

Assignment 11.3 Ai Assisted Coding

Ht.no: 2303A51305

Batch: 05

Task 1:

Smart Contact Manager (Arrays & Linked Lists)

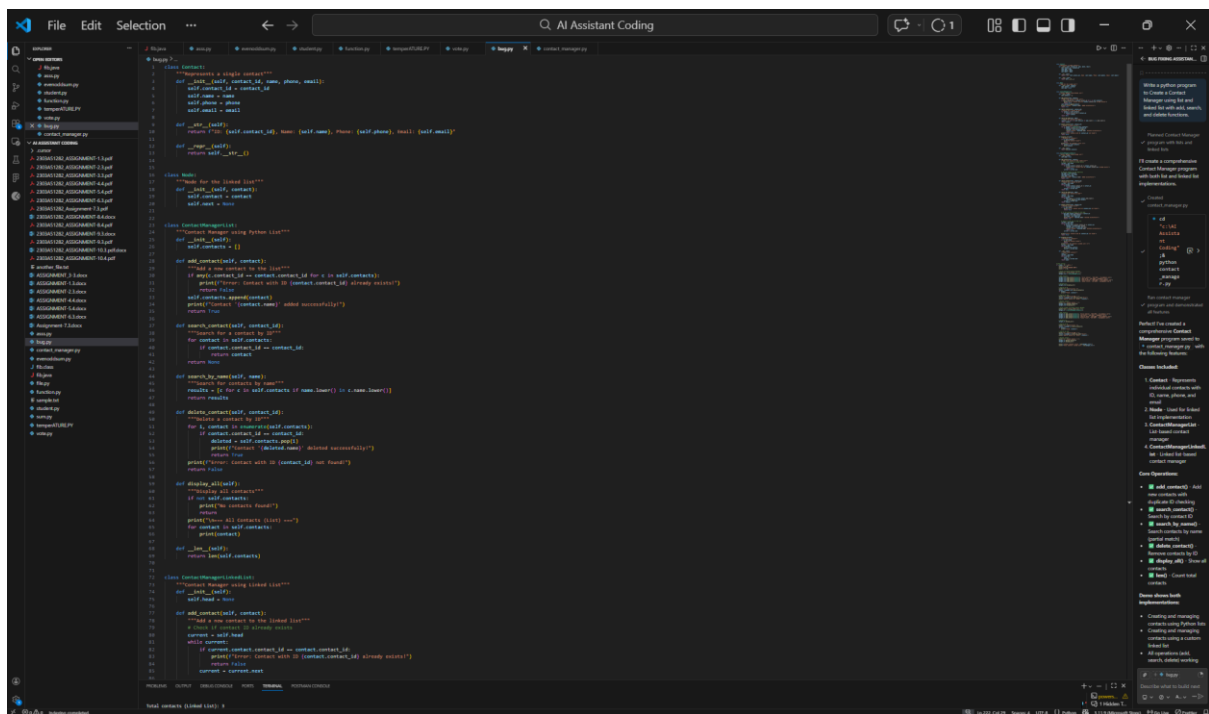
Scenario

SR University's student club requires a simple Contact Manager Application to store members' names and phone numbers. The system should support efficient addition, searching, and deletion of contacts.

Prompt:

Write a python program to Create a Contact Manager using list and linked list with add, search, and delete functions.

Code:

