Ουσιαστικα για το συγκεκρειμενο υποερωτημα βασιστητακμε και στο πως γινοταν create το index του btree. Υπαρχει ένα function το create_index() στο database.py οποτε ουσιαστικα θελαμε να κινηθουμε γυρω από αυτό.
Στο database.py ουσιαστικα προσθεσαμε την ακολουθη γραμμη κωδικα στην αρχη **from hash import ExtendibleHash** για να κανουμε import την κλαση ExtendibleHash από ένα νέο αρχειο που φτιαξαμε το **hash.py**.
Κατω από **if index_name == 'btree'** προσθεσαμε ένα νέο if το οποιο θα αφορουσε το **hash,** μιας και το hash είναι και αυτό ένα index. Το ακολουθως μερος του κωδικα προσθεσαμε είναι το ακολουθο:

psklriris@LAPTOP-FF9OU0L0: ~/miniDB/miniDB

```
            new_stack: string. The stack that will be used to replace the existing one.
        '''
        self.tables['meta_insert_stack']._update_rows(new_stack, 'indexes', f'table_name={table_name}')

    # indexes
    def create_index(self, index_name, table_name, index_type):
        '''
        Creates an index on a specified table with a given name.
        Important: An index can only be created on a primary key (the user does not specify the column).

        Args:
            table_name: string. Table name (must be part of database).
            index_name: string. Name of the created index.
        '''
        if self.tables[table_name].pk_idx is None:  # if no primary key, no index
            raise Exception('Cannot create index. Table has no primary key.')
        if index_name not in self.tables['meta_indexes'].column_by_name('index_name'):
            # currently only btree is supported. This can be changed by adding another if.
            if index_type == 'btree':
                logging.info('Creating Btree index.')
                # insert a record with the name of the index and the table on which it's created to the meta_indexes ta
                self.tables['meta_indexes']._insert([table_name, index_name])
                # crate the actual index
                self._construct_index(table_name, index_name)
                self.save_database()
            if index_type== 'hash':
                hash_table=ExtendibleHash(10)
                hash_name=index_name
                self.tables['meta_indexes']._insert([hash_table,hash_name])
                self._save_index(hash_table,hash_name)
                self.save_database()
                print(self._load_idx(hash_name))
        else:
            raise Exception('Cannot create index. Another index with the same name already exists.')

    def _construct_index(self, table_name, index_name):
        '''
```

Εδώ ορισαμε ουσιαστικα το hash_table στο οποιο ουσιαστικα θελαμε να ορισαμε το νέο table το οποιο θα εσπαγε το table σε buckets μεσω των dictionaries.
Στην συνεχεια απλα μετονομασαμε και εκχωρησαμε το index_name μεσα σε ένα νέο variable το hash_name. Επειτα θελαμε να εχωρησουμε το hash_table αυτό και το αντιστοιχο του hash_name στα meta_indexes. Χρησιμοποιησαμε το function _save_index() για να κανουμε save το index αυτό.

Επειτα στο hash.py δημιουργησαμε μια κλαση την ExtendibleHash όπως προαναφεραμε πανω και φαινεται από το ακολουθο screenshot.

psklIris@LAPTOP-FF9OU0L0: ~/miniDB/miniDB

```
class ExtendibleHash:
    def __init__(self,size):
        self.size = size
        self.dictionary = {}

    def implemet_hash_function(self,key):
        return hash(key) & ((1 << self.size) - 1)

    def get_function_for_bucket(self,key):
        bckt_idx = self.implement_hash_function(key)
        if bckt_idx in self.dictionary:
            return self.dictionary[bckt_idx]
        else:
            return None
~
```

Αρχικα δημιουργησαμε την _init_ function (constructor της python) ορισαμε σαν παραμετρους το size για το μεγεθος που θα εχει το bucket , σε μορφη dictionary και αντιστοιχα τους δωσαμε τις τιμες που θα παρουν οριζοντας το self.size=size το μεγεθος του bucket και ένα default dictionary το self.dictionary={}.
Στην συνεχεια ορισαμε την implement_hash_function οπου key το ονομα του κλειδιου κάθε πινακα που θελουμε να εφαρμοσαμε και να δημιουργησουμε ένα hash index. Στην συνεχεια εκει χρησιμοποιουμε την ηδη υπαρχουσα hash function που εχει η python μεσω της λειτουργιας των bit για να αντιστοιχησουμε το κλειδι του κάθε πινακα με ένα bucket, εφαρμοζοντας ένα λογικο and μεταξυ αυτων.
Η 3^{η} function αφορα μια μεθοδο get στην οποια δινουμε το κλειδι του πινακα σαν παραμετρο και εξεταζει να δει ουσιαστικα αν υπαρχει το εκχωρημενο dictionary αν υπαρχει επιστρεφει none αλλιως το εκχωρει.