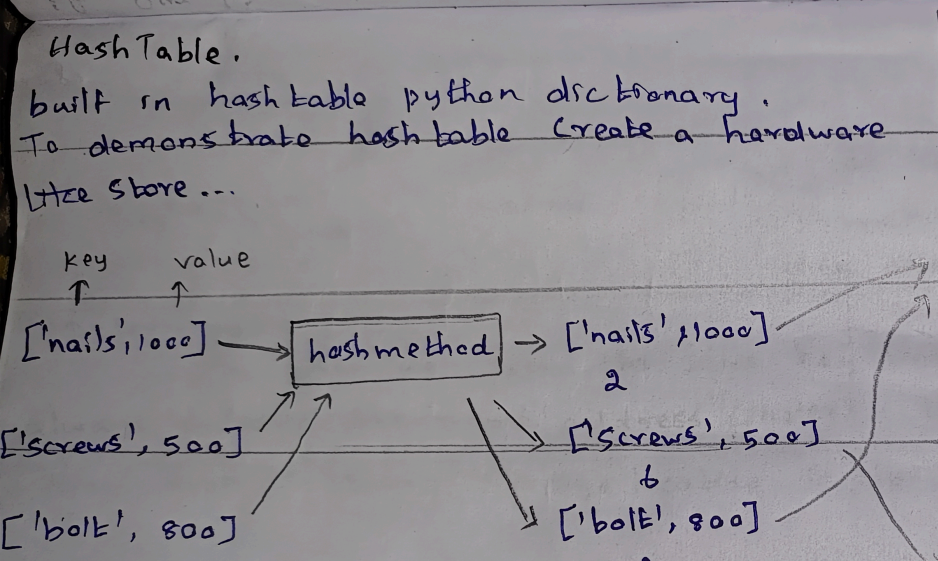
Hash Table.

built in hash table python dictionary.

To demonstrate hash table create a hardware store...



0	
1	
2	[['nails', 1000], ['bolt', 800]]
3	
4	
5	
6	[['screws', 500], ['bolts', 800]]

Here we ran through a key through this hash method and we got our address space (0 to 6) using that address as a index we can store the key value pairs.

hash itself.

i) its oneway, ii) Deterministic

i) its one way.

'key' → hash function → address → we use key to get address through the hash method we cant reverse that. using address through hash cant get the key

ii) Deterministic.

» when we pass the same key through hash function, it will always produce the same address (index). this ensures the given key will always map the same storage location in the table.

Hash table - collisions.

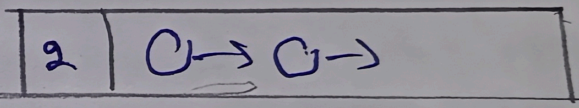
[['nails', 1000], ['nuts', 1200]] → when we put a multiple key value pairs in same address its called collision.

techniques for deal with collisions.

1. put them in the same address → separate chaining
   2 [[key, value], [key, value]]

2. if we already have a value in that address we need to go down until find a empty address and put the key value. → Linear probing (form of open addressing)

3. another way to do separate chaining, instead having a nested loop we can create a linked list.



constructor.

```
class HashTable:
    def __init__(self, size=7):
        self.data_map = [None] * size
```

default size.

represents,

0	None
1	None
2	None
3	None
4	None
5	None
6	None

```
    def __hash(self, key):
        my_hash = 0
        for letter in key:
            my_hash = (my_hash + ord(letter) * 23)
                      % len(self.data_map)

        return my_hash
```

hash method.

ord(letter) → short for ordinal it will give a ascii number for each char.

23 → (prime number). helps distribute values more evenly across table.

%.7 → when we divide any integer using 7 the reminder always 0 and 6 inculded and between.

