

2303A51739

Batch:25

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.

Tasks

1. Implement the contact manager using arrays (lists).
 2. Implement the same functionality using a linked list for dynamic memory allocation.
 3. Implement the following operations in both approaches:
 - o Add a contact
 - o Search for a contact
 - o Delete a contact
 4. Use GitHub Copilot to assist in generating search and delete methods.
 5. Compare array vs. linked list approaches with respect to:
 - o Insertion efficiency
 - o Deletion efficiency
- Expected Outcome
- Two working implementations (array-based and linked-list-based).
 - A brief comparison explaining performance differences.

```

File Edit Selection View Go Run Terminal Help ← → ⌘ Search
RUN AND DEBUG
RUN
Run and Debug
To customize Run and Debug, open a folder and create a launch.json file.
Debug using a terminal command or in an interactive shell.
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\Chinmari\Downloads> cd "C:\Users\Chinmari\Downloads"; & "C:\Users\Chinmari\AppData\Local\Microsoft\WindowsApps\python3.11.exe" "C:\Users\Chinmari\.vscode\extensions\ms-python.python.debug-2025.10.0-win32-x64\bundle\lib\debug\launch"
Array Contacts:
Alice - 1111111111
Linked List Contacts:
Alice - 1111111111

Comparison:
Insertion Efficiency:
Array: O(1) average but resizing may occur.
Linked List: O(n) for search but deletion is pointer update.
PS C:\Users\Chinmari\Downloads>

```

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.

Tasks

1. Implement a Queue (FIFO) to manage book requests.
2. Extend the system to a Priority Queue, prioritizing faculty requests.
3. Use GitHub Copilot to assist in generating:
 - o enqueue() method
 - o dequeue() method
4. Test the system with a mix of student and faculty requests.

Expected Outcome

- Working queue and priority queue implementations.
- Correct prioritization of faculty requests

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows a folder structure including 'new.py'.
- Code Editor:** Displays the content of 'new.py'. The code defines two classes: 'QueueSystem' and 'PriorityQueueSystem'. The 'QueueSystem' class uses a simple list for requests. The 'PriorityQueueSystem' class uses a list of tuples where each tuple contains priority (0 or 1), name, role, and book. It includes methods for enqueueing, dequeuing, displaying requests, and printing queue details.
- Terminal:** Shows the command to run the file: 'code . & python new.py'.
- Output:** Shows the execution of the script, outputting 'Queue Requests' and 'Priority Queue Requests' with sample data.
- Search:** A search bar at the top.
- Run:** A sidebar with tabs for 'Run and Debug' and 'Run'.
- Breakpoints:** A sidebar showing breakpoints for 'Alice', 'Student', 'Data Structures', 'Bob', 'Student', and 'Machine Learning'.

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows a folder structure for 'RUN AND DEBUG' and 'Run and Debug'.
- Search Bar:** Contains the text 'newby'.
- Code Editor:** Displays Python code for a priority queue system. The code defines a class `PriorityQueueSystem` with methods for enqueueing and dequeuing items based on priority (name). It includes a `display` method to print the queue state and a `process` method to handle multiple enqueue and dequeue operations.
- Terminal:** Shows the command line output of running the script. The output includes:
 - Priority Queue Requests:
 - (Dr. Ree, Faculty, "Machine Learning")
 - (Alice, Student, "Data Structures")
 - (Bob, Student, "DBMS")
 - Priority Queue Processing:
 - Processing: (Dr. Ree, Faculty, "Machine Learning")
 - Processing: (Alice, Student, "Data Structures")
 - Processing: (Bob, Student, "DBMS")
- Output:** Shows the results of the processing steps.
- Problems:** No problems found.
- Breakpoints:** Shows breakpoints for 'Raised Exceptions' and 'Unc caught Exceptions'.
- Bottom Status Bar:** Shows file number 74, column 17, status bar icons for Selection A, UTF-8, CRLF, Python 3.13, and Python 3.13.

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.

Tasks

1. Implement a Stack to manage support tickets.
 2. Provide the following operations:
 - o push(ticket)

- o pop()

- o peek()

3. Simulate at least five tickets being raised and resolved.

4. Use GitHub Copilot to suggest additional stack operations such as:

- o Checking whether the stack is empty
 - o Checking whether the stack is full (if applicable)

Expected Outcome

- Functional stack-based ticket management system.
 - Clear demonstration of LIFO behavior.

