

# ASSIGNMENT-11.4

**NAME :-P.CHARANKUMAR**

**ROLL NO :-2303A52138**

**BATCH -41**

## Task 1: Stack Implementation for Undo Operations (LIFO)

Scenario

You are building a text editor where users can undo their recent actions (typing, deleting, formatting). Each action must be reversed in Last-In-First-Out (LIFO) order.

**Prompt Used:**

# Build a Stack class for undo operations in a text editor.

# Include push, pop, peek, is\_empty with docstrings.

# Add test cases and an alternative deque-based implementation.

**AI-Generated Code**

```
4 class Stack:
5     """A simple Stack implementation for undo operations in a text editor."""
6
7     def __init__(self):
8         """Initialize an empty stack."""
9         self.stack = []
10
11    def push(self, item):
12        """Add an item to the top of the stack.
13
14        Args:
15            item: The item to be added to the stack.
16        """
17        self.stack.append(item)
18
19    def pop(self):
20        """Remove and return the item at the top of the stack.
21
22        Returns:
23            The item at the top of the stack.
24
25        Raises:
26            IndexError: If the stack is empty.
27        """
28        return self.stack.pop()
29
30    def peek(self):
31        """Return the item at the top of the stack without removing it.
32
33        Returns:
34            The item at the top of the stack.
35
36        Raises:
37            IndexError: If the stack is empty.
38        """
39        return self.stack[-1] if self.stack else None
40
41    def is_empty(self):
42        """Check if the stack is empty.
43
44        Returns:
45            bool: True if the stack is empty, False otherwise.
46        """
47        return len(self.stack) == 0
48
49    def __str__(self):
50        """Return a string representation of the stack.
51
52        Returns:
53            str: A string representation of the stack.
54        """
55        return str(self.stack)
56
57    def __repr__(self):
58        """Return a string representation of the stack.
59
60        Returns:
61            str: A string representation of the stack.
62        """
63        return f"Stack({self.stack})"
64
65    def clear(self):
66        """Clear the stack.
67
68        Returns:
69            None
70        """
71        self.stack = []
72
73    def __len__(self):
74        """Return the length of the stack.
75
76        Returns:
77            int: The length of the stack.
78        """
79        return len(self.stack)
80
81    def __iter__(self):
82        """Return an iterator over the stack.
83
84        Returns:
85            iterator: An iterator over the stack.
86        """
87        return iter(self.stack)
88
89    def __reversed__(self):
90        """Return a reversed iterator over the stack.
91
92        Returns:
93            iterator: A reversed iterator over the stack.
94        """
95        return reversed(self.stack)
96
97    def __getitem__(self, index):
98        """Return the item at the given index.
99
100       Args:
101           index: The index of the item to be retrieved.
102       """
103       return self.stack[index]
104
105    def __setitem__(self, index, value):
106        """Set the item at the given index to the given value.
107
108       Args:
109           index: The index of the item to be set.
110           value: The value to be set.
111       """
112       self.stack[index] = value
113
114    def __delitem__(self, index):
115        """Delete the item at the given index.
116
117       Args:
118           index: The index of the item to be deleted.
119       """
120       del self.stack[index]
121
122    def __contains__(self, item):
123        """Check if the item is in the stack.
124
125       Args:
126           item: The item to be checked.
127       """
128       return item in self.stack
129
130    def __eq__(self, other):
131        """Check if the stack is equal to the other stack.
132
133       Args:
134           other: The other stack to be compared.
135       """
136       return self.stack == other.stack
137
138    def __neq__(self, other):
139        """Check if the stack is not equal to the other stack.
140
141       Args:
142           other: The other stack to be compared.
143       """
144       return self.stack != other.stack
145
146    def __lt__(self, other):
147        """Check if the stack is less than the other stack.
148
149       Args:
150           other: The other stack to be compared.
151       """
152       return self.stack < other.stack
153
154    def __gt__(self, other):
155        """Check if the stack is greater than the other stack.
156
157       Args:
158           other: The other stack to be compared.
159       """
160       return self.stack > other.stack
161
162    def __le__(self, other):
163        """Check if the stack is less than or equal to the other stack.
164
165       Args:
166           other: The other stack to be compared.
167       """
168       return self.stack <= other.stack
169
170    def __ge__(self, other):