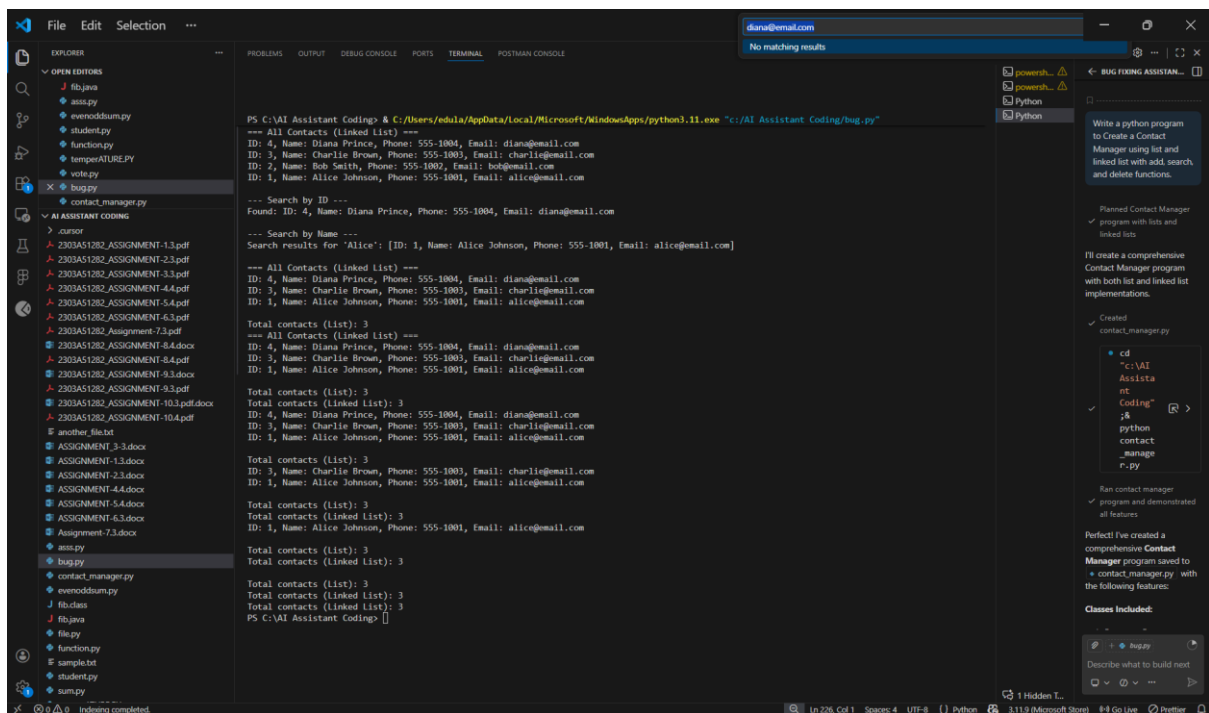


```
1 class Contact:
2     """Represents a single contact"""
3     def __init__(self, contact_id, name, phone, email):
4         self.contact_id = contact_id
5         self.name = name
6         self.phone = phone
7         self.email = email
8
9     def __str__(self):
10         return f"({self.contact_id}, {self.name}, {self.phone}, {self.email})"
11
12 class ContactManager:
13     """Contact Manager using linked list"""
14     def __init__(self):
15         self.head = None
16
17     def add_contact(self, contact):
18         """Add a new contact to the linked list"""
19         if not self.head:
20             self.head = contact
21             return
22         current = self.head
23         while current.next:
24             current = current.next
25         current.next = contact
26
27     def search_contact(self, contact_id):
28         """Search for a contact by ID"""
29         current = self.head
30         while current:
31             if current.contact_id == contact_id:
32                 return current
33             current = current.next
34         return None
35
36     def delete_contact(self, contact_id):
37         """Delete a contact by ID"""
38         if not self.head:
39             return
40         if self.head.contact_id == contact_id:
41             self.head = self.head.next
42             return
43         current = self.head
44         while current.next:
45             if current.next.contact_id == contact_id:
46                 current.next = current.next.next
47                 return
48             current = current.next
49         return
50
51     def display_all(self):
52         """Display all contacts"""
53         if not self.head:
54             print("No contacts found")
55             return
56         current = self.head
57         while current:
58             print(f"({current.contact_id}, {current.name}, {current.phone}, {current.email})")
59             current = current.next
60
61     def __str__(self):
62         return str(self.contacts)
63
64 class ContactManagerLinkedList:
65     """Contact Manager using linked list"""
66     def __init__(self):
67         self.head = None
68
69     def add_contact(self, contact):
70         """Add a new contact to the linked list"""
71         if not self.head:
72             self.head = contact
73             return
74         current = self.head
75         while current.next:
76             current = current.next
77         current.next = contact
78
79     def search_contact(self, contact_id):
80         """Search for a contact by ID"""
81         current = self.head
82         while current:
83             if current.contact_id == contact_id:
84                 return current
85             current = current.next
86         return None
87
88     def delete_contact(self, contact_id):
89         """Delete a contact by ID"""
90         if not self.head:
91             return
92         if self.head.contact_id == contact_id:
93             self.head = self.head.next
94             return
95         current = self.head
96         while current.next:
97             if current.next.contact_id == contact_id:
98                 current.next = current.next.next
99                 return
100            current = current.next
101        return
102
103     def display_all(self):
104         """Display all contacts"""
105         if not self.head:
106             print("No contacts found")
107             return
108         current = self.head
109         while current:
110             print(f"({current.contact_id}, {current.name}, {current.phone}, {current.email})")
111             current = current.next
112
113     def __str__(self):
114         return str(self.contacts)
115
116 if __name__ == "__main__":
117     manager = ContactManager()
118     manager.add_contact(Contact(1, "John Doe", "1234567890", "john.doe@example.com"))
119     manager.add_contact(Contact(2, "Jane Smith", "9876543210", "jane.smith@example.com"))
120     manager.add_contact(Contact(3, "Bob Johnson", "5555555555", "bob.johnson@example.com"))
121     manager.add_contact(Contact(4, "Alice Brown", "1111111111", "alice.brown@example.com"))
122     manager.add_contact(Contact(5, "Charlie Davis", "2222222222", "charlie.davis@example.com"))
123     manager.add_contact(Contact(6, "Diana Prince", "3333333333", "diana.prince@example.com"))
124     manager.add_contact(Contact(7, "Ethan Hunt", "4444444444", "ethan.hunt@example.com"))
125     manager.add_contact(Contact(8, "Fiona Glenanne", "6666666666", "fiona.glenanne@example.com"))
126     manager.add_contact(Contact(9, "Giles Kane", "7777777777", "giles.kane@example.com"))
127     manager.add_contact(Contact(10, "Hiro Nakamura", "8888888888", "hiro.nakamura@example.com"))
128     manager.add_contact(Contact(11, "Ivy Fong", "9999999999", "ivy.fong@example.com"))
129     manager.add_contact(Contact(12, "James Earl Ray", "1010101010", "james.earl.ray@example.com"))
130     manager.add_contact(Contact(13, "Katherine Tegen", "1111111111", "katherine.tegen@example.com"))
131     manager.add_contact(Contact(14, "Liam Neeson", "1212121212", "liam.neeson@example.com"))
132     manager.add_contact(Contact(15, "Milla Jovovich", "1313131313", "milla.jovovich@example.com"))
133     manager.add_contact(Contact(16, "Natalie Portman", "1414141414", "natalie.portman@example.com"))
134     manager.add_contact(Contact(17, "Oscar Isaac", "1515151515", "oscar.isaac@example.com"))
135     manager.add_contact(Contact(18, "Penelope Cruz", "1616161616", "penelope.cruz@example.com"))
136     manager.add_contact(Contact(19, "Quentin Tarantino", "1717171717", "quentin.tarantino@example.com"))
137     manager.add_contact(Contact(20, "Ryan Murphy", "1818181818", "ryan.murphy@example.com"))
138     manager.add_contact(Contact(21, "Saoirse Ronan", "1919191919", "saoirse.ronan@example.com"))
139     manager.add_contact(Contact(22, "Tina Turner", "2020202020", "tina.turner@example.com"))
140     manager.add_contact(Contact(23, "Uma Thurman", "2121212121", "uma.thurman@example.com"))
141     manager.add_contact(Contact(24, "Viggo Mortensen", "2222222222", "viggo.mortensen@example.com"))
142     manager.add_contact(Contact(25, "Wendie Renai", "2323232323", "wendie.renai@example.com"))
143     manager.add_contact(Contact(26, "Xosha Roquemore", "2424242424", "xosha.roquemore@example.com"))
144     manager.add_contact(Contact(27, "Yara Shahidi", "2525252525", "yara.shahidi@example.com"))
145     manager.add_contact(Contact(28, "Zoe Lister-Jones", "2626262626", "zoe.lister-jones@example.com"))
146     manager.add_contact(Contact(29, "Adam Driver", "2727272727", "adam.driver@example.com"))
147     manager.add_contact(Contact(30, "Brie Larson", "2828282828", "brie.larson@example.com"))
148     manager.add_contact(Contact(31, "Cobie Smulders", "2929292929", "cobie.smulders@example.com"))
149     manager.add_contact(Contact(32, "Dakota Fanning", "3030303030", "dakota.fanning@example.com"))
150     manager.add_contact(Contact(33, "Ewan McGregor", "3131313131", "ewan.mcgregor@example.com"))
151     manager.add_contact(Contact(34, "Florence Pugh", "3232323232", "florence.pugh@example.com"))
152     manager.add_contact(Contact(35, "Gael García Bernal", "3333333333", "gael.garcia.bernal@example.com"))
153     manager.add_contact(Contact(36, "Hailee Steinfeld", "3434343434", "hailee.steinfeld@example.com"))
154     manager.add_contact(Contact(37, "Idina Menzel", "3535353535", "idina.menzel@example.com"))
155     manager.add_contact(Contact(38, "Jacob Tremblay", "3636363636", "jacob.tremblay@example.com"))
156     manager.add_contact(Contact(39, "Kaiti Herewini", "3737373737", "kaiti.herewini@example.com"))
157     manager.add_contact(Contact(40, "Liam Neeson", "3838383838", "liam.neeson@example.com"))
158     manager.add_contact(Contact(41, "Milla Jovovich", "3939393939", "milla.jovovich@example.com"))
159     manager.add_contact(Contact(42, "Natalie Portman", "4040404040", "natalie.portman@example.com"))
160     manager.add_contact(Contact(43, "Oscar Isaac", "4141414141", "oscar.isaac@example.com"))
161     manager.add_contact(Contact(44, "Penelope Cruz", "4242424242", "penelope.cruz@example.com"))
162     manager.add_contact(Contact(45, "Quentin Tarantino", "4343434343", "quentin.tarantino@example.com"))
163     manager.add_contact(Contact(46, "Ryan Murphy", "4444444444", "ryan.murphy@example.com"))
164     manager.add_contact(Contact(47, "Saoirse Ronan", "4545454545", "saoirse.ronan@example.com"))
165     manager.add_contact(Contact(48, "Tina Turner", "4646464646", "tina.turner@example.com"))
166     manager.add_contact(Contact(49, "Uma Thurman", "4747474747", "uma.thurman@example.com"))
167     manager.add_contact(Contact(50, "Viggo Mortensen", "4848484848", "viggo.mortensen@example.com"))
168     manager.add_contact(Contact(51, "Wendie Renai", "4949494949", "wendie.renai@example.com"))
169     manager.add_contact(Contact(52, "Xosha Roquemore", "5050505050", "xosha.roquemore@example.com"))
170     manager.add_contact(Contact(53, "Yara Shahidi", "5151515151", "yara.shahidi@example.com"))
171     manager.add_contact(Contact(54, "Zoe Lister-Jones", "5252525252", "zoe.lister-jones@example.com"))
172     manager.add_contact(Contact(55, "Adam Driver", "5353535353", "adam.driver@example.com"))
173     manager.add_contact(Contact(56, "Brie Larson", "5454545454", "brie.larson@example.com"))
174     manager.add_contact(Contact(57, "Cobie Smulders", "5555555555", "cobie.smulders@example.com"))
175     manager.add_contact(Contact(58, "Dakota Fanning", "5656565656", "dakota.fanning@example.com"))
176     manager.add_contact(Contact(59, "Ewan McGregor", "5757575757", "ewan.mcgregor@example.com"))
177     manager.add_contact(Contact(60, "Florence Pugh", "5858585858", "florence.pugh@example.com"))
178     manager.add_contact(Contact(61, "Gael García Bernal", "5959595959", "gael.garcia.bernal@example.com"))
179     manager.add_contact(Contact(62, "Hailee Steinfeld", "6060606060", "hailee.steinfeld@example.com"))
180     manager.add_contact(Contact(63, "Idina Menzel", "6161616161", "idina.menzel@example.com"))
181     manager.add_contact(Contact(64, "Jacob Tremblay", "6262626262", "jacob.tremblay@example.com"))
182     manager.add_contact(Contact(65, "Kaiti Herewini", "6363636363", "kaiti.herewini@example.com"))
183     manager.add_contact(Contact(66, "Liam Neeson", "6464646464", "liam.neeson@example.com"))
184     manager.add_contact(Contact(67, "Milla Jovovich", "6565656565", "milla.jovovich@example.com"))
185     manager.add_contact(Contact(68, "Natalie Portman", "6666666666", "natalie.portman@example.com"))
186     manager.add_contact(Contact(69, "Oscar Isaac", "6767676767", "oscar.isaac@example.com"))
187     manager.add_contact(Contact(70, "Penelope Cruz", "6868686868", "penelope.cruz@example.com"))
188     manager.add_contact(Contact(71, "Quentin Tarantino", "6969696969", "quentin.tarantino@example.com"))
189     manager.add_contact(Contact(72, "Ryan Murphy", "7070707070", "ryan.murphy@example.com"))
190     manager.add_contact(Contact(73, "Saoirse Ronan", "7171717171", "saoirse.ronan@example.com"))
191     manager.add_contact(Contact(74, "Tina Turner", "7272727272", "tina.turner@example.com"))
192     manager.add_contact(Contact(75, "Uma Thurman", "7373737373", "uma.thurman@example.com"))
193     manager.add_contact(Contact(76, "Viggo Mortensen", "7474747474", "viggo.mortensen@example.com"))
194     manager.add_contact(Contact(77, "Wendie Renai", "7575757575", "wendie.renai@example.com"))
195     manager.add_contact(Contact(78, "Xosha Roquemore", "7676767676", "xosha.roquemore@example.com"))
196     manager.add_contact(Contact(79, "Yara Shahidi", "7777777777", "yara.shahidi@example.com"))