The screenshot shows a Windows desktop environment with the Visual Studio Code application open. The title bar reads "File Edit Selection View Go Run Terminal Help". The main workspace displays a Python file named "new.py" with the following code:

```
#!/usr/bin/env python
# coding: utf-8

class Stack:
    def __init__(self):
        self.stack = []

    def is_empty(self):
        return len(self.stack) == 0

    def is_full(self):
        return len(self.stack) == self.capacity

    def display(self):
        print("Current Tickets:")
        for t in reversed(self.stack):
            print(t)

    if __name__ == "__main__":
        helpdesk = Stack()

        helpdesk.push("Login Issue")
        helpdesk.push("WiFi Not Working")
        helpdesk.push("Email Error")
        helpdesk.push("Software Installation")
        helpdesk.push("System Crash")

        helpdesk.display()

        print("\nTop Ticket: ", helpdesk.peek())
        print("\nIs Empty: ", helpdesk.is_empty())
        print("Is Full: ", helpdesk.is_full())
```

Below the code editor, the "TERMINAL" tab is active, showing the command line output:

```
PS C:\Users\Chinmari\Downloads> cd C:\Users\Chinmari\Downloads & & C:\Users\Chinmari\AppData\Local\Microsoft\WindowsApps\python3.13.exe C:\Users\Chinmari\vscode\extensions\ms-python.debugger-2025.10.0-win32-x64\bundles\lsp\debug\Launch.py
Ticket Resolved: Login Issue
WiFi Not working
Email Error
Software Installation
System Crash

Top Ticket: System Crash
Ticket Resolved: System Crash
Ticket Resolved: Software Installation

Current Tickets:
Email Error
WiFi Not working
Login Issue

Is Empty: False
Is Full: False
PS C:\Users\Chinmari\Downloads>
```

The status bar at the bottom indicates the terminal has 14 rows of history.

Task 4: Hash Table

Objective

To implement a Hash Table and understand collision handling.

Task Description

Use AI to generate a hash table with:

- Insert
- Search
- Delete

Starter Code

class HashTable:

pass

Expected Outcome

- Collision handling using chaining
- Well-commented methods

```
newpy | RUN AND DEBUG | RUN | Run and Debug | File Edit Selection View Go Terminal Help ← → | Search | RUN AND DEBUG | RUN | Run and Debug | To customize Run and Debug, open a folder and create a launch.json file. | Debug using a terminal command or in an interactive chat. | RUN | Run and Debug | newpy | X | C:\Users\Chinmari\Downloads> python newpy.py -> class HashTable:  
1     def __init__(self, size=10):  
2         self.size = size  
3         self.table = [{} for _ in range(size)]  
4  
5     def hash_function(self, key):  
6         return hash(key) % self.size  
7  
8     def insert(self, key, value):  
9         index = self.hash_function(key)  
10        bucket = self.table[index]  
11        for i, v in enumerate(bucket):  
12            if k == key:  
13                bucket[i] = (key, value)  
14                print("Updated", key)  
15            else:  
16                bucket.append((key, value))  
17                print("Inserted", key)  
18  
19    def search(self, key):  
20        index = self.hash_function(key)  
21        bucket = self.table[index]  
22        for k, v in bucket:  
23            if k == key:  
24                return v  
25            else:  
26                return None  
27  
28    def delete(self, key):  
29        index = self.hash_function(key)  
30        bucket = self.table[index]  
31        for i, v in enumerate(bucket):  
32            if k == key:  
33                del bucket[i]  
34                print("Deleted", key)  
35            else:  
36                return None  
37  
38    def print_table(self):  
39        for i, v in enumerate(self.table):  
40            print(f"Bucket {i}: {v}")  
41  
42    def __str__(self):  
43        return str(self.table)  
44  
45    def __repr__(self):  
46        return str(self.table)  
47  
48    def __len__(self):  
49        return len(self.table)  
50  
51    def __iter__(self):  
52        for i, v in enumerate(self.table):  
53            yield i, v  
54  
55    def __getitem__(self, key):  
56        index = self.hash_function(key)  
57        bucket = self.table[index]  
58        for k, v in bucket:  
59            if k == key:  
60                return v  
61            else:  
62                raise KeyError(f"Key {key} not found")  
63  
64    def __setitem__(self, key, value):  
65        index = self.hash_function(key)  
66        bucket = self.table[index]  
67        for i, v in enumerate(bucket):  
68            if k == key:  
69                bucket[i] = (key, value)  
70            else:  
71                bucket.append((key, value))  
72  
73    def __contains__(self, key):  
74        index = self.hash_function(key)  
75        bucket = self.table[index]  
76        for k, v in bucket:  
77            if k == key:  
78                return True  
79            else:  
80                return False  
81  
82    def __eq__(self, other):  
83        if type(other) != HashTable:  
84            return False  
85        if len(self) != len(other):  
86            return False  
87        for i in range(len(self)): # O(n^2)  
88            if self[i] != other[i]:  
89                return False  
90        return True  
91  
92    def __ne__(self, other):  
93        return not self == other  
94  
95    def __hash__(self):  
96        return hash(str(self.table))  
97  
98    def __iter__(self):  
99        for i, v in enumerate(self.table):  
100            yield i, v  
101  
102    def __repr__(self):  
103        return str(self.table)  
104  
105    def __str__(self):  
106        return str(self.table)  
107  
108    def __len__(self):  
109        return len(self.table)  
110  
111    def __iter__(self):  
112        for i, v in enumerate(self.table):  
113            yield i, v  
114  
115    def __getitem__(self, key):  
116        index = self.hash_function(key)  
117        bucket = self.table[index]  
118        for k, v in bucket:  
119            if k == key:  
120                return v  
121            else:  
122                raise KeyError(f"Key {key} not found")  
123  
124    def __setitem__(self, key, value):  
125        index = self.hash_function(key)  
126        bucket = self.table[index]  
127        for i, v in enumerate(bucket):  
128            if k == key:  
129                bucket[i] = (key, value)  
130            else:  
131                bucket.append((key, value))  
132  
133    def __contains__(self, key):  
134        index = self.hash_function(key)  
135        bucket = self.table[index]  
136        for k, v in bucket:  
137            if k == key:  
138                return True  
139            else:  
140                return False  
141  
142    def __eq__(self, other):  
143        if type(other) != HashTable:  
144            return False  
145        if len(self) != len(other):  
146            return False  
147        for i in range(len(self)): # O(n^2)  
148            if self[i] != other[i]:  
149                return False  
150        return True  
151  
152    def __ne__(self, other):  
153        return not self == other  
154  
155    def __hash__(self):  