171        """Check if the stack is greater than or equal to the other stack.
172
173       Args:
174           other: The other stack to be compared.
175       """
176       return self.stack >= other.stack
177
178    def __add__(self, other):
179        """Add the other stack to the stack.
180
181       Args:
182           other: The other stack to be added.
183       """
184       self.stack += other.stack
185
186    def __sub__(self, other):
187        """Subtract the other stack from the stack.
188
189       Args:
190           other: The other stack to be subtracted.
191       """
192       self.stack -= other.stack
193
194    def __mul__(self, other):
195        """Multiply the stack by the other stack.
196
197       Args:
198           other: The other stack to be multiplied.
199       """
200       self.stack *= other.stack
201
202    def __div__(self, other):
203        """Divide the stack by the other stack.
204
205       Args:
206           other: The other stack to be divided.
207       """
208       self.stack /= other.stack
209
210    def __mod__(self, other):
211        """Modulo the stack by the other stack.
212
213       Args:
214           other: The other stack to be moduloed.
215       """
216       self.stack %= other.stack
217
218    def __divmod__(self, other):
219        """Divide the stack by the other stack and return the quotient and remainder.
220
221       Args:
222           other: The other stack to be divided.
223       """
224       return self.stack // other.stack, self.stack % other.stack
225
226    def __pow__(self, other):
227        """Power the stack by the other stack.
228
229       Args:
230           other: The other stack to be powered.
231       """
232       self.stack **= other.stack
233
234    def __radd__(self, other):
235        """Add the other stack to the stack.
236
237       Args:
238           other: The other stack to be added.
239       """
240       other.stack += self.stack
241
242    def __rsub__(self, other):
243        """Subtract the stack from the other stack.
244
245       Args:
246           other: The other stack to be subtracted.
247       """
248       other.stack -= self.stack
249
250    def __rmul__(self, other):
251        """Multiply the other stack by the stack.
252
253       Args:
254           other: The other stack to be multiplied.
255       """
256       other.stack *= self.stack
257
258    def __rdiv__(self, other):
259        """Divide the other stack by the stack.
260
261       Args:
262           other: The other stack to be divided.
263       """
264       other.stack /= self.stack
265
266    def __rmod__(self, other):
267        """Modulo the other stack by the stack.
268
269       Args:
270           other: The other stack to be moduloed.
271       """
272       other.stack %= self.stack
273
274    def __rdivmod__(self, other):
275        """Divide the other stack by the stack and return the quotient and remainder.
276
277       Args:
278           other: The other stack to be divided.
279       """
280       return other.stack // self.stack, other.stack % self.stack
281
282    def __rpow__(self, other):
283        """Power the other stack by the stack.
284
285       Args:
286           other: The other stack to be powered.
287       """
288       other.stack **= self.stack
289
290    def __iadd__(self, other):
291        """Add the other stack to the stack.
292
293       Args:
294           other: The other stack to be added.
295       """
296       self.stack += other.stack
297
298    def __isub__(self, other):
299        """Subtract the other stack from the stack.
300
301       Args:
302           other: The other stack to be subtracted.
303       """
304       self.stack -= other.stack
305
306    def __imul__(self, other):
307        """Multiply the stack by the other stack.
308
309       Args:
310           other: The other stack to be multiplied.
311       """
312       self.stack *= other.stack
313
314    def __idiv__(self, other):
315        """Divide the stack by the other stack.
316
317       Args:
318           other: The other stack to be divided.
319       """
320       self.stack /= other.stack
321
322    def __imod__(self, other):
323        """Modulo the stack by the other stack.
324
325       Args:
326           other: The other stack to be moduloed.
327       """
328       self.stack %= other.stack
329
330    def __idivmod__(self, other):
331        """Divide the stack by the other stack and return the quotient and remainder.
332
333       Args:
334           other: The other stack to be divided.
335       """
336       return self.stack // other.stack, self.stack % other.stack
337
338    def __ipow__(self, other):
339        """Power the stack by the other stack.
340
341       Args:
342           other: The other stack to be powered.
343       """
344       self.stack **= other.stack
345
346    def __iand__(self, other):
347        """Bitwise AND the stack with the other stack.
348
349       Args:
350           other: The other stack to be ANDed.
351       """