197     manager.add_contact(Contact(80, "Zoe Lister-Jones", "7878787878", "zoe.lister-jones@example.com"))
198     manager.add_contact(Contact(81, "Adam Driver", "7979797979", "adam.driver@example.com"))
199     manager.add_contact(Contact(82, "Brie Larson", "8080808080", "brie.larson@example.com"))
200     manager.add_contact(Contact(83, "Cobie Smulders", "8181818181", "cobie.smulders@example.com"))
201     manager.add_contact(Contact(84, "Dakota Fanning", "8282828282", "dakota.fanning@example.com"))
202     manager.add_contact(Contact(85, "Ewan McGregor", "8383838383", "ewan.mcgregor@example.com"))
203     manager.add_contact(Contact(86, "Florence Pugh", "8484848484", "florence.pugh@example.com"))
204     manager.add_contact(Contact(87, "Gael García Bernal", "8585858585", "gael.garcia.bernal@example.com"))
205     manager.add_contact(Contact(88, "Hailee Steinfeld", "8686868686", "hailee.steinfeld@example.com"))
206     manager.add_contact(Contact(89, "Idina Menzel", "8787878787", "idina.menzel@example.com"))
207     manager.add_contact(Contact(90, "Jacob Tremblay", "8888888888", "jacob.tremblay@example.com"))
208     manager.add_contact(Contact(91, "Kaiti Herewini", "8989898989", "kaiti.herewini@example.com"))
209     manager.add_contact(Contact(92, "Liam Neeson", "9090909090", "liam.neeson@example.com"))
210     manager.add_contact(Contact(93, "Milla Jovovich", "9191919191", "milla.jovovich@example.com"))
211     manager.add_contact(Contact(94, "Natalie Portman", "9292929292", "natalie.portman@example.com"))
212     manager.add_contact(Contact(95, "Oscar Isaac", "9393939393", "oscar.isaac@example.com"))
213     manager.add_contact(Contact(96, "Penelope Cruz", "9494949494", "penelope.cruz@example.com"))
214     manager.add_contact(Contact(97, "Quentin Tarantino", "9595959595", "quentin.tarantino@example.com"))
215     manager.add_contact(Contact(98, "Ryan Murphy", "9696969696", "ryan.murphy@example.com"))
216     manager.add_contact(Contact(99, "Saoirse Ronan", "9797979797", "saoirse.ronan@example.com"))
217     manager.add_contact(Contact(100, "Tina Turner", "9898989898", "tina.turner@example.com"))
218     manager.add_contact(Contact(101, "Uma Thurman", "9999999999", "uma.thurman@example.com"))
219     manager.add_contact(Contact(102, "Viggo Mortensen", "1010101010", "viggo.mortensen@example.com"))
220     manager.add_contact(Contact(103, "Wendie Renai", "1111111111", "wendie.renai@example.com"))
221     manager.add_contact(Contact(104, "Xosha Roquemore", "1212121212", "xosha.roquemore@example.com"))
222     manager.add_contact(Contact(105, "Yara Shahidi", "1313131313", "yara.shahidi@example.com"))
223     manager.add_contact(Contact(106, "Zoe Lister-Jones", "1414141414", "zoe.lister-jones@example.com"))
224     manager.add_contact(Contact(107, "Adam Driver", "1515151515", "adam.driver@example.com"))
225     manager.add_contact(Contact(108, "Brie Larson", "1616161616", "brie.larson@example.com"))
226     manager.add_contact(Contact(109, "Cobie Smulders", "1717171717", "cobie.smulders@example.com"))
227     manager.add_contact(Contact(110, "Dakota Fanning", "1818181818", "dakota.fanning@example.com"))
228     manager.add_contact(Contact(111, "Ewan McGregor", "1919191919", "ewan.mcgregor@example.com"))
229     manager.add_contact(Contact(112, "Florence Pugh", "2020202020", "florence.pugh@example.com"))
230     manager.add_contact(Contact(113, "Gael García Bernal", "2121212121", "gael.garcia.bernal@example.com"))
231     manager.add_contact(Contact(114, "Hailee Steinfeld", "2222222222", "hailee.steinfeld@example.com"))
232     manager.add_contact(Contact(115, "Idina Menzel", "2323232323", "idina.menzel@example.com"))
233     manager.add_contact(Contact(116, "Jacob Tremblay", "2424242424", "jacob.tremblay@example.com"))
234     manager.add_contact(Contact(117, "Kaiti Herewini", "2525252525", "kaiti.herewini@example.com"))
235     manager.add_contact(Contact(118, "Liam Neeson", "2626262626", "liam.neeson@example.com"))
236     manager.add_contact(Contact(119, "Milla Jovovich", "2727272727", "milla.jovovich@example.com"))
237     manager.add_contact(Contact(120, "Natalie Portman", "2828282828", "natalie.portman@example.com"))
238     manager.add_contact(Contact(121, "Oscar Isaac", "2929292929", "oscar.isaac@example.com"))
239     manager.add_contact(Contact(122, "Penelope Cruz", "3030303030", "penelope.cruz@example.com"))
240     manager.add_contact(Contact(123, "Quentin Tarantino", "3131313131", "quentin.tarantino@example.com"))
241     manager.add_contact(Contact(124, "Ryan Murphy", "3232323232", "ryan.murphy@example.com"))
242     manager.add_contact(Contact(125, "Saoirse Ronan", "3333333333", "saoirse.ronan@example.com"))
243     manager.add_contact(Contact(126, "Tina Turner", "3434343434", "tina.turner@example.com"))
244     manager.add_contact(Contact(127, "Uma Thurman", "3535353535", "uma.thurman@example.com"))
245     manager.add_contact(Contact(128, "Viggo Mortensen", "3636363636", "viggo.mortensen@example.com"))
246     manager.add_contact(Contact(129, "Wendie Renai", "3737373737", "wendie.renai@example.com"))
247     manager.add_contact(Contact(130, "Xosha Roquemore", "3838383838", "xosha.roquemore@example.com"))
248     manager.add_contact(Contact(131, "Yara Shahidi", "3939393939", "yara.shahidi@example.com"))
249     manager.add_contact(Contact(132, "Zoe Lister-Jones", "4040404040", "zoe.lister-jones@example.com"))
250     manager.add_contact(Contact(133, "Adam Driver", "4141414141", "adam.driver@example.com"))
251     manager.add_contact(Contact(134, "Brie Larson", "4242424242", "brie.larson@example.com"))
252     manager.add_contact(Contact(135, "Cobie Smulders", "4343434343", "cobie.smulders@example.com"))
253     manager.add_contact(Contact(136, "Dakota Fanning", "4444444444", "dakota.fanning@example.com"))
254     manager.add_contact(Contact(137, "Ewan McGregor", "4545454545", "ewan.mcgregor@example.com"))
255     manager.add_contact(Contact(138, "Florence Pugh", "4646464646", "florence.pugh@example.com"))
256     manager.add_contact(Contact(139, "Gael García Bernal", "4747474747", "gael.garcia.bernal@example.com"))
257     manager.add_contact(Contact(140, "Hailee Steinfeld", "4848484848", "hailee.steinfeld@example.com"))
258     manager.add_contact(Contact(141, "Idina Menzel", "4949494949", "idina.menzel@example.com"))
259     manager.add_contact(Contact(142, "Jacob Tremblay", "5050505050", "jacob.tremblay@example.com"))
260     manager.add_contact(Contact(143, "Kaiti Herewini", "5151515151", "kaiti.herewini@example.com"))
261     manager.add_contact(Contact(144, "Liam Neeson", "5252525252", "liam.neeson@example.com"))
262     manager.add_contact(Contact(145, "Milla Jovovich", "5353535353", "milla.jovovich@example.com"))
263     manager.add_contact(Contact(146, "Natalie Portman", "5454545454", "natalie.portman@example.com"))
264     manager.add_contact(Contact(147, "Oscar Isaac", "5555555555", "oscar.isaac@example.com"))
265     manager.add_contact(Contact(148, "Penelope Cruz", "5656565656", "penelope.cruz@example.com"))
266     manager.add_contact(Contact(149, "Quentin Tarantino", "5757575757", "quentin.tarantino@example.com"))
267     manager.add_contact(Contact(150, "Ryan Murphy", "5858585858", "ryan.murphy@example.com"))
268     manager.add_contact(Contact(151, "Saoirse Ronan", "5959595959", "saoirse.ronan@example.com"))
269     manager.add_contact(Contact(152, "Tina Turner", "6060606060", "tina.turner@example.com"))
270     manager.add_contact(Contact(153, "Uma Thurman", "6161616161", "uma.thurman@example.com"))
271     manager.add_contact(Contact(154, "Viggo Mortensen", "6262626262", "viggo.mortensen@example.com"))
272     manager.add_contact(Contact(155, "Wendie Renai", "6363636363", "wendie.renai@example.com"))
273     manager.add_contact(Contact(156, "Xosha Roquemore", "6464646464", "xosha.roquemore@example.com"))
274     manager.add_contact(Contact(157, "Yara Shahidi", "6565656565", "yara.shahidi@example.com"))
275     manager.add_contact(Contact(158, "Zoe Lister-Jones", "6666666666", "zoe.lister-jones@example.com"))
276     manager.add_contact(Contact(159, "Adam Driver", "6767676767", "adam.driver@example.com"))
277     manager.add_contact(Contact(160, "Brie Larson", "6868686868", "brie.larson@example.com"))
278     manager.add_contact(Contact(161, "Cobie Smulders", "6969696969", "cobie.smulders@example.com"))
279     manager.add_contact(Contact(162, "Dakota Fanning", "7070707070", "dakota.fanning@example.com"))
280     manager.add_contact(Contact(163, "Ewan McGregor", "7171717171", "ewan.mcgregor@example.com"))
281     manager.add_contact(Contact(164, "Florence Pugh", "7272727272", "florence.pugh@example.com"))
282     manager.add_contact(Contact(165, "Gael García Bernal", "7373737373", "gael.garcia.bernal@example.com"))
283     manager.add_contact(Contact(166, "Hailee Steinfeld", "7474747474", "hailee.steinfeld@example.com"))
284     manager.add_contact(Contact(167, "Idina Menzel", "7575757575", "idina.menzel@example.com"))
285     manager.add_contact(Contact(168, "Jacob Tremblay", "7676767676", "jacob.tremblay@example.com"))
286     manager.add_contact(Contact(169, "Kaiti Herewini", "7777777777", "kaiti.herewini@example.com"))
287     manager.add_contact(Contact(170, "Liam Neeson", "7878787878", "liam.neeson@example.com"))
288     manager.add_contact(Contact(171, "Milla Jovovich", "7979797979", "milla.jovovich@example.com"))
289     manager.add_contact(Contact(172, "Natalie Portman", "8080808080", "natalie.portman@example.com"))
290     manager.add_contact(Contact(173, "Oscar Isaac", "8181818181", "oscar.isaac@example.com"))
291     manager.add_contact(Contact(174, "Penelope Cruz", "8282828282", "penelope.cruz@example.com"))
292     manager.add_contact(Contact(175, "Quentin Tarantino", "8383838383", "quentin.tarantino@example.com"))
293     manager.add_contact(Contact(176, "Ryan Murphy", "8484848484", "ryan.murphy@example.com"))
294     manager.add_contact(Contact(177, "Saoirse Ronan", "8585858585", "saoirse.ronan@example.com"))
295     manager.add_contact(Contact(178, "Tina Turner", "8686868686", "tina.turner@example.com"))
296     manager.add_contact(Contact(179, "Uma Thurman", "8787878787", "uma.thurman@example.com"))
297     manager.add_contact(Contact(180, "Viggo Mortensen", "8888888888", "viggo.mortensen@example.com"))
298     manager.add_contact(Contact(181, "Wendie Renai", "8989898989", "wendie.renai@example.com"))
299     manager.add_contact(Contact(182, "Xosha Roquemore", "9090909090", "xosha.roquemore@example.com"))
300     manager.add_contact(Contact(183, "Yara Shahidi", "9191919191", "yara.shahidi@example.com"))
301     manager.add_contact(Contact(184, "Zoe Lister-Jones", "9292929292", "zoe.lister-jones@example.com"))
302     manager.add_contact(Contact(185, "Adam Driver", "9393939393", "adam.driver@example.com"))
303     manager.add_contact(Contact(186, "Brie Larson", "9494949494", "brie.larson@example.com"))
304     manager.add_contact(Contact(187, "Cobie Smulders", "9595959595", "cobie.smulders@example.com"))
305     manager.add_contact(Contact(188, "Dakota Fanning", "9696969696", "dakota.fanning@example.com"))
306     manager.add_contact(Contact(189, "Ewan McGregor", "9797979797", "ewan.mcgregor@example.com"))
307     manager.add_contact(Contact(190, "Florence Pugh", "9898989898", "florence.pugh@example.com"))
308     manager.add_contact(Contact(191, "Gael García Bernal", "9999999999", "gael.garcia.bernal@example.com"))
309     manager.add_contact(Contact(192, "Hailee Steinfeld", "1010101010", "hailee.steinfeld@example.com"))
310     manager.add_contact(Contact(193, "Idina Menzel", "1111111111", "idina.menzel@example.com"))
311     manager.add_contact(Contact(194, "Jacob Tremblay", "1212121212", "jacob.tremblay@example.com"))