156        return hash(str(self.table))  
157  
158    def __iter__(self):  
159        for i, v in enumerate(self.table):  
160            yield i, v  
161  
162    def __repr__(self):  
163        return str(self.table)  
164  
165    def __str__(self):  
166        return str(self.table)  
167  
168    def __len__(self):  
169        return len(self.table)  
170  
171    def __iter__(self):  
172        for i, v in enumerate(self.table):  
173            yield i, v  
174  
175    def __getitem__(self, key):  
176        index = self.hash_function(key)  
177        bucket = self.table[index]  
178        for k, v in bucket:  
179            if k == key:  
180                return v  
181            else:  
182                raise KeyError(f"Key {key} not found")  
183  
184    def __setitem__(self, key, value):  
185        index = self.hash_function(key)  
186        bucket = self.table[index]  
187        for i, v in enumerate(bucket):  
188            if k == key:  
189                bucket[i] = (key, value)  
190            else:  
191                bucket.append((key, value))  
192  
193    def __contains__(self, key):  
194        index = self.hash_function(key)  
195        bucket = self.table[index]  
196        for k, v in bucket:  
197            if k == key:  
198                return True  
199            else:  
200                return False  
201  
202    def __eq__(self, other):  
203        if type(other) != HashTable:  
204            return False  
205        if len(self) != len(other):  
206            return False  
207        for i in range(len(self)): # O(n^2)  
208            if self[i] != other[i]:  
209                return False  
210        return True  
211  
212    def __ne__(self, other):  
213        return not self == other  
214  
215    def __hash__(self):  
216        return hash(str(self.table))  
217  
218    def __iter__(self):  
219        for i, v in enumerate(self.table):  
220            yield i, v  
221  
222    def __repr__(self):  
223        return str(self.table)  
224  
225    def __str__(self):  
226        return str(self.table)  
227  
228    def __len__(self):  
229        return len(self.table)  
230  
231    def __iter__(self):  
232        for i, v in enumerate(self.table):  
233            yield i, v  
234  
235    def __getitem__(self, key):  
236        index = self.hash_function(key)  
237        bucket = self.table[index]  
238        for k, v in bucket:  
239            if k == key:  
240                return v  
241            else:  
242                raise KeyError(f"Key {key} not found")  
243  
244    def __setitem__(self, key, value):  
245        index = self.hash_function(key)  
246        bucket = self.table[index]  
247        for i, v in enumerate(bucket):  
248            if k == key:  
249                bucket[i] = (key, value)  
250            else:  
251                bucket.append((key, value))  
252  
253    def __contains__(self, key):  
254        index = self.hash_function(key)  
255        bucket = self.table[index]  
256        for k, v in bucket:  
257            if k == key:  
258                return True  
259            else:  
260                return False  
261  
262    def __eq__(self, other):  
263        if type(other) != HashTable:  
264            return False  
265        if len(self) != len(other):  
266            return False  
267        for i in range(len(self)): # O(n^2)  
268            if self[i] != other[i]:  
269                return False  
270        return True  
271  
272    def __ne__(self, other):  
273        return not self == other  
274  
275    def __hash__(self):  
276        return hash(str(self.table))  
277  
278    def __iter__(self):  
279        for i, v in enumerate(self.table):  
280            yield i, v  
281  
282    def __repr__(self):  
283        return str(self.table)  
284  
285    def __str__(self):  
286        return str(self.table)  
287  
288    def __len__(self):  
289        return len(self.table)  
290  
291    def __iter__(self):  
292        for i, v in enumerate(self.table):  
293            yield i, v  
294  
295    def __getitem__(self, key):  
296        index = self.hash_function(key)  
297        bucket = self.table[index]  
298        for k, v in bucket:  
299            if k == key:  
300                return v  
301            else:  
302                raise KeyError(f"Key {key} not found")  
303  
304    def __setitem__(self, key, value):  
305        index = self.hash_function(key)  
306        bucket = self.table[index]  
307        for i, v in enumerate(bucket):  
308            if k == key:  
309                bucket[i] = (key, value)  
310            else:  
311                bucket.append((key, value))  
312  
313    def __contains__(self, key):  
314        index = self.hash_function(key)  
315        bucket = self.table[index]  
316        for k, v in bucket:  
317            if k == key:  
318                return True  
319            else:  
320                return False  
321  
322    def __eq__(self, other):  
323        if type(other) != HashTable:  
324            return False  
325        if len(self) != len(other):  
326            return False  
327        for i in range(len(self)): # O(n^2)  