352       self.stack &= other.stack
353
354    def __ior__(self, other):
355        """Bitwise OR the stack with the other stack.
356
357       Args:
358           other: The other stack to be ORed.
359       """
360       self.stack |= other.stack
351
362    def __ixor__(self, other):
363        """Bitwise XOR the stack with the other stack.
364
365       Args:
366           other: The other stack to be XORed.
367       """
368       self.stack ^= other.stack
369
370    def __ilshift__(self, other):
371        """Left shift the stack by the other stack.
372
373       Args:
374           other: The other stack to be shifted.
375       """
376       self.stack <<= other.stack
377
378    def __irshift__(self, other):
379        """Right shift the stack by the other stack.
380
381       Args:
382           other: The other stack to be shifted.
383       """
384       self.stack >>= other.stack
385
386    def __iandl__(self, other):
387        """Left shift the stack by the other stack.
388
389       Args:
390           other: The other stack to be shifted.
391       """
392       self.stack <<= other.stack
393
394    def __irshl__(self, other):
395        """Right shift the stack by the other stack.
396
397       Args:
398           other: The other stack to be shifted.
399       """
400       self.stack >>= other.stack
401
402    def __iandl__(self, other):
403        """Left shift the stack by the other stack.
404
405       Args:
406           other: The other stack to be shifted.
407       """
408       self.stack <<= other.stack
409
410    def __irshl__(self, other):
411        """Right shift the stack by the other stack.
412
413       Args:
414           other: The other stack to be shifted.
415       """
416       self.stack >>= other.stack
417
418    def __iandl__(self, other):
419        """Left shift the stack by the other stack.
420
421       Args:
422           other: The other stack to be shifted.
423       """
424       self.stack <<= other.stack
425
426    def __irshl__(self, other):
427        """Right shift the stack by the other stack.
428
429       Args:
430           other: The other stack to be shifted.
431       """
432       self.stack >>= other.stack
433
434    def __iandl__(self, other):
435        """Left shift the stack by the other stack.
436
437       Args:
438           other: The other stack to be shifted.
439       """
440       self.stack <<= other.stack
441
442    def __irshl__(self, other):
443        """Right shift the stack by the other stack.
444
445       Args:
446           other: The other stack to be shifted.
447       """
448       self.stack >>= other.stack
449
450    def __iandl__(self, other):
451        """Left shift the stack by the other stack.
452
453       Args:
454           other: The other stack to be shifted.
455       """
456       self.stack <<= other.stack
457
458    def __irshl__(self, other):
459        """Right shift the stack by the other stack.
460
461       Args:
462           other: The other stack to be shifted.
463       """
464       self.stack >>= other.stack
465
466    def __iandl__(self, other):
467        """Left shift the stack by the other stack.
468
469       Args:
470           other: The other stack to be shifted.
471       """
472       self.stack <<= other.stack
473
474    def __irshl__(self, other):
475        """Right shift the stack by the other stack.
476
477       Args:
478           other: The other stack to be shifted.
479       """
480       self.stack >>= other.stack
481
482    def __iandl__(self, other):
483        """Left shift the stack by the other stack.
484
485       Args:
486           other: The other stack to be shifted.
487       """
488       self.stack <<= other.stack
489
490    def __irshl__(self, other):
491        """Right shift the stack by the other stack.
492
493       Args:
494           other: The other stack to be shifted.
495       """
496       self.stack >>= other.stack
497
498    def __iandl__(self, other):
499        """Left shift the stack by the other stack.
500
501       Args:
502           other: The other stack to be shifted.
503       """
504       self.stack <<= other.stack
505
506    def __irshl__(self, other):
507        """Right shift the stack by the other stack.
508
509       Args:
510           other: The other stack to be shifted.
511       """
512       self.stack >>= other.stack
513
514    def __iandl__(self, other):
515        """Left shift the stack by the other stack.
516
517       Args:
518           other: The other stack to be shifted.
519       """
520       self.stack <<= other.stack
521
522    def __irshl__(self, other):
523        """Right shift the stack by the other stack.
524
525       Args:
526           other: The other stack to be shifted.
527       """
528       self.stack >>= other.stack
529
530    def __iandl__(self, other):
531        """Left shift the stack by the other stack.
532
533       Args:
534           other: The other stack to be shifted.
535       """
536       self.stack <<= other.stack