312     manager.add_contact(Contact(195, "Kaiti Herewini", "1313131313", "kaiti.herewini@example.com"))
313     manager.add_contact(Contact(196, "Liam Neeson", "1414141414", "liam.neeson@example.com"))
314     manager.add_contact(Contact(197, "Milla Jovovich", "1515151515", "milla.jovovich@example.com"))
315     manager.add_contact(Contact(198, "Natalie Portman", "1616161616", "natalie.portman@example.com"))
316     manager.add_contact(Contact(199, "Oscar Isaac", "1717171717", "oscar.isaac@example.com"))
317     manager.add_contact(Contact(200, "Penelope Cruz", "1818181818", "penelope.cruz@example.com"))
318     manager.add_contact(Contact(201, "Quentin Tarantino", "1919191919", "quentin.tarantino@example.com"))
319     manager.add_contact(Contact(202, "Ryan Murphy", "2020202020", "ryan.murphy@example.com"))
320     manager.add_contact(Contact(203, "Saoirse Ronan", "2121212121", "saoirse.ronan@example.com"))
321     manager.add_contact(Contact(204, "Tina Turner", "2222222222", "tina.turner@example.com"))
322     manager.add_contact(Contact(205, "Uma Thurman", "2323232323", "uma.thurman@example.com"))
323     manager.add_contact(Contact(206, "Viggo Mortensen", "2424242424", "viggo.mortensen@example.com"))
324     manager.add_contact(Contact(207, "Wendie Renai", "2525252525", "wendie.renai@example.com"))
325     manager.add_contact(Contact(208, "Xosha Roquemore", "2626262626", "xosha.roquemore@example.com"))
326     manager.add_contact(Contact(209, "Yara Shahidi", "2727272727", "yara.shahidi@example.com"))
327     manager.add_contact(Contact(210, "Zoe Lister-Jones", "2828282828", "zoe.lister-jones@example.com"))
328     manager.add_contact(Contact(211, "Adam Driver", "2929292929", "adam.driver@example.com"))
329     manager.add_contact(Contact(212, "Brie Larson", "3030303030", "brie.larson@example.com"))
330     manager.add_contact(Contact(213, "Cobie Smulders", "3131313131", "cobie.smulders@example.com"))
331     manager.add_contact(Contact(214, "Dakota Fanning", "3232323232", "dakota.fanning@example.com"))
332     manager.add_contact(Contact(215, "Ewan McGregor", "3333333333", "ewan.mcgregor@example.com"))
333     manager.add_contact(Contact(216, "Florence Pugh", "3434343434", "florence.pugh@example.com"))
334     manager.add_contact(Contact(217, "Gael García Bernal", "3535353535", "gael.garcia.bernal@example.com"))
335     manager.add_contact(Contact(218, "Hailee Steinfeld", "3636363636", "hailee.steinfeld@example.com"))
336     manager.add_contact(Contact(219, "Idina Menzel", "3737373737", "idina.menzel@example.com"))
337     manager.add_contact(Contact(220, "Jacob Tremblay", "3838383838", "jacob.tremblay@example.com"))
338     manager.add_contact(Contact(221, "Kaiti Herewini", "3939393939", "kaiti.herewini@example.com"))
339     manager.add_contact(Contact(222, "Liam Neeson", "4040404040", "liam.neeson@example.com"))
340     manager.add_contact(Contact(223, "Milla Jovovich", "4141414141", "milla.jovovich@example.com"))
341     manager.add_contact(Contact(224, "Natalie Portman", "4242424242", "natalie.portman@example.com"))
342     manager.add_contact(Contact(225, "Oscar Isaac", "4343434343", "oscar.isaac@example.com"))
343     manager.add_contact(Contact(226, "Penelope Cruz", "4444444444", "penelope.cruz@example.com"))
344     manager.add_contact(Contact(227, "Quentin Tarantino", "4545454545", "quentin.tarantino@example.com"))
345     manager.add_contact(Contact(228, "Ryan Murphy", "4646464646", "ryan.murphy@example.com"))
346     manager.add_contact(Contact(229, "Saoirse Ronan", "4747474747", "saoirse.ronan@example.com"))
347     manager.add_contact(Contact(230, "Tina Turner", "4848484848", "tina.turner@example.com"))
348     manager.add_contact(Contact(231, "Uma Thurman", "4949494949", "uma.thurman@example.com"))
349     manager.add_contact(Contact(232, "Viggo Mortensen", "5050505050", "viggo.mortensen@example.com"))
350     manager.add_contact(Contact(233, "Wendie Renai", "5151515151", "wendie.renai@example.com"))
351     manager.add_contact(Contact(234, "Xosha Roquemore", "5252525252", "xosha.roquemore@example.com"))
352     manager.add_contact(Contact(235, "Yara Shahidi", "5353535353", "yara.shahidi@example.com"))
353     manager.add_contact(Contact(236, "Zoe Lister-Jones", "5454545454", "zoe.lister-jones@example.com"))
354     manager.add_contact(Contact(237, "Adam Driver", "5555555555", "adam.driver@example.com"))
355     manager.add_contact(Contact(238, "Brie Larson", "5656565656", "brie.larson@example.com"))
356     manager.add_contact(Contact(239, "Cobie Smulders", "5757575757", "cobie.smulders@example.com"))
357     manager.add_contact(Contact(240, "Dakota Fanning", "5858585858", "dakota.fanning@example.com"))
358     manager.add_contact(Contact(241, "Ewan McGregor", "5959595959", "ewan.mcgregor@example.com"))
359     manager.add_contact(Contact(242, "Florence Pugh", "6060606060", "florence.pugh@example.com"))
360     manager.add_contact(Contact(243, "Gael García Bernal", "6161616161", "gael.garcia.bernal@example.com"))
361     manager.add_contact(Contact(244, "Hailee Steinfeld", "6262626262", "hailee.steinfeld@example.com"))
362     manager.add_contact(Contact(245, "Idina Menzel", "6363636363", "idina.menzel@example.com"))
363     manager.add_contact(Contact(246, "Jacob Tremblay", "6464646464", "jacob.tremblay@example.com"))
364     manager.add_contact(Contact(247, "Kaiti Herewini", "6565656565", "kaiti.herewini@example.com"))
365     manager.add_contact(Contact(248, "Liam Neeson", "6666666666", "liam.neeson@example.com"))
366     manager.add_contact(Contact(249, "Milla Jovovich", "6767676767", "milla.jovovich@example.com"))
367     manager.add_contact(Contact(250, "Natalie Portman", "6868686868", "natalie.portman@example.com"))
368     manager.add_contact(Contact(251, "Oscar Isaac", "6969696969", "oscar.isaac@example.com"))
369     manager.add_contact(Contact(252, "Penelope Cruz", "7070707070", "penelope.cruz@example.com"))
370     manager.add_contact(Contact(253, "Quentin Tarantino", "7171717171", "quentin.tarantino@example.com"))
371     manager.add_contact(Contact(254, "Ryan Murphy", "7272727272", "ryan.murphy@example.com"))
372     manager.add_contact(Contact(255, "Saoirse Ronan", "7373737373", "saoirse.ronan@example.com"))
373     manager.add_contact(Contact(256, "Tina Turner", "7474747474", "tina.turner@example.com"))
374     manager.add_contact(Contact(257, "Uma Thurman", "7575757575", "uma.thurman@example.com"))
375     manager.add_contact(Contact(258, "Viggo Mortensen", "7676767676", "viggo.mortensen@example.com"))
376     manager.add_contact(Contact(259, "Wendie Renai", "7777777777", "wendie.renai@example.com"))
377     manager.add_contact(Contact(260, "Xosha Roquemore", "7878787878", "xosha.roquemore@example.com"))
378     manager.add_contact(Contact(261, "Yara Shahidi", "7979797979", "yara.shahidi@example.com"))
379     manager.add_contact(Contact(262, "Zoe Lister-Jones", "8080808080", "zoe.lister-jones@example.com"))
380     manager.add_contact(Contact(263, "Adam Driver", "8181818181", "adam.driver@example.com"))
381     manager.add_contact(Contact(264, "Brie Larson", "8282828282", "brie.larson@example.com"))
382     manager.add_contact(Contact(265, "Cobie Smulders", "8383838383", "cobie.smulders@example.com"))
383     manager.add_contact(Contact(266, "Dakota Fanning", "8484848484", "dakota.fanning@example.com"))
384     manager.add_contact(Contact(267, "Ewan McGregor", "8585858585", "ewan.mcgregor@example.com"))
385     manager.add_contact(Contact(268, "Florence Pugh", "8686868686", "florence.pugh@example.com"))
386     manager.add_contact(Contact(269, "Gael García Bernal", "8787878787", "gael.garcia.bernal@example.com"))
387     manager.add_contact(Contact(270, "Hailee Steinfeld", "88888
```

```
1 # Contact Manager
2
3 # Importing modules
4 import sys
5
6 # Global variables
7 contacts = []
8
9 # Function to add a new contact
10 def add_contact():
11     print("\nAdd a new contact")
12     name = input("Name: ")
13     phone = input("Phone: ")
14     email = input("Email: ")
15     contacts.append({'name': name, 'phone': phone, 'email': email})
16     print("Contact added successfully")
17
18 # Function to delete a contact
19 def delete_contact():
20     print("\nDelete a contact")
21     name = input("Name: ")
22     for i, contact in enumerate(contacts):
23         if contact['name'] == name:
24             del contacts[i]
25             print("Contact deleted successfully")
26             return
27     print("Contact not found")
28
29 # Function to search for a contact
30 def search_contact():
31     print("\nSearch for a contact")
32     name = input("Name: ")
33     results = []
34     for contact in contacts:
35         if name in contact['name']:
36             results.append(contact)
37     print("Search results for '%s': %s" % (name, results))
38
39 # Function to display all contacts
40 def display_contacts():
41     print("\nDisplay all contacts")
42     if len(contacts) == 0:
43         print("No contacts found")
44     else:
45         for contact in contacts:
46             print(contact)
47
48 # Main function
49 def main():
50     while True:
51         print("\nContact Manager")
52         print("1. Add contact")
53         print("2. Delete contact")
54         print("3. Search contact")
55         print("4. Display all contacts")
56         print("5. Exit")
57         choice = input("Enter your choice: ")
58         if choice == '1':
59             add_contact()
60         elif choice == '2':
61             delete_contact()
62         elif choice == '3':
63             search_contact()
64         elif choice == '4':
65             display_contacts()
66         elif choice == '5':
67             break
68
69 if __name__ == '__main__':
70     main()
```

```
1 # Contact Manager
2
3 # Importing modules
4 import sys
5
6 # Global variables
7 contacts = []
8
9 # Function to add a new contact
10 def add_contact():
11     print("\nAdd a new contact")
12     name = input("Name: ")
13     phone = input("Phone: ")
14     email = input("Email: ")
15     contacts.append({'name': name, 'phone': phone, 'email': email})
16     print("Contact added successfully")
17
18 # Function to delete a contact
19 def delete_contact():
20     print("\nDelete a contact")
21     name = input("Name: ")
22     for i, contact in enumerate(contacts):
23         if contact['name'] == name:
24             del contacts[i]
25             print("Contact deleted successfully")
26             return
27     print("Contact not found")
28
29 # Function to search for a contact
30 def search_contact():
31     print("\nSearch for a contact")
32     name = input("Name: ")
33     results = []
34     for contact in contacts:
35         if name in contact['name']:
36             results.append(contact)
37     print("Search results for '%s': %s" % (name, results))
38
39 # Function to display all contacts
40 def display_contacts():
41     print("\nDisplay all contacts")
42     if len(contacts) == 0:
43         print("No contacts found")
44     else:
45         for contact in contacts:
46             print(contact)
47
48 # Main function
49 def main():
50     while True:
51         print("\nContact Manager")
52         print("1. Add contact")
53         print("2. Delete contact")
54         print("3. Search contact")
55         print("4. Display all contacts")
56         print("5. Exit")
57         choice = input("Enter your choice: ")
58         if choice == '1':
59             add_contact()
60         elif choice == '2':
61             delete_contact()
62         elif choice == '3':
63             search_contact()
64         elif choice == '4':
65             display_contacts()
66         elif choice == '5':
67             break
68
69 if __name__ == '__main__':
70     main()
```

