

AI Assisted Coding

Lab 11.2 - Data Structures with AI

2303A51207

MD SUBHANI

BATCH 04

Objective:

To use AI-assisted coding to design and implement fundamental data structures in Python including Stack, Queue, Linked List, Binary Search Tree, and Hash Table.

Task 1: Stack Implementation

Prompt:

"Using AI-assisted coding, design and implement a Stack data structure in Python with push, pop, peek, and isEmpty operations."

Code:

```
class Stack:  
    def __init__(self):  
        self.stack = []  
  
    def push(self, item):  
        self.stack.append(item)  
  
    def pop(self):  
        if self.isEmpty():  
            return "Stack is empty"
```

```
return self.stack.pop()

def peek(self):
    if self.isEmpty():
        return "Stack is empty"
    return self.stack[-1]

def isEmpty(self):
    return len(self.stack) == 0
```

Explanation:

This program implements Stack using list with LIFO principle.

Output:

Top element: 20

Popped element: 20

The screenshot shows a code editor window with a Python file named '11.2.py'. The code defines a 'Stack' class with methods for push, pop, peek, and isEmpty. It includes a testing section at the bottom. Below the code editor is a terminal window showing the execution of the script and its output.

```
1 Welcome  2 11.2.py  X
1 11.2.py > ...
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
```

Task 2: Queue Implementation

Prompt:

"Design a Queue using FIFO principle with enqueue, dequeue, front and size operations."

Code:

```
class Queue:
```

```
    def __init__(self):
```

```
        self.queue = []
```

```
    def enqueue(self, item):
```

```
        self.queue.append(item)
```

```
def dequeue(self):  
    if self.isEmpty():  
        return "Queue is empty"  
    return self.queue.pop(0)
```

```
def front(self):  
    return self.queue[0]
```

```
def size(self):  
    return len(self.queue)
```

```
def isEmpty(self):  
    return len(self.queue) == 0
```

Explanation:

Queue follows FIFO principle.

Output:

Front element: 5

Dequeued element: 5

```
⚡ Welcome ⚡ 11.2.py ⚡
⚡ 11.2.py > ...
1 class Queue:
13
14     def front(self):
15         if self.isEmpty():
16             return "Queue is empty"
17         return self.queue[0]
18
19     def size(self):
20         return len(self.queue)
21
22     def isEmpty(self):
23         return len(self.queue) == 0
24
25
26 # Testing
27 q = Queue()
28 q.enqueue(5)
29 q.enqueue(15)
30 print("Front element:", q.front())
31 print("Dequeued element:", q.dequeue())
32 print("Queue size:", q.size())
```

```
PS F:\AI-A-coding> & C:/Python314/python.exe F:/AI-A-coding/11.2.py
● 5 added to queue
15 added to queue
Front element: 5
Dequeued element: 5
Queue size: 1
○ PS F:\AI-A-coding>
```

Task 3: Singly Linked List

Prompt:

"Create singly linked list with insertion and traversal."

Code:

```
class Node:
    def __init__(self,data):
        self.data=data
        self.next=None
```

```
class LinkedList:
```

```
def __init__(self):
    self.head=None

def insert(self,data):
    new_node=Node(data)
    if self.head is None:
        self.head=new_node
    else:
        temp=self.head
        while temp.next:
            temp=temp.next
        temp.next=new_node

def display(self):
    temp=self.head
    while temp:
        print(temp.data,end="->")
        temp=temp.next
```

Explanation:

Linked list connects nodes using pointers.

Output:

10->20->30->None

```

11.2.py > ...
1   class Node:
2   |   def __init__(self, data):
3   |       self.data = data
4   |       self.next = None
5
6   class LinkedList:
7   |   def __init__(self):
8   |       self.head = None
9
10  def insert(self, data):
11  |   new_node = Node(data)
12  |   if self.head is None:
13  |       self.head = new_node
14  |   else:
15  |       temp = self.head
16  |       while temp.next:
17  |           temp = temp.next
18  |       temp.next = new_node
19
20  def display(self):
21  |   temp = self.head
22  |   while temp:
23  |       print(temp.data, end=" -> ")
24  |       temp = temp.next
25  |   print("None")
26
● PS F:\AI-A-coding> & C:/Python314/python.exe f:/AI-A-coding/11.2.py
10 -> 20 -> 30 -> None
○ PS F:\AI-A-coding>

```

Task 4: Binary Search Tree

Prompt:

"Implement BST with insertion and inorder traversal."