537
538    def __irshl__(self, other):
539        """Right shift the stack by the other stack.
540
541       Args:
542           other: The other stack to be shifted.
543       """
544       self.stack >>= other.stack
545
546    def __iandl__(self, other):
547        """Left shift the stack by the other stack.
548
549       Args:
550           other: The other stack to be shifted.
551       """
552       self.stack <<= other.stack
553
554    def __irshl__(self, other):
555        """Right shift the stack by the other stack.
556
557       Args:
558           other: The other stack to be shifted.
559       """
560       self.stack >>= other.stack
561
562    def __iandl__(self, other):
563        """Left shift the stack by the other stack.
564
565       Args:
566           other: The other stack to be shifted.
567       """
568       self.stack <<= other.stack
569
570    def __irshl__(self, other):
571        """Right shift the stack by the other stack.
572
573       Args:
574           other: The other stack to be shifted.
575       """
576       self.stack >>= other.stack
577
578    def __iandl__(self, other):
579        """Left shift the stack by the other stack.
580
581       Args:
582           other: The other stack to be shifted.
583       """
584       self.stack <<= other.stack
585
586    def __irshl__(self, other):
587        """Right shift the stack by the other stack.
588
589       Args:
590           other: The other stack to be shifted.
591       """
592       self.stack >>= other.stack
593
594    def __iandl__(self, other):
595        """Left shift the stack by the other stack.
596
597       Args:
598           other: The other stack to be shifted.
599       """
600       self.stack <<= other.stack
601
602    def __irshl__(self, other):
603        """Right shift the stack by the other stack.
604
605       Args:
606           other: The other stack to be shifted.
607       """
608       self.stack >>= other.stack
609
610    def __iandl__(self, other):
611        """Left shift the stack by the other stack.
612
613       Args:
614           other: The other stack to be shifted.
615       """
616       self.stack <<= other.stack
617
618    def __irshl__(self, other):
619        """Right shift the stack by the other stack.
620
621       Args:
622           other: The other stack to be shifted.
623       """
624       self.stack >>= other.stack
625
626    def __iandl__(self, other):
627        """Left shift the stack by the other stack.
628
629       Args:
630           other: The other stack to be shifted.
631       """
632       self.stack <<= other.stack
633
634    def __irshl__(self, other):
635        """Right shift the stack by the other stack.
636
637       Args:
638           other: The other stack to be shifted.
639       """
640       self.stack >>= other.stack
641
642    def __iandl__(self, other):
643        """Left shift the stack by the other stack.
644
645       Args:
646           other: The other stack to be shifted.
647       """
648       self.stack <<= other.stack
649
650    def __irshl__(self, other):
651        """Right shift the stack by the other stack.
652
653       Args:
654           other: The other stack to be shifted.
655       """
656       self.stack >>= other.stack
657
658    def __iandl__(self, other):
659        """Left shift the stack by the other stack.
660
661       Args:
662           other: The other stack to be shifted.
663       """
664       self.stack <<= other.stack
665
666    def __irshl__(self, other):
667        """Right shift the stack by the other stack.
668
669       Args:
670           other: The other stack to be shifted.
671       """
672       self.stack >>= other.stack
673
674    def __iandl__(self, other):
675        """Left shift the stack by the other stack.
676
677       Args:
678           other: The other stack to be shifted.
679       """
680       self.stack <<= other.stack
681
682    def __irshl__(self, other):
683        """Right shift the stack by the other stack.
684
685       Args:
686           other: The other stack to be shifted.
687       """
688       self.stack >>= other.stack
689
690    def __iandl__(self, other):
691        """Left shift the stack by the other stack.
692
693       Args:
694           other: The other stack to be shifted.
695       """
696       self.stack <<= other.stack
697
698    def __irshl__(self, other):
699        """Right shift the stack by the other stack.
700
701       Args:
702           other: The other stack to be shifted.
703       """
704       self.stack >>= other.stack
705
706    def __iandl__(self, other):
707        """Left shift the stack by the other stack.
708
709       Args:
710           other: The other stack to be shifted.
711       """
712       self.stack <<= other.stack
713
714    def __irshl__(self, other):
715        """Right shift the stack by the other stack.
716
717       Args:
718           other: The other stack to be shifted.
719       """
720       self.stack >>= other.stack
721
722    def __iandl__(self, other):
723