Output:



Explanation:

- In an array, adding at the end is fast, but inserting in the middle is slow because elements must shift.
- In a linked list, insertion is fast because no shifting is needed.
- Searching takes the same time in both (you must check each element).
- Deleting in an array is slower due to shifting elements.
- Linked list is better for frequent insertions and deletions.

Task 2:

Library Book Search System (Queues & Priority Queues)

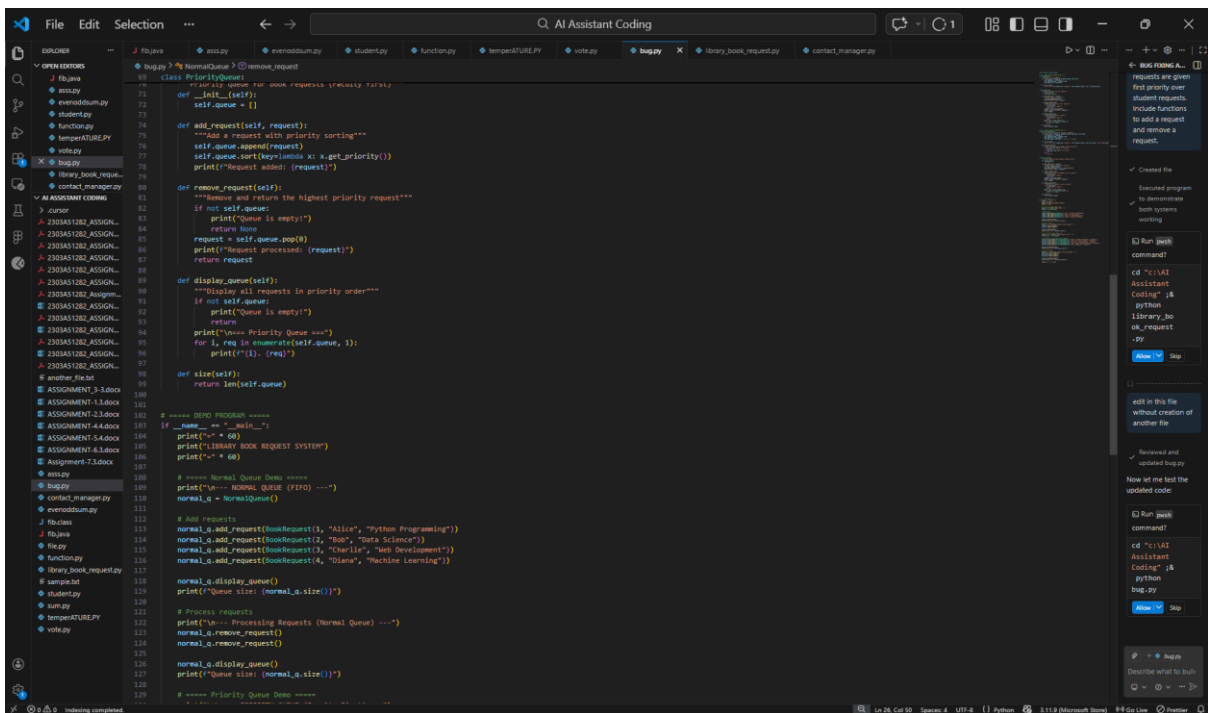
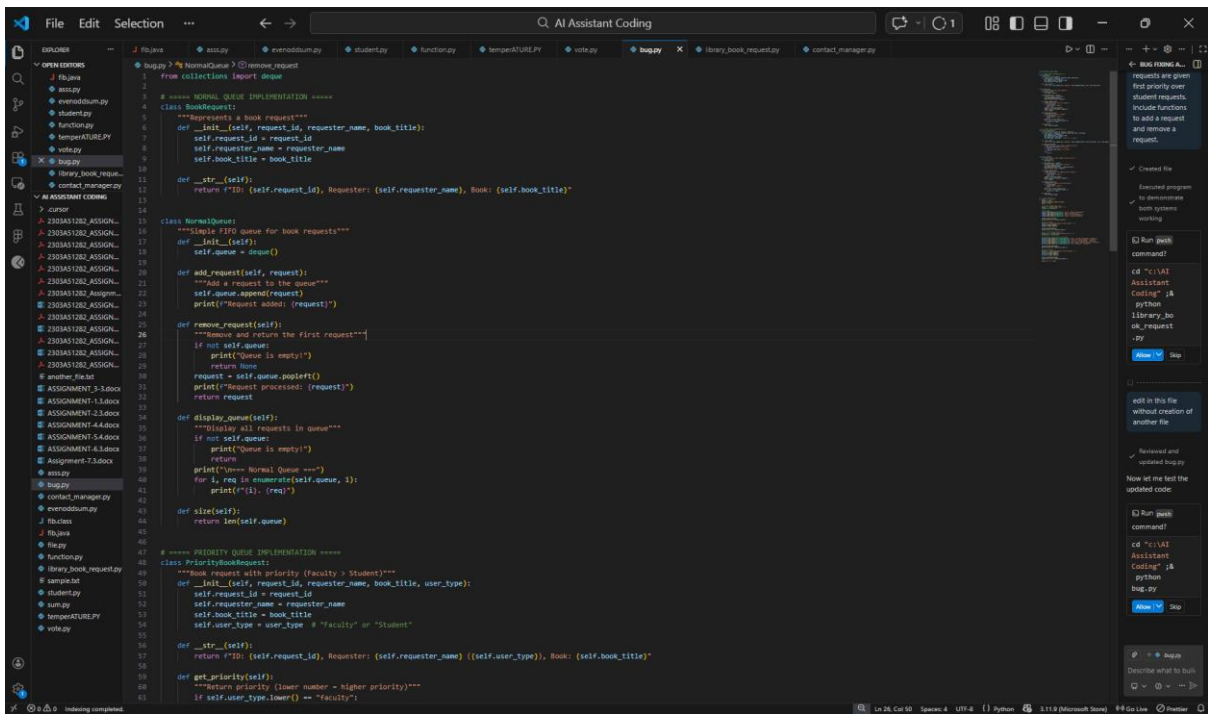
Scenario

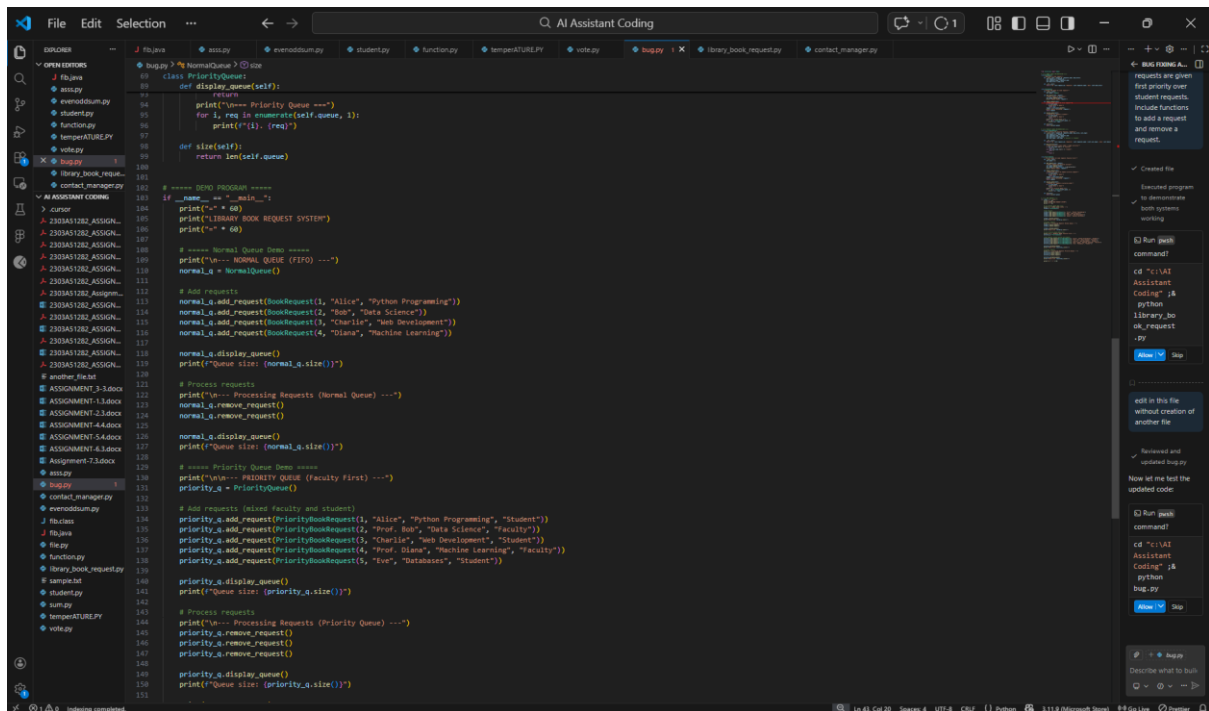
The SRU Library manages book borrow requests. Students and faculty submit requests, but faculty requests must be prioritized over student requests.

Prompt:

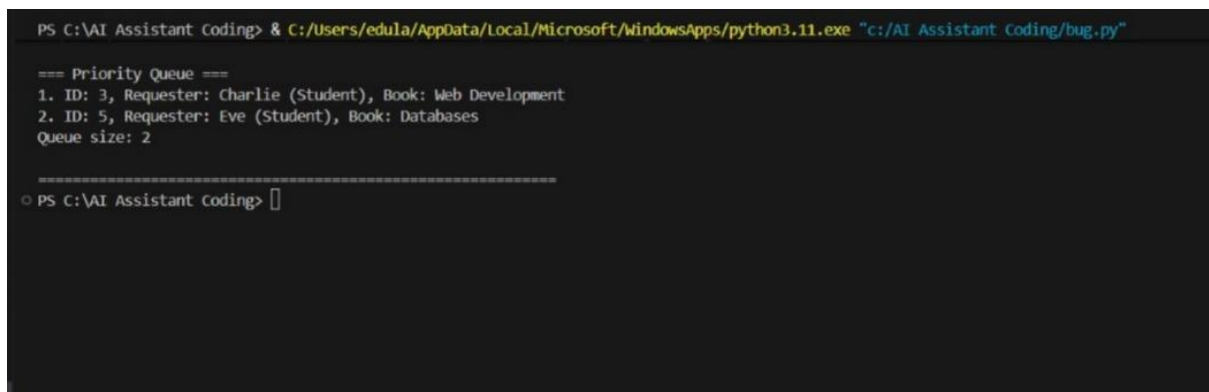
Write a Python program for a library book request system. First, make a normal queue where requests are handled in the order they come. Then, make another version where faculty requests are given first priority over student requests. Include functions to add a request and remove a request.

Code:





Output:



Explanation:

- Queue (FIFO) → First request comes, first served.(If a student requests first, they get the book first.)
- Priority Queue → Faculty requests are served before students, even if they come later.
- enqueue() → Adds a request to the system.
- dequeue() → Removes and processes the next request.

Task 3: Emergency Help Desk (Stack Implementation)

Scenario

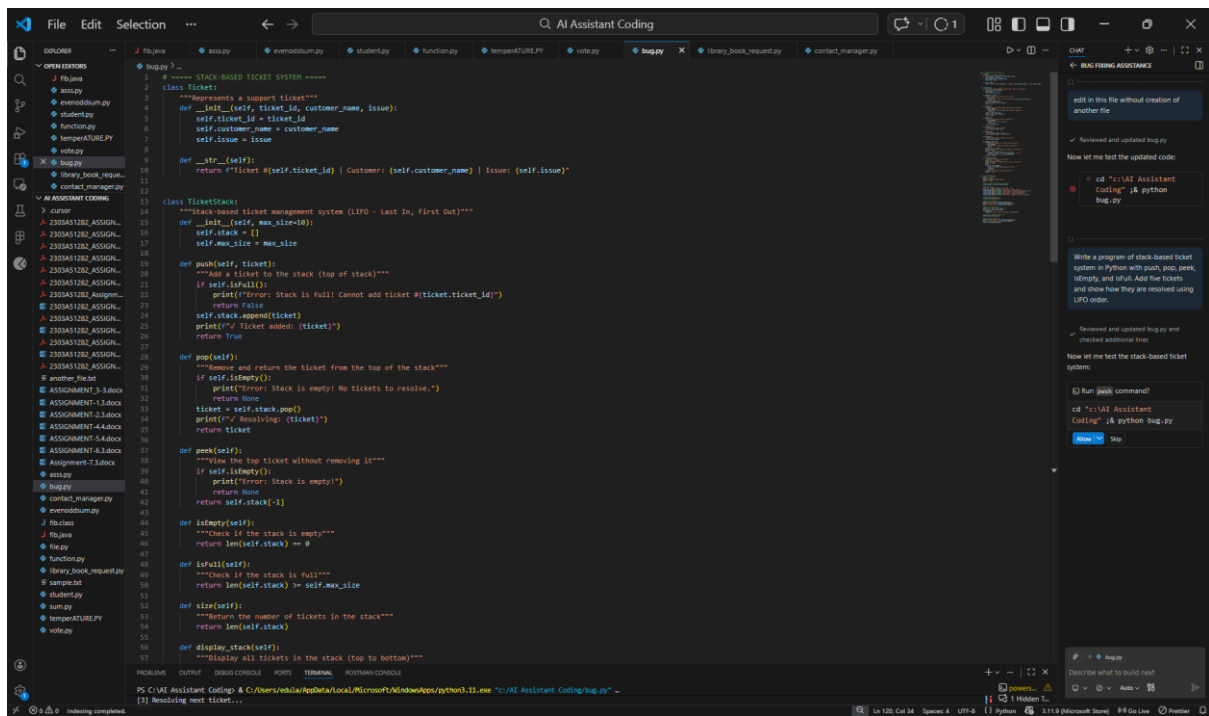
SR University's IT Help Desk receives technical support tickets from students and staff.