328            if self[i] != other[i]:  
329                return False  
330        return True  
331  
332    def __ne__(self, other):  
333        return not self == other  
334  
335    def __hash__(self):  
336        return hash(str(self.table))  
337  
338    def __iter__(self):  
339        for i, v in enumerate(self.table):  
340            yield i, v  
341  
342    def __repr__(self):  
343        return str(self.table)  
344  
345    def __str__(self):  
346        return str(self.table)  
347  
348    def __len__(self):  
349        return len(self.table)  
350  
351    def __iter__(self):  
352        for i, v in enumerate(self.table):  
353            yield i, v  
354  
355    def __getitem__(self, key):  
356        index = self.hash_function(key)  
357        bucket = self.table[index]  
358        for k, v in bucket:  
359            if k == key:  
360                return v  
361            else:  
362                raise KeyError(f"Key {key} not found")  
363  
364    def __setitem__(self, key, value):  
365        index = self.hash_function(key)  
366        bucket = self.table[index]  
367        for i, v in enumerate(bucket):  
368            if k == key:  
369                bucket[i] = (key, value)  
370            else:  
371                bucket.append((key, value))  
372  
373    def __contains__(self, key):  
374        index = self.hash_function(key)  
375        bucket = self.table[index]  
376        for k, v in bucket:  
377            if k == key:  
378                return True  
379            else:  
380                return False  
381  
382    def __eq__(self, other):  
383        if type(other) != HashTable:  
384            return False  
385        if len(self) != len(other):  
386            return False  
387        for i in range(len(self)): # O(n^2)  
388            if self[i] != other[i]:  
389                return False  
390        return True  
391  
392    def __ne__(self, other):  
393        return not self == other  
394  
395    def __hash__(self):  
396        return hash(str(self.table))  
397  
398    def __iter__(self):  
399        for i, v in enumerate(self.table):  
400            yield i, v  
401  
402    def __repr__(self):  
403        return str(self.table)  
404  
405    def __str__(self):  
406        return str(self.table)  
407  
408    def __len__(self):  
409        return len(self.table)  
410  
411    def __iter__(self):  
412        for i, v in enumerate(self.table):  
413            yield i, v  
414  
415    def __getitem__(self, key):  
416        index = self.hash_function(key)  
417        bucket = self.table[index]  
418        for k, v in bucket:  
419            if k == key:  
420                return v  
421            else:  
422                raise KeyError(f"Key {key} not found")  
423  
424    def __setitem__(self, key, value):  
425        index = self.hash_function(key)  
426        bucket = self.table[index]  
427        for i, v in enumerate(bucket):  
428            if k == key:  
429                bucket[i] = (key, value)  
430            else:  
431                bucket.append((key, value))  
432  
433    def __contains__(self, key):  
434        index = self.hash_function(key)  
435        bucket = self.table[index]  
436        for k, v in bucket:  
437            if k == key:  
438                return True  
439            else:  
440                return False  
441  
442    def __eq__(self, other):  
443        if type(other) != HashTable:  
444            return False  
445        if len(self) != len(other):  
446            return False  
447        for i in range(len(self)): # O(n^2)  
448            if self[i] != other[i]:  
449                return False  
450        return True  
451  
452    def __ne__(self, other):  
453        return not self == other  
454  
455    def __hash__(self):  
456        return hash(str(self.table))  
457  
458    def __iter__(self):  
459        for i, v in enumerate(self.table):  
460            yield i, v  
461  
462    def __repr__(self):  
463        return str(self.table)  
464  
465    def __str__(self):  
466        return str(self.table)  
467  
468    def __len__(self):  
469        return len(self.table)  
470  
471    def __iter__(self):  
472        for i, v in enumerate(self.table):  
473            yield i, v  
474  
475    def __getitem__(self, key):  
476        index = self.hash_function(key)  
477        bucket = self.table[index]  
478        for k, v in bucket:  
479            if k == key:  
480                return v  
481            else:  
482                raise KeyError(f"Key {key} not found")  
483  
484    def __setitem__(self, key, value):  
485        index = self.hash_function(key)  
486        bucket = self.table[index]  
487        for i, v in enumerate(bucket):  
488            if k == key:  
489                bucket[i] = (key, value)  
490            else:  
491                bucket.append((key, value))  
492  
493    def __contains__(self, key):  
494        index = self.hash_function(key)  
495        bucket = self.table[index]  
496        for k, v in bucket:  
497            if k == key:  
498                return True  
499            else:  