↓

That exactly the address space.

```
def print_table(self):
    for i, value in enumerate(self.data_map):
        print(f"{i}: {value}")
```

→

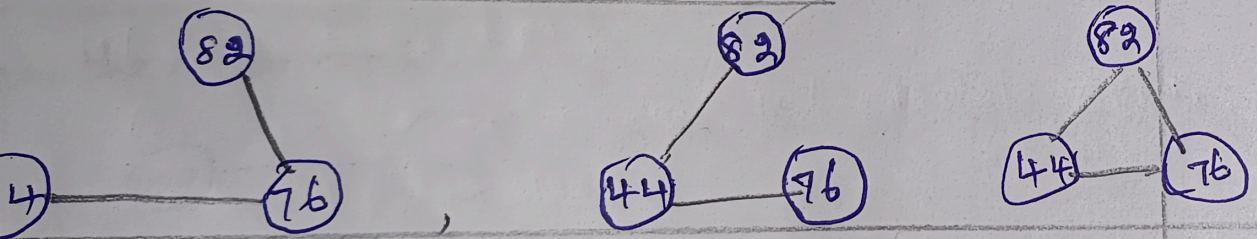
0: None
1: None
2: None
3: None

Graph.

linked list are the form of a tree, Tree is a form of a graph. therefore, a linked list is a form of a graph with the limitations that they can only point to one other node.

← vertex (or) node (proper way to say vertex here).

} → plural is vertices between the vertices we have a edge or connection (proper to say edge)



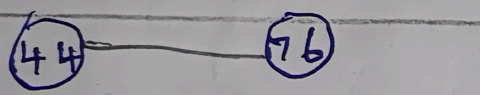
→ there is no limitations how many other vertices that a vertex can connect to

Graph.

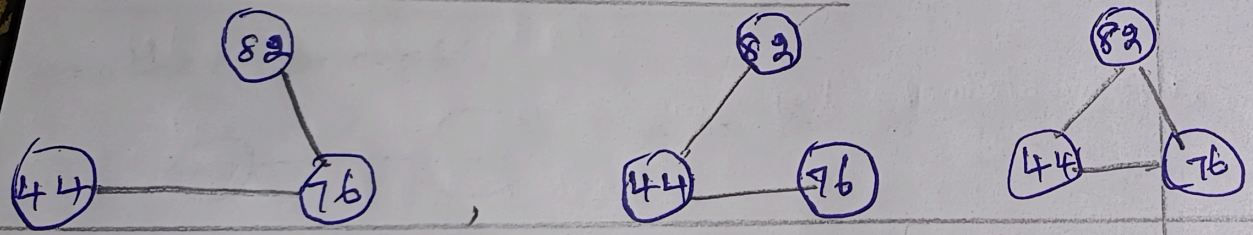
linked list are the form of a tree, Tree is a form of a graph.

Therefore, a linked list is a form of a graph with the limitations that they can only point to one other node.

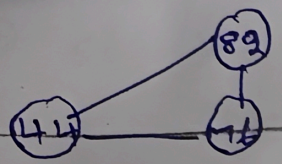
(44) ← vertex (or) node (proper way to say vertex here).



} → plural is vertices between the vertices we have a edge or connection (proper to say edge)



→ there is no limitations how many other vertices that a vertex can connect to

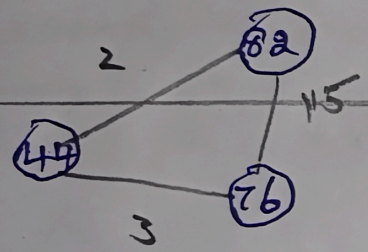


⇒ assume we need to go 76 to 89.

i) 76 to 89 → 1 (direct route).

ii) 76 to 44 and 44 to 89 → 2.

(*) But graph, we can have not always but can have a weighted edges.

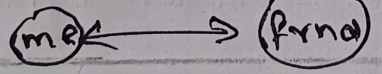


⇒ we can use this like maps app, if yoring gmaps the road between 76 and 82 has lot of traffic so the other way instead.

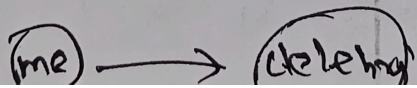
(*). ⇒ The edges can be weighted or non weighted.

another concept.

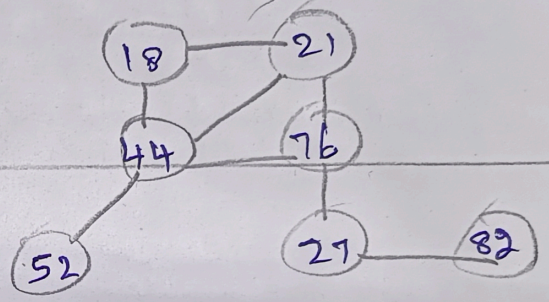
⇒ in fb we both friend follows each other (bi directional relationship)



⇒ in twitter i follow a person, but they dont (directional).