While tickets are received sequentially, issue escalation follows a Last-In, First-Out (LIFO) approach.

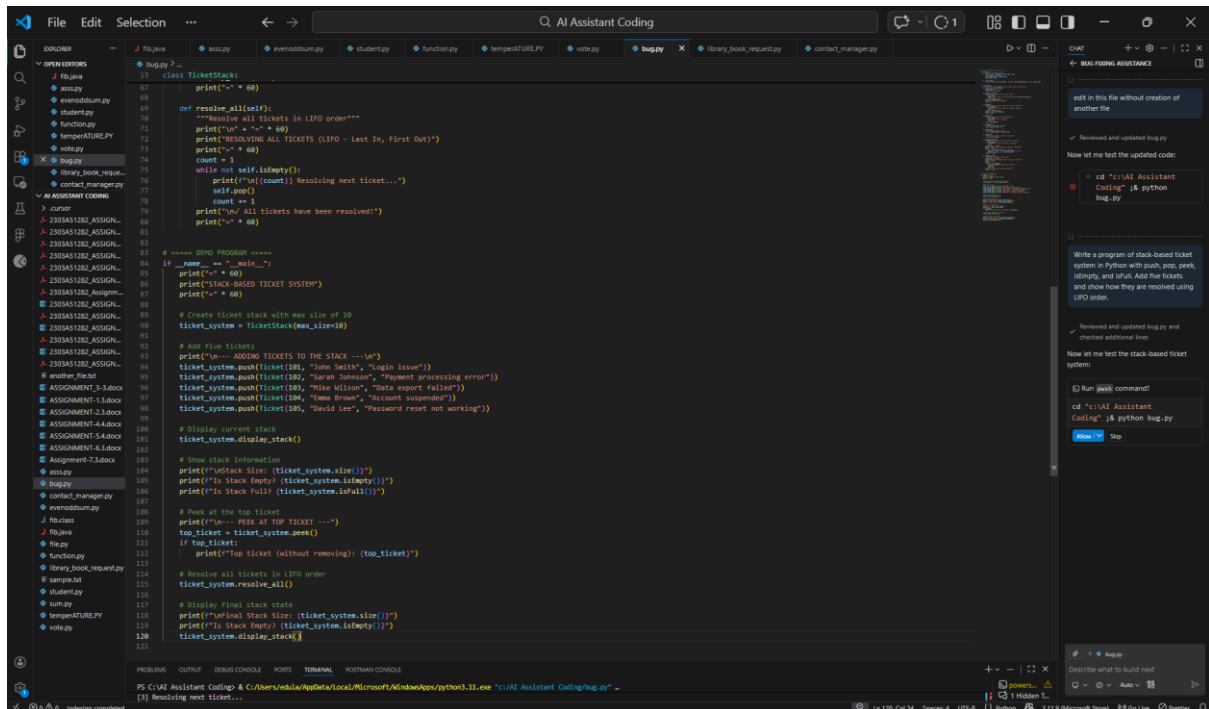
Prompt:

Write a program of stack-based ticket system in Python with push, pop, peek, isEmpty, and isFull. Add five tickets and show how they are resolved using LIFO order.

Code:



```
1 # ===== STACK-BASED TICKET SYSTEM =====
2 class Ticket:
3     """Represents a support ticket"""
4     def __init__(self, ticket_id, customer_name, issue):
5         self.ticket_id = ticket_id
6         self.customer_name = customer_name
7         self.issue = issue
8
9     def __str__(self):
10         return f"Ticket #{self.ticket_id} | Customer: {self.customer_name} | Issue: {self.issue}"
11
12 class TicketStack:
13     """Stack-based ticket management system (LIFO - Last In, First Out)"""
14     def __init__(self, max_size=10):
15         self.stack = []
16         self.max_size = max_size
17
18     def push(self, ticket):
19         """Add a ticket to the stack (top of stack)"""
20         if self.is_full():
21             print("Error: Stack is full! Cannot add ticket #", ticket.ticket_id)
22             return False
23         self.stack.append(ticket)
24         print(f"/ Pushed ticket #{ticket.ticket_id}")
25         return True
26
27     def pop(self):
28         """Remove and return the ticket from the top of the stack"""
29         if self.is_empty():
30             print("Error: Stack is empty! No tickets to resolve.")
31             return None
32         ticket = self.stack.pop()
33         print(f"/ Resolving ticket #{ticket.ticket_id}")
34         return ticket
35
36     def peek(self):
37         """View the top ticket without removing it"""
38         if self.is_empty():
39             print("Error: Stack is empty!")
40             return None
41         return self.stack[-1]
42
43     def is_empty(self):
44         """Check if the stack is empty"""
45         return len(self.stack) == 0
46
47     def is_full(self):
48         """Check if the stack is full"""
49         return len(self.stack) == self.max_size
50
51     def size(self):
52         """Return the number of tickets in the stack"""
53         return len(self.stack)
54
55     def display_stack(self):
56         """Display all tickets in the stack (top to bottom)"""
57         for ticket in reversed(self.stack):
58             print(ticket)
```



```
1 # ===== STACK-BASED TICKET SYSTEM =====
2 class Ticket:
3     """Represents a support ticket"""
4     def __init__(self, ticket_id, customer_name, issue):
5         self.ticket_id = ticket_id
6         self.customer_name = customer_name
7         self.issue = issue
8
9     def __str__(self):
10         return f"Ticket #{self.ticket_id} | Customer: {self.customer_name} | Issue: {self.issue}"
11
12 class TicketStack:
13     """Stack-based ticket management system (LIFO - Last In, First Out)"""
14     def __init__(self, max_size=10):
15         self.stack = []
16         self.max_size = max_size
17
18     def push(self, ticket):
19         """Add a ticket to the stack (top of stack)"""
20         if self.is_full():
21             print("Error: Stack is full! Cannot add ticket #", ticket.ticket_id)
22             return False
23         self.stack.append(ticket)
24         print(f"/ Pushed ticket #{ticket.ticket_id}")
25         return True
26
27     def pop(self):
28         """Remove and return the ticket from the top of the stack"""
29         if self.is_empty():
30             print("Error: Stack is empty! No tickets to resolve.")
31             return None
32         ticket = self.stack.pop()
33         print(f"/ Resolving ticket #{ticket.ticket_id}")
34         return ticket
35
36     def peek(self):
37         """View the top ticket without removing it"""
38         if self.is_empty():
39             print("Error: Stack is empty!")
40             return None
41         return self.stack[-1]
42
43     def is_empty(self):
44         """Check if the stack is empty"""
45         return len(self.stack) == 0
46
47     def is_full(self):
48         """Check if the stack is full"""
49         return len(self.stack) == self.max_size
50
51     def size(self):
52         """Return the number of tickets in the stack"""
53         return len(self.stack)
54
55     def display_stack(self):
56         """Display all tickets in the stack (top to bottom)"""
57         for ticket in reversed(self.stack):
58             print(ticket)
59
60 # ===== MAIN PROGRAM =====
61 if __name__ == "__main__":
62     print("===== STACK-BASED TICKET SYSTEM =====")
63
64     # Create ticket stack with max size of 10
65     ticket_system = TicketStack(max_size=10)
66
67     # Add five tickets
68     print("\n--- ADDING TICKETS TO THE STACK ---\n")
69     ticket_system.push(ticket(100, "John Smith", "Login issue"))
70     ticket_system.push(ticket(101, "Sarah Johnson", "Payment processing error"))
71     ticket_system.push(ticket(102, "Mike Wilson", "Data export failed"))
72     ticket_system.push(ticket(103, "Emma Brown", "Account suspended"))
73     ticket_system.push(ticket(104, "David Lee", "Password reset not working"))
74
75     # Display current stack
76     ticket_system.display_stack()
77
78     # Show stack information
79     print("\nStack Size: ", ticket_system.size())
80     print("Is Stack Empty? ", ticket_system.is_empty())
81     print("Is Stack Full? ", ticket_system.is_full())
82
83     # Peek at the top ticket
84     print("\n--- PEAK AT TOP TICKET ---\n")
85     top_ticket = ticket_system.peek()
86     if top_ticket:
87         print(f"Top ticket (without removing): {top_ticket}")
88
89     # Resolve all tickets in LIFO order
90     ticket_system.resolve_all()
91
92     # Display final stack state
93     print("\nFinal Stack Size: ", ticket_system.size())
94     print("Is Stack Empty? ", ticket_system.is_empty())
95     ticket_system.display_stack()
```

Output:

```
File Edit Selection ... AI Assistant Coding
EXPLORER
  OPEN EDITORS
    files.py
    helper.py
    evenoddsum.py
    student.py
    function.py
    temperature.py
    volk.py
    bug.py
    helper_book_request.py
    contact_manager.py
    evenoddsum.py
    files.py
    function.py
    helper_book_request.py
    sample.txt
    student.py
    sum.py
    temperature.py
    volk.py
  PROBLEMS
  OUTPUT
  DEBUG CONSOLE
  PORTS
  TERMINAL
  POSTMAN CONSOLE
  PS C:\AI Assistant Coding> C:\Users\eddie\AppData\Local\Microsoft\WindowsApps\python3.11.exe "C:\AI Assistant Coding\bug.py"
  [1] Resolving next ticket...
  ✓ Resolving Ticket #181 | Customer: John Smith | Issue: Login issue
  ✓ All tickets have been resolved!
  Final Stack Size: 0
  Is Stack Empty? True
  Stack is empty! No tickets to display.
  PS C:\AI Assistant Coding>
  [2] Resolving next ticket...
  ✓ Resolving Ticket #184 | Customer: Emma Brown | Issue: Account suspended
  [3] Resolving next ticket...
  ✓ Resolving Ticket #183 | Customer: Mike Wilson | Issue: Data export failed
  [4] Resolving next ticket...
  ✓ Resolving Ticket #182 | Customer: Sarah Johnson | Issue: Payment processing error
  [5] Resolving next ticket...
  ✓ Resolving Ticket #181 | Customer: John Smith | Issue: Login issue
  ✓ All tickets have been resolved!
  [2] Resolving next ticket...
  ✓ Resolving Ticket #184 | Customer: Emma Brown | Issue: Account suspended
  [3] Resolving next ticket...
  ✓ Resolving Ticket #183 | Customer: Mike Wilson | Issue: Data export failed
  [4] Resolving next ticket...
  ✓ Resolving Ticket #182 | Customer: Sarah Johnson | Issue: Payment processing error
  [5] Resolving next ticket...
  ✓ Resolving Ticket #181 | Customer: John Smith | Issue: Login issue
  ✓ All tickets have been resolved!
  [2] Resolving next ticket...
  ✓ Resolving Ticket #184 | Customer: Emma Brown | Issue: Account suspended
  [3] Resolving next ticket...
  ✓ Resolving Ticket #183 | Customer: Mike Wilson | Issue: Data export failed
  [4] Resolving next ticket...
  ✓ Resolving Ticket #182 | Customer: Sarah Johnson | Issue: Payment processing error
  [5] Resolving next ticket...
  ✓ Resolving Ticket #181 | Customer: John Smith | Issue: Login issue
  ✓ All tickets have been resolved!
  Final Stack Size: 0
  Is Stack Empty? True
  Stack is empty! No tickets to display.
  PS C:\AI Assistant Coding>
```

Explanation:

The program uses a stack to manage help desk tickets.

A stack works in last in, first solved order.

When a new ticket is raised, it is added to the top.

When solving a ticket, the most recent one is handled first.

The program can also check if there are no tickets left or if the stack is full.