Code:

```

class Node:

    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

```

```
def insert(root, key):
```

```
    if root is None:
```

```
return Node(key)

if key<root.val:
    root.left=insert(root.left,key)

else:
    root.right=insert(root.right,key)

return root
```

```
def inorder(root):

    if root:
        inorder(root.left)
        print(root.val,end=" ")
        inorder(root.right)
```

Explanation:

BST stores values in sorted order.

Output:

20 30 40 50 60 70 80

```
Welcome 11.2.py X
11.2.py > ...
1 class Node:
2     def __init__(self, key):
3         self.left = None
4         self.right = None
5         self.val = key
6
7     def insert(self, key):
8         if self is None:
9             return Node(key)
10        if key < self.val:
11            self.left = insert(self.left, key)
12        else:
13            self.right = insert(self.right, key)
14        return self
15
16    def inorder(self):
17        if self:
18            inorder(self.left)
19            print(self.val, end=" ")
20            inorder(self.right)
21
22
23 # Testing
24 root = None
25 elements = [50, 30, 20, 40, 70, 60, 80]
26 for e in elements:
```

● PS F:\AI-A-coding> & c:/Python314/python.exe f:/AI-A-coding/11.2.py
Inorder traversal:
20 30 40 50 60 70 80
○ PS F:\AI-A-coding>

Task 5: Hash Table

Prompt:

Create Hash Table with collision handling using chaining.

Code:

```
class HashTable:
```

```
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(size)]
```

```
    def hashFunction(self, key):
```

```
        return key % self.size
```

```
def insert(self,key,value):  
    index=self.hashFunction(key)  
    self.table[index].append((key,value))
```

```
def search(self,key):  
    index=self.hashFunction(key)  
    for pair in self.table[index]:  
        if pair[0]==key:  
            return pair[1]  
    return "Not Found"
```

```
def delete(self,key):  
    index=self.hashFunction(key)  
    for pair in self.table[index]:  
        if pair[0]==key:  
            self.table[index].remove(pair)  
            return "Deleted"  
    return "Not Found"
```

Explanation:

Hash table stores key-value pairs using hashing with chaining.

Output:

Apple

Deleted

Not Found

```
❖ Welcome ❖ 11.2.py ●
❖ 11.2.py > ...
1  class HashTable:
2      def __init__(self, size):
3          self.size = size
4          self.table = [[] for _ in range(size)]
5
6      def hashFunction(self, key):
7          return key % self.size
8
9      def insert(self, key, value):
10         index = self.hashFunction(key)
11         self.table[index].append((key, value))
12
13     def search(self, key):
14         index = self.hashFunction(key)
15         for pair in self.table[index]:
16             if pair[0] == key:
17                 return pair[1]
18         return "Not Found"
19
20     def delete(self, key):
21         index = self.hashFunction(key)
22         for pair in self.table[index]:
23             if pair[0] == key:
24                 self.table[index].remove(pair)
25                 return "Deleted"
26         return "Not Found"
27
28
29 # Testing
30 ht = HashTable(5)
31 ht.insert(1, "Apple")
32 ht.insert(6, "Banana")
33 print(ht.search(1))
34 print(ht.delete(6))
35 print(ht.search(6))
```

```
● PS F:\AI-A-coding> & C:/Python314/python.exe f:/AI-A-coding/11.2.py
Apple
Deleted
Not Found
○ PS F:\AI-A-coding>
```

Conclusion:

AI-assisted coding simplifies implementation of data structures and improves understanding of algorithms and logic.