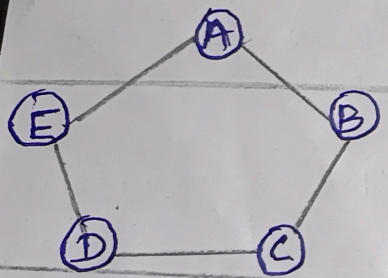


So the edges can be weighted (or) nonweighted
They can be directional or bidirectional.



when we think about graph it can look something like this

Adjacency matrix.
2 way to locally represent graph.
1) is 2. anjacency list.



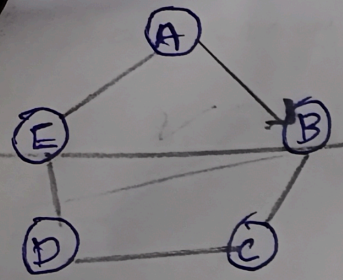
This axis is the item it has an edge with.

	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	0	0
C	0	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

This axis represent the vertex.

Starting looking at A has 2 edges b and e. so put the edge (e.g) 1 in the box
now B has edges with A and C and....
- A and point to A, B and point E to B....
so we always genna have the 45 degree line of zeros like

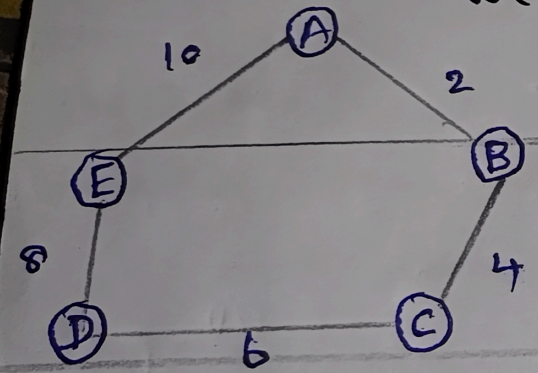
And if we have bidirectional matrix like there always have imirror image on each side of the 45° line.
(ex). only if this is bidirectional.



Here now its not bidirectional A pointing B ; but B not pointing A.

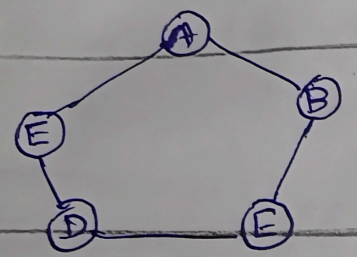
	A	B	C	D	E
A	0	1	0	0	1
B	0	0	1	0	0
C	0	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

weighted.



	A	B	C	D	E
A	0	2	0	0	10
B	2	0	4	0	0
C	0	4	0	6	0
D	0	0	6	0	8
D	10	0	0	8	0

Adjacency list.



⇒ represents

```
{ 'A' : ['B', 'E'],
  'B' : ['A', 'C'],
  'C' : ['B', 'D'],
  'D' : ['C', 'E'], 'D' : ['C', 'E'] }
```

Big O graph.

Adding item.

i) Adjacency matrix ⇒ $O(|V^2|)$

ii) Adjacency list ⇒ $O(1)$

Adding Edge : i) AM ⇒ $O(1)$

ii) AL ⇒ $O(1)$

removing edge :

i) AM → $O(1)$

ii) AL → $O(|E|)$ ⇒ $O(n)$ ← num of edges.

remove vertex.

i) AM ⇒ $O(|V|^2)$

ii) AL ⇒ $O(|V| + |E|)$

(*) matrix method not only storing problem for small graph assume a large million of vertex graph so its very ineffiient in storage perspective also storing 0, its not

(*) in list method we dont have store zero its more simpler and more effective.

Add vertex.

(A) ⇒ {'A' : []}

```
class Graph :
  def __init__ (self):
    self.adj_list = {}
  def print_graph (self):
    for vertex in self.adj_list :
      print(vertex, ' : ', self.adj_list[vertex])
  def add_vertex (self, vertex) :
    if vertex not in self.adj_list.keys() :
      self.adj_list[vertex] = []
      return True
    return False
```

```
my_graph = Graph()
my_graph.add_vertex ('A')
my_graph.print_graph()
```

A : []