Task 4:

Hash Table

Objective

To implement a Hash Table and understand collision handling.

Prompt:

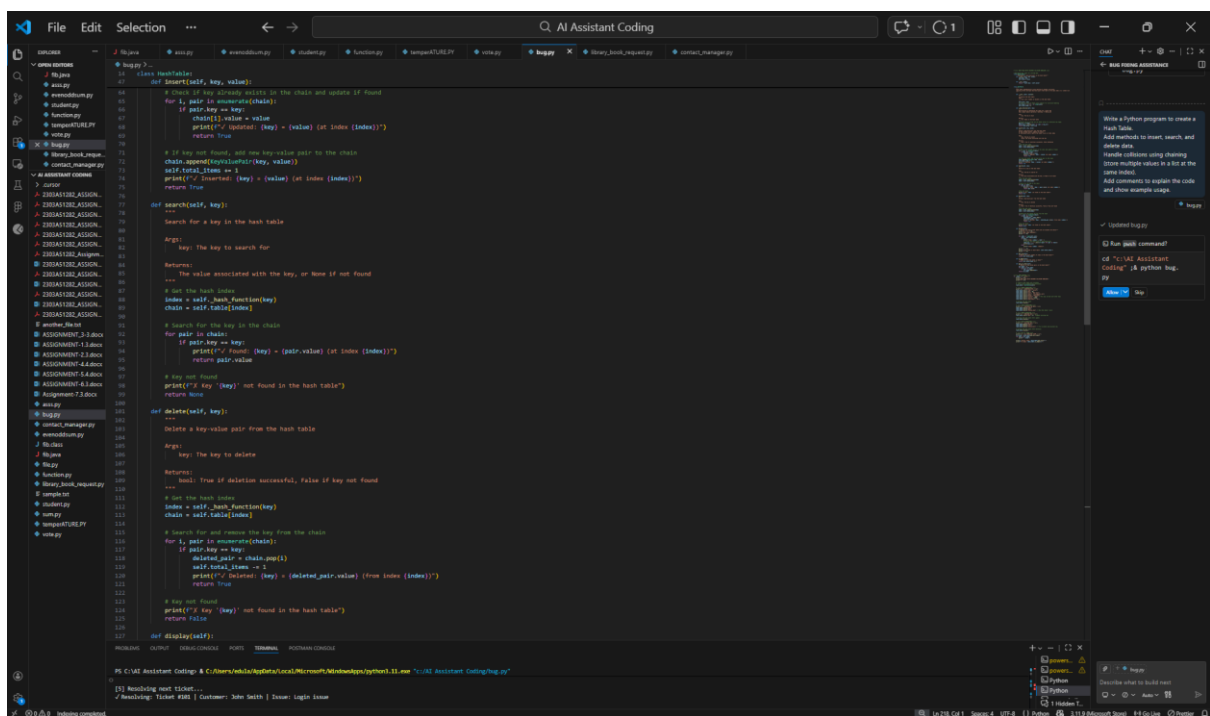
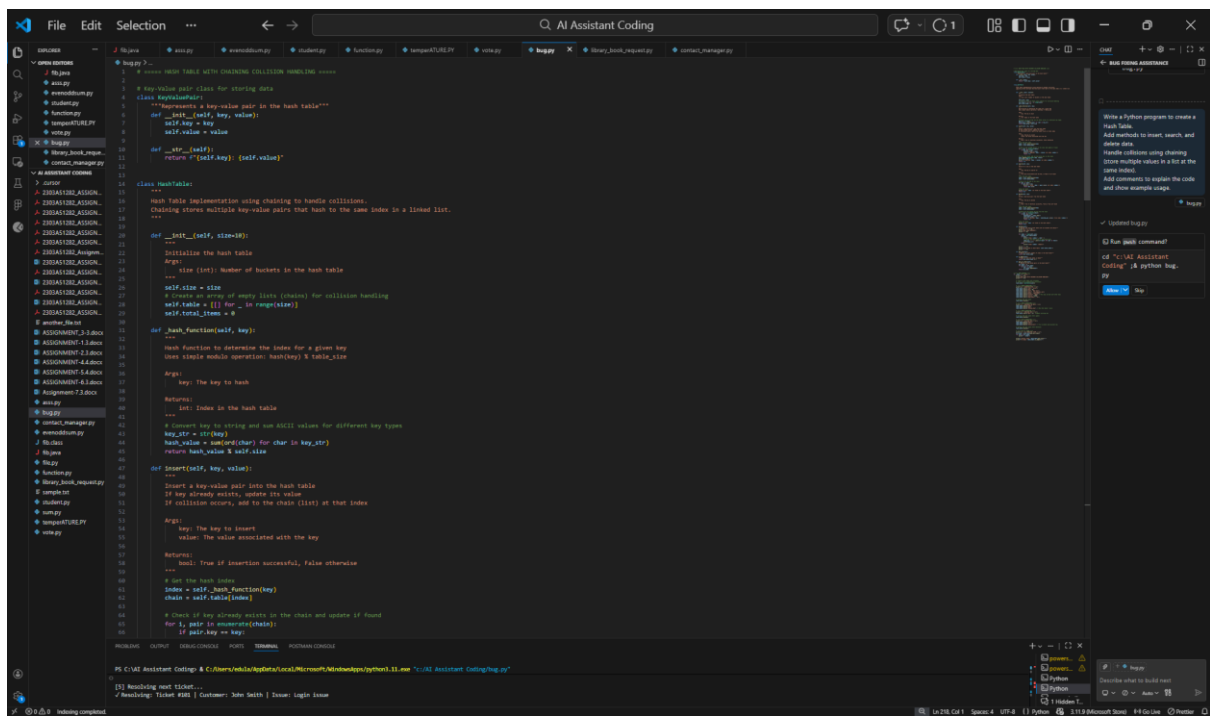
Write a Python program to create a Hash Table.

Add methods to insert, search, and delete data.

Handle collisions using chaining (store multiple values in a list at the same index).

Add comments to explain the code and show example usage.

Code:




```
def get_all_items(self):  
    """Returns all key-value pairs in the hash table"""  
    all_items = []  
    for chain in self.table:  
        for pair in chain:  
            all_items.append(pair)  
    return all_items  
  
# ===== DONE PROGRAM =====  
if __name__ == "__main__":  
    print("\n***\n")  
    print("HASH TABLE WITH CHAINING COLLISION HANDLING")  
    print("\n***\n")  
    # Create a hash table with 5 buckets  
    # (we'll also use a separate collision handler)  
    hash_table = HashTable(10)  
  
    # ===== INSERT OPERATIONS =====  
    print("\n--- INSERTING DATA ---\n")  
    hash_table.insert("name", "Alice")  
    hash_table.insert("age", 30)  
    hash_table.insert("city", "New York")  
    hash_table.insert("email", "alice@gmail.com")  
    hash_table.insert("phone", "555-1234")  
    hash_table.insert("country", "USA") # This key will collide with other keys  
    hash_table.insert("salary", 75000)  
    hash_table.insert("department", "IT")  
  
    # Display the hash table  
    hash_table.display()  
  
    # ===== SEARCH OPERATIONS =====  
    print("\n--- SEARCHING FOR DATA ---\n")  
    hash_table.search("name")  
    hash_table.search("age")  
    hash_table.search("unknown_key") # Key that doesn't exist  
  
    # ===== UPDATE OPERATIONS =====  
    print("\n--- UPDATING DATA ---\n")  
    hash_table.update("age", 35) # Update existing key  
  
    # Display the hash table after update  
    hash_table.display()  
  
    # ===== DELETE OPERATIONS =====  
    print("\n--- DELETING DATA ---\n")  
    hash_table.delete("email")  
    hash_table.delete("city")  
    hash_table.delete("nonexistent") # Try to delete non-existent key  
  
    # Display the hash table after deletions  
    hash_table.display()  
  
    # ===== GET ALL ITEMS =====  
    print("\n--- ALL EXISTING ITEMS ---\n")  
    all_items = hash_table.get_all_items()  
    for item in all_items:  
        print(f"Item: {item}")  
  
    print(f"Total items: {len(hash_table.get_all_items())}")  
    print(f"Is empty: {hash_table.is_empty()}")
```

Output:

```
PS C:\AI Assistant Coding> cd "C:\Users\adrian\AppData\Local\Microsoft\WindowsApps\python3.11.exe" "C:\AI Assistant Coding\bug.py"  
Index 0: [empty]  
Index 1: [empty]  
Index 2: [empty]  
Index 3: [empty]  
Index 4: [empty]  
Index 5: [empty]  
Index 6: [empty]  
Index 7: [empty]  
Index 8: [empty]  
Index 9: [empty]  
Total items in hash table: 0  
  
--- INSERTING DATA ---  
Found: name = Alice (at index 2)  
Found: age = 30 (at index 1)  
Key 'unknown_key' not found in the hash table  
Updated: age = 31 (at index 1)  
Total items in hash table: 2  
  
--- SEARCHING FOR DATA ---  
Found: name = Alice (at index 2)  
Found: age = 31 (at index 1)  
Key 'unknown_key' not found in the hash table  
Total items in hash table: 2  
  
--- UPDATING DATA ---  
Updated: age = 31 (at index 1)  
Total items in hash table: 2  
  
--- DELETING DATA ---  
Deleted: email = alice@gmail.com (from index 0)  
Deleted: city = New York (from index 1)  
Key 'nonexistent' not found in the hash table  
Total items in hash table: 0  
  
--- ALL EXISTING ITEMS ---  
Item: {}  
Item: {}  
Item: {}  
Item: {}  
Item: {}  
Item: {}  
Item: {}  
Item: {}  
Item: {}  
Item: {}  
Total items: 0  
Is empty: True  
PS C:\AI Assistant Coding>
```

Explanation:

- A Hash Table stores data using a key and value.
- A hash function decides where to store the data.
- Sometimes two keys go to the same place. This is called a collision.
- To solve collisions, we use chaining, meaning we store multiple items in a list at the same index.
- The program should allow adding, finding, and removing data correctly.

Task 5:

Real-Time Application Challenge Scenario

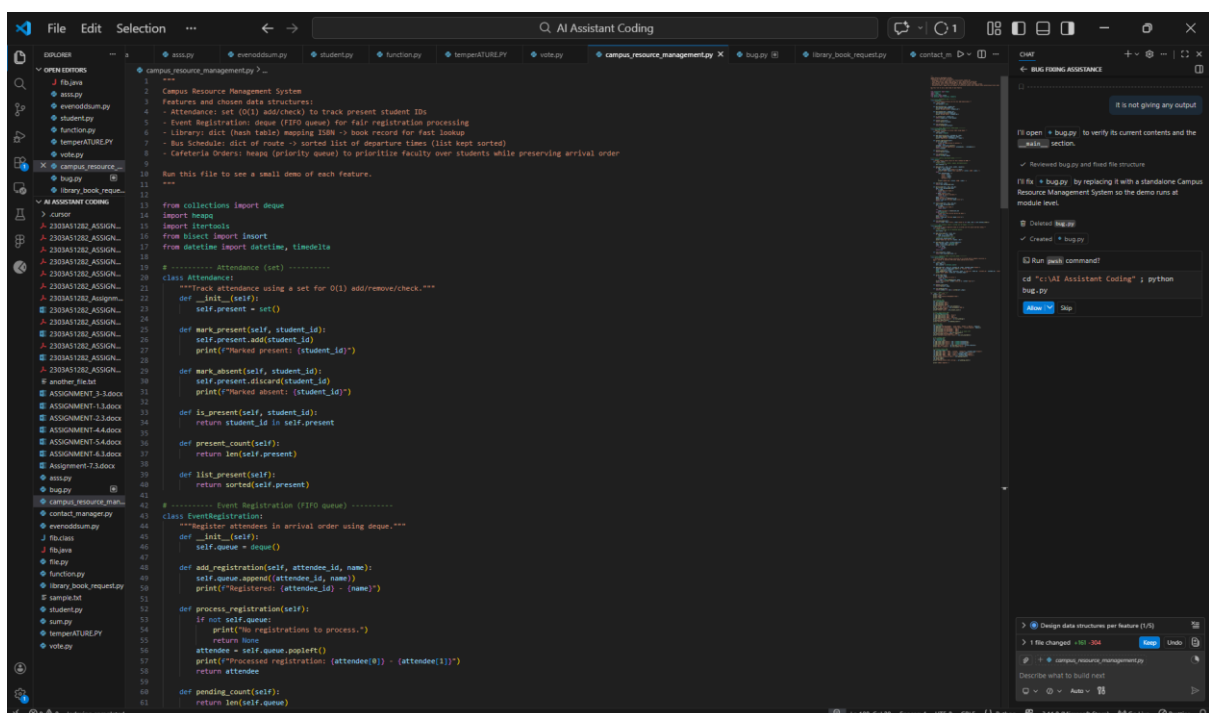
Design a Campus Resource Management System with the following features:

- Student Attendance Tracking
- Event Registration System
- Library Book Borrowing
- Bus Scheduling System
- Cafeteria Order Queue

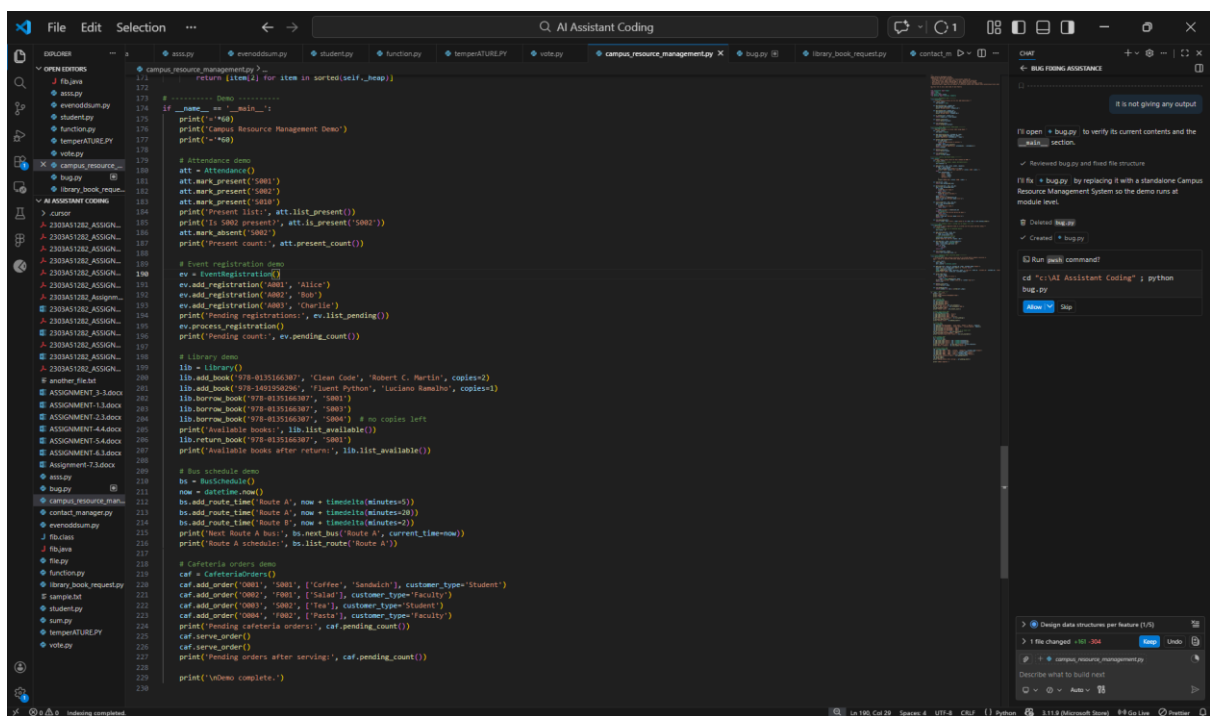
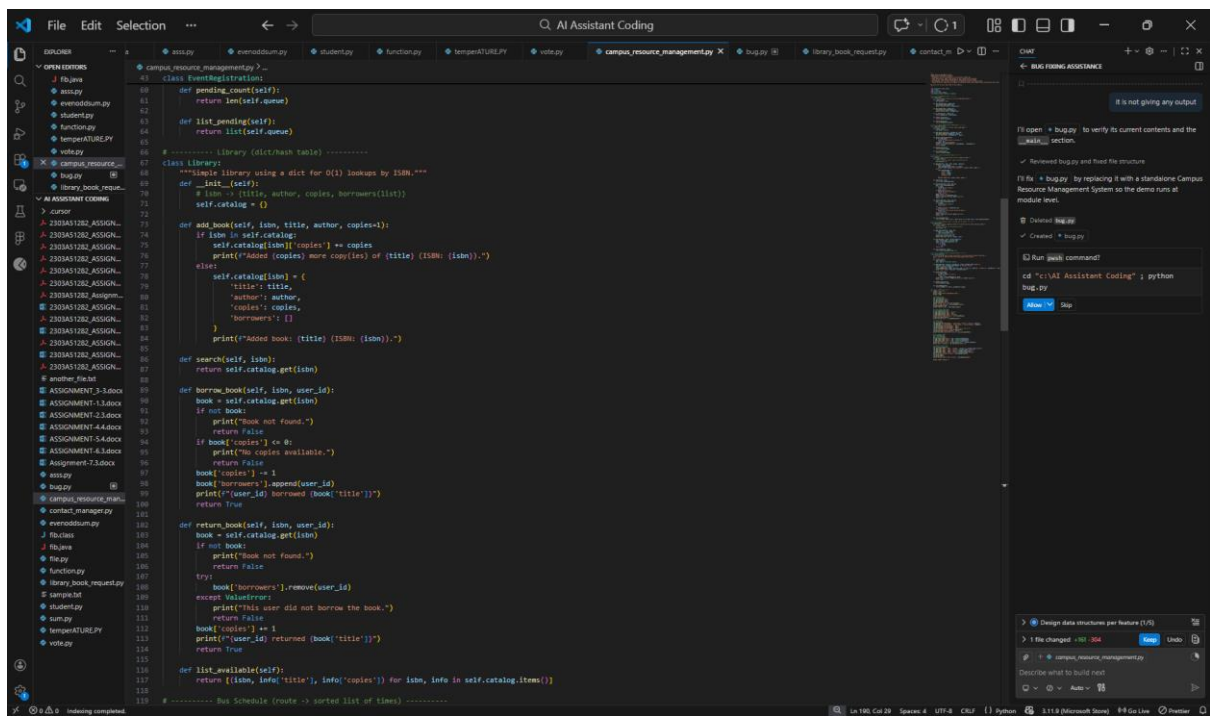
Prompt:

Create a Campus Resource Management System in Python. For each feature (Attendance, Event Registration, Library, Bus Schedule, Cafeteria Orders), choose the best data structure

Code:



```
1 # File Name: campus_resource_management.py
2 """
3 Campus Resource Management System
4 Features and chosen data structures:
5 - Attendance: set (O(1) add/check) to track present student IDs
6 - Event Registration: queue (FIFO queue) for fair registration processing
7 - Library: dict (hash table) mapping ISBN -> book record for fast lookup
8 - Bus Schedule: dict of route -> sorted list of departure times (list kept sorted)
9 - Cafeteria Orders: heap (priority queue) to prioritize faculty over students while preserving arrival order
10 """
11
12 # Run this file to see a small demo of each feature.
13 """
14 """
15
16 from collections import deque
17 import heapq
18 import heapq
19 from bisect import bisect
20 from datetime import datetime, timedelta
21
22 # ----- Attendance (set) -----
23 class Attendance:
24     """Track attendance using a set for O(1) add/remove/check."""
25     def __init__(self):
26         self.present = set()
27
28     def mark_present(self, student_id):
29         self.present.add(student_id)
30         print(f"Marked present: {student_id}")
31
32     def mark_absent(self, student_id):
33         self.present.discard(student_id)
34         print(f"Marked absent: {student_id}")
35
36     def is_present(self, student_id):
37         return student_id in self.present
38
39     def present_count(self):
40         return len(self.present)
41
42     def list_present(self):
43         return sorted(self.present)
44
45 # ----- Event Registration (FIFO queue) -----
46 class EventRegistration:
47     """Register attendees in arrival order using deque."""
48     def __init__(self):
49         self.queue = deque()
50
51     def add_registration(self, attendee_id, name):
52         self.queue.append((attendee_id, name))
53         print(f"Registered: {attendee_id} - {name}")
54
55     def process_registration(self):
56         if not self.queue:
57             print("No registrations to process.")
58             return None
59         attendee = self.queue.popleft()
60         print(f"Processed registration: {attendee[0]} - {attendee[1]}")
61         return attendee
62
63     def pending_count(self):
64         return len(self.queue)
```



Output:

```
PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL POSTMAN CONSOLE

PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/bug.py"
Is empty: False
PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/bug.py"
PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/campus_resource_management.py"

=====
Campus Resource Management Demo
=====

Marked present: 5001
Marked present: 5002
Marked present: 5010
Present list: ['5001', '5002', '5010']
Is 5002 present? True
Marked absent: 5002
Present count: 2
Registered: A001 - Alice
Registered: A002 - Bob
Registered: A003 - Charlie
Pending registrations: [('A001', 'Alice'), ('A002', 'Bob'), ('A003', 'Charlie')]
Processed registration: A001 - Alice
Pending count: 2
Added book: Clean Code (ISBN: 978-0135166307).
Added book: Fluent Python (ISBN: 978-1491958296).
5001 borrowed Clean Code
5003 borrowed Clean Code
No copies available.
Available books: [('978-0135166307', 'Clean Code', 0), ('978-1491958296', 'Fluent Python', 1)]
5001 returned Clean Code
Available books after return: [('978-0135166307', 'Clean Code', 1), ('978-1491958296', 'Fluent Python', 1)]
Added bus time for Route A: 2026-02-18 10:42:24.367227
Added bus time for Route A: 2026-02-18 10:57:24.367227
Added bus time for Route B: 2026-02-18 10:39:24.367227
Next Route A bus: 2026-02-18 10:42:24.367227
Route A schedule: [datetime.datetime(2026, 2, 18, 10, 42, 24, 367227), datetime.datetime(2026, 2, 18, 10, 57, 24, 367227)]
Order added: 0001 (Student)
Order added: 0002 (Faculty)
Order added: 0003 (Student)
Order added: 0004 (Faculty)
Pending cafeteria orders: 4
Serving order: 0002 (Faculty)
Serving order: 0004 (Faculty)
Pending orders after serving: 2

Demo complete.
PS C:\AI Assistant Coding> 
```

Explanation:

Library Book Borrowing using a queue:

- The queue stores student names who request a book.
- When a student requests a book, we use `enqueue()` to add them to the queue.
- When a book becomes available, we use `dequeue()` to give it to the first student in line.
- This ensures fairness because the first requester gets the book first.