500                return False  
501  
502    def __eq__(self, other):  
503        if type(other) != HashTable:  
504            return False  
505        if len(self) != len(other):  
506            return False  
507        for i in range(len(self)): # O(n^2)  
508            if self[i] != other[i]:  
509                return False  
510        return True  
511  
512    def __ne__(self, other):  
513        return not self == other  
514  
515    def __hash__(self):  
516        return hash(str(self.table))  
517  
518    def __iter__(self):  
519        for i, v in enumerate(self.table):  
520            yield i, v  
521  
522    def __repr__(self):  
523        return str(self.table)  
524  
525    def __str__(self):  
526        return str(self.table)  
527  
528    def __len__(self):  
529        return len(self.table)  
530  
531    def __iter__(self):  
532        for i, v in enumerate(self.table):  
533            yield i, v  
534  
535    def __getitem__(self, key):  
536        index = self.hash_function(key)  
537        bucket = self.table[index]  
538        for k, v in bucket:  
539            if k == key:  
540                return v  
541            else:  
542                raise KeyError(f"Key {key} not found")  
543  
544    def __setitem__(self, key, value):  
545        index = self.hash_function(key)  
546        bucket = self.table[index]  
547        for i, v in enumerate(bucket):  
548            if k == key:  
549                bucket[i] = (key, value)  
550            else:  
551                bucket.append((key, value))  
552  
553    def __contains__(self, key):  
554        index = self.hash_function(key)  
555        bucket = self.table[index]  
556        for k, v in bucket:  
557            if k == key:  
558                return True  
559            else:  
560                return False  
561  
562    def __eq__(self, other):  
563        if type(other) != HashTable:  
564            return False  
565        if len(self) != len(other):  
566            return False  
567        for i in range(len(self)): # O(n^2)  
568            if self[i] != other[i]:  
569                return False  
570        return True  
571  
572    def __ne__(self, other):  
573        return not self == other  
574  
575    def __hash__(self):  
576        return hash(str(self.table))  
577  
578    def __iter__(self):  
579        for i, v in enumerate(self.table):  
580            yield i, v  
581  
582    def __repr__(self):  
583        return str(self.table)  
584  
585    def __str__(self):  
586        return str(self.table)  
587  
588    def __len__(self):  
589        return len(self.table)  
590  
591    def __iter__(self):  
592        for i, v in enumerate(self.table):  
593            yield i, v  
594  
595    def __getitem__(self, key):  
596        index = self.hash_function(key)  
597        bucket = self.table[index]  
598        for k, v in bucket:  
599            if k == key:  
600                return v  
601            else:  
602                raise KeyError(f"Key {key} not found")  
603  
604    def __setitem__(self, key, value):  
605        index = self.hash_function(key)  
606        bucket = self.table[index]  
607        for i, v in enumerate(bucket):  
608            if k == key:  
609                bucket[i] = (key, value)  
610            else:  
611                bucket.append((key, value))  
612  
613    def __contains__(self, key):  
614        index = self.hash_function(key)  
615        bucket = self.table[index]  
616        for k, v in bucket:  
617            if k == key:  
618                return True  
619            else:  
620                return False  
621  
622    def __eq__(self, other):  
623        if type(other) != HashTable:  
624            return False  
625        if len(self) != len(other):  
626            return False  
627        for i in range(len(self)): # O(n^2)  
628            if self[i] != other[i]:  
629                return False  
630        return True  
631  
632    def __ne__(self, other):  
633        return not self == other  
634  
635    def __hash__(self):  
636        return hash(str(self.table))  
637  
638    def __iter__(self):  
639        for i, v in enumerate(self.table):  
640            yield i, v  
641  
642    def __repr__(self):  
643        return str(self.table)  
644  
645    def __str__(self):  
646        return str(self.table)  
647  
648    def __len__(self):  
649        return len(self.table)  
650  
651    def __iter__(self):  
652        for i, v in enumerate(self.table):  
653            yield i, v  
654  
655    def __getitem__(self, key):  
656        index = self.hash_function(key)  
657        bucket = self.table[index]  
658        for k, v in bucket:  
659            if k == key:  
660                return v  
661            else:  
662                raise KeyError(f"Key {key} not found")  
663  
664    def __setitem__(self, key, value):  
665        index = self.hash_function(key)  
666        bucket = self.table[index]  
667        for i, v in enumerate(bucket):  
668            if k == key:  
669                bucket[i] = (key, value)  
670            else:  
671                bucket.append((key, value))  
