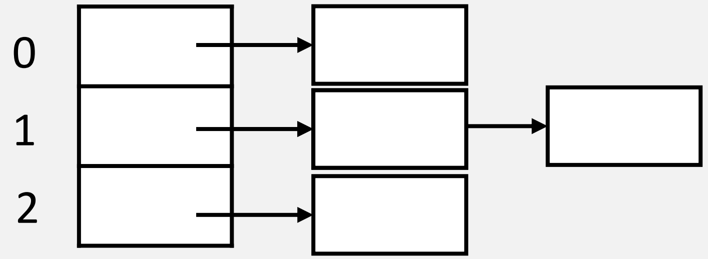
Hash Table

Closed Address Hashing (Chained Hashing)

- The address is **closed** (fixed). Each key has a corresponding fixed address

- If there are *n* records to store in the hash table, then $\alpha = \frac{n}{h}$ is the **load factor** of the hash table.

Chained Hash Table

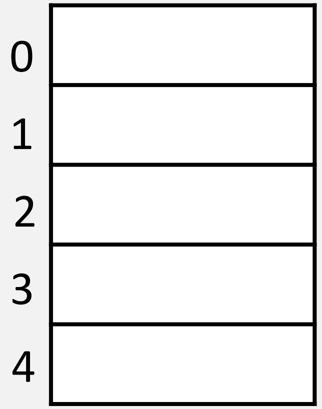


Collision: When multiple keys hash to the same index, they are stored in the linked list at that index.

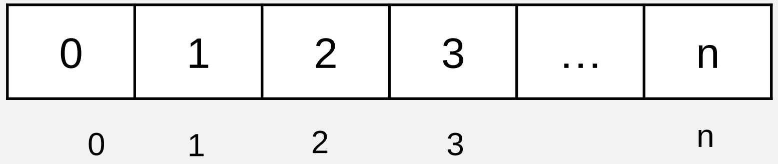
Open Address Hashing

The address is open (not fixed)
- Linear Probing
- Quadratic Probing
- Double Hashing

The load factor is never greater than 1



Linear Hash Table



Collision: If a collision occurs, the algorithm "probes" the next available slot until an empty one is found

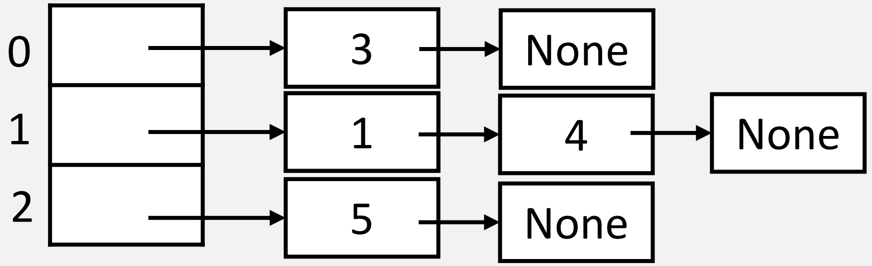
Q1 Closed Address Hashing

Implement a closed addressing hash table to perform insertion and key searching. The insertion may not have to insert at the end of the linklist. The function prototype is given below:

```
def hash_search(self, key):
def hash_insert(self, key):
```

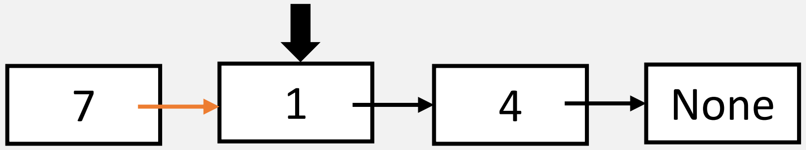
The default load factor is 3. The number of hash slots of the created hash table depends on the provided amount of data.

Insert {1, 3, 4, 5, 7}

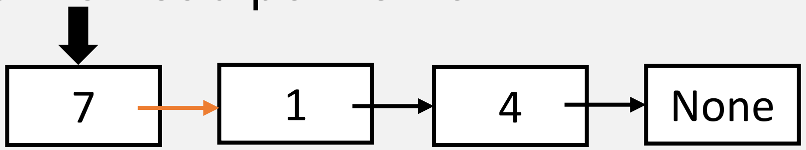


self.table[1] → 1

- Create a new node “7”
- Link “7”’s next to the linklist



- Update the head pointer to “7”



Q2 Open Address Hashing with Linear Probing

Implement an open addressing hash table with linear probing to perform insertion, deletion, and key searching. The function prototype is given below:

```
def hash_search(self, key):
def hash_insert(self, key):
def hash_delete(self, key):
```

$H(k,\ i)\ =(H'(k)\ +\ i)\ mod\ h$, where $H'(k)\ =\ k\ mod\ h$

0	5
1	1
2	2
3	3
4	8

size = h = 5 $\quad H(k,i)=(H'(k)+i)\ mod\ h$

Insert "8", $i\ =\ 0,\ \ H(8,\ 0)=(8+0)\ mod\ 5=3$

Use linear probing, $i\mathrel{+}=1$, $H(8,1)=(8+1)\ mod\ 5=4$

If $self.table[4]\ ==\ None$: $self.table[4]\ =\ Node(8)$

- Boundary of i
- Deleted or not?