672  
673    def __contains__(self, key):  
674        index = self.hash_function(key)  
675        bucket = self.table[index]  
676        for k, v in bucket:  
677            if k == key:  
678                return True  
679            else:  
680                return False  
681  
682    def __eq__(self, other):  
683        if type(other) != HashTable:  
684            return False  
685        if len(self) != len(other):  
686            return False  
687        for i in range(len(self)): # O(n^2)  
688            if self[i] != other[i]:  
689                return False  
690        return True  
691  
692    def __ne__(self, other):  
693        return not self == other  
694  
695    def __hash__(self):  
696        return hash(str(self.table))  
697  
698    def __iter__(self):  
699        for i, v in enumerate(self.table):  
700            yield i, v  
701  
702    def __repr__(self):  
703        return str(self.table)  
704  
705    def __str__(self):  
706        return str(self.table)  
707  
708    def __len__(self):  
709        return len(self.table)  
710  
711    def __iter__(self):  
712        for i, v in enumerate(self.table):  
713            yield i, v  
714  
715    def __getitem__(self, key):  
716        index = self.hash_function(key)  
717        bucket = self.table[index]  
718        for k, v in bucket:  
719            if k == key:  
720                return v  
721            else:  
722                raise KeyError(f"Key {key} not found")  
723  
724    def __setitem__(self, key, value):  
725        index = self.hash_function(key)  
726        bucket = self.table[index]  
727        for i, v in enumerate(bucket):  
728            if k == key:  
729                bucket[i] = (key, value)  
730            else:  
731                bucket.append((key, value))  
732  
733    def __contains__(self, key):  
734        index = self.hash_function(key)  
735        bucket = self.table[index]  
736        for k, v in bucket:  
737            if k == key:  
738                return True  
739            else:  
740                return False  
741  
742    def __eq__(self, other):  
743        if type(other) != HashTable:  
744            return False  
745        if len(self) != len(other):  
746            return False  
747        for i in range(len(self)): # O(n^2)  
748            if self[i] != other[i]:  
749                return False  
750        return True  
751  
752    def __ne__(self, other):  
753        return not self == other  
754  
755    def __hash__(self):  
756        return hash(str(self.table))  
757  
758    def __iter__(self):  
759        for i, v in enumerate(self.table):  
760            yield i, v  
761  
762    def __repr__(self):  
763        return str(self.table)  
764  
765    def __str__(self):  
766        return str(self.table)  
767  
768    def __len__(self):  
769        return len(self.table)  
770  
771    def __iter__(self):  
772        for i, v in enumerate(self.table):  
773            yield i, v  
774  
775    def __getitem__(self, key):  
776        index = self.hash_function(key)  
777        bucket = self.table[index]  
778        for k, v in bucket:  
779            if k == key:  
780                return v  
781            else:  
782                raise KeyError(f"Key {key} not found")  
783  
784    def __setitem__(self, key, value):  
785        index = self.hash_function(key)  
786        bucket = self.table[index]  
787        for i, v in enumerate(bucket):  
788            if k == key:  
789                bucket[i] = (key, value)  
790            else:  
791                bucket.append((key, value))  
792  
793    def __contains__(self, key):  
794        index = self.hash_function(key)  
795        bucket = self.table[index]  
796        for k, v in bucket:  
797            if k == key:  
798                return True  
799            else:  
800                return False  
801  
802    def __eq__(self, other):  
803        if type(other) != HashTable:  
804            return False  
805        if len(self) != len(other):  
806            return False  
807        for i in range(len(self)): # O(n^2)  
808            if self[i] != other[i]:  
809                return False  
810        return True  
811  
812    def __ne__(self, other):  
813        return not self == other  
814  
815    def __hash__(self):  
816        return hash(str(self.table))  
817  
818    def __iter__(self):  
819        for i, v in enumerate(self.table):  
820            yield i, v  
821  
822    def __repr__(self):  
823        return str(self.table)  
824  
825    def __str__(self):  
826        return str(self.table)  
827  
828    def __len__(self):  
829        return len(self.table)  
830  
831    def __iter__(self):  
832        for i, v in enumerate(self.table):  
833            yield i, v  
834  
835    def __getitem__(self, key):  
836        index = self.hash_function(key)  
837        bucket = self.table[index]  
838        for k, v in bucket:  
839            if k == key:  
840                return v  
841            else:  
842                raise KeyError(f"Key {key} not found")  
843  
844    def __setitem__(self, key, value):  
845        index = self.hash_function(key)  
846        bucket = self.table[index]  
847        for i, v in enumerate(bucket):  
848            if k == key:  
849                bucket[i] = (key, value)  
850            else:  
851                bucket.append((key, value))  
852  
853    def __contains__(self, key):  
854        index = self.hash_function(key)  
855        bucket = self.table[index]  
856        for k, v in bucket:  
857            if k == key:  
858                return True  
859            else:  
860                return False  
861  
862    def __eq__(self, other):  
863        if type(other) != HashTable:  
864            return False  
865        if len(self) != len(other):  
866            return False  
867        for i in range(len(self)): # O(n^2)  
868            if self[i] != other[i]:  
869                return False  
870        return True  
871  
872    def __ne__(self, other):  
873        return not self == other  
874  
875    def __hash__(self):  
876        return hash(str(self.table))  
877  
878    def __iter__(self):  
879        for i, v in enumerate(self.table):  
880            yield i, v  
881  
882    def __repr__(self):  
883        return str(self.table)  
884  
885    def __str__(self):  
886        return str(self.table)  
887  
888    def __len__(self):  
889        return len(self.table)  
890  
891    def __iter__(self):  
892        for i, v in enumerate(self.table):  
893            yield i, v  
894  
895    def __getitem__(self, key):  
896        index = self.hash_function(key)  
897        bucket = self.table[index]  
898        for k, v in bucket:  
899            if k == key:  
900                return v  
901            else:  
902                raise KeyError(f"Key {key} not found")  
903  
904    def __setitem__(self, key, value):  
905        index = self.hash_function(key)  
906        bucket = self.table[index]  
907        for i, v in enumerate(bucket):  
908            if k == key:  
909                bucket[i] = (key, value)  
910            else:  
911                bucket.append((key, value))  
912  
913    def __contains__(self, key):  
914        index = self.hash_function(key)  
915        bucket = self.table[index]  
916        for k, v in bucket:  
917            if k == key:  
918                return True  
919            else:  
920                return False  
921  
922    def __eq__(self, other):  
923        if type(other) != HashTable:  
924            return False  
925        if len(self) != len(other):  
926            return False  
927        for i in range(len(self)): # O(n^2)  
928            if self[i] != other[i]:  
929                return False  
930        return True  
931  
932    def __ne__(self, other):  
933        return not self == other  
934  
935    def __hash__(self):  
936        return hash(str(self.table))  
937  
938    def __iter__(self):  
939        for i, v in enumerate(self.table):  
940            yield i, v  
941  
942    def __repr__(self):  
943        return str(self.table)  
944  
945    def __str__(self):  
946        return str(self.table)  
947  
948    def __len__(self):  
949        return len(self.table)  
950  
951    def __iter__(self):  
952        for i, v in enumerate(self.table):  
953            yield i, v  
954  
955    def __getitem__(self, key):  
956        index = self.hash_function(key)  
957        bucket = self.table[index]  
958        for k, v in bucket:  
959            if k == key:  
960                return v  
961            else:  
962                raise KeyError(f"Key {key} not found")  
963  
964    def __setitem__(self, key, value):  
965        index = self.hash_function(key)  
966        bucket = self.table[index]  
967        for i, v in enumerate(bucket):  
968            if k == key:  
969                bucket[i] = (
```

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

Student Tasks

1. Choose the most appropriate data structure for each feature.
 2. Justify your choice in 2–3 sentences.
 3. Implement one selected feature using AI-assisted code generation.

Expected Outcome

- Mapping table: Feature → Data Structure → Justification
 - One fully working Python implementation

The screenshot shows the Visual Studio Code interface with the following details:

- File Menu:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Run and Debug Bar:** Shows "RUN AND DEBUG" and "RUN".
- Code Editor:** Displays the content of a Python file named `new.py`. The code defines a `CafeteriaQueue` class with methods for placing orders, processing them, and displaying pending orders.
- Terminal:** Shows the command line output of running the script. It includes imports, the class definition, and the execution of various methods like `place_order`, `process_order`, and `display_orders`.
- Output Panel:** Shows standard output and error messages from the script's execution.
- Breakpoints:** A sidebar titled "BREAKPOINTS" shows three types of exceptions: Raised Exceptions, Uncought Exceptions (with a checked checkbox), and User Uncought Exceptions.
- Status Bar:** Shows the current line (Ln 38), column (Col 26), and encoding (UTF-8). It also indicates the Python version (Python 3.11) and the file type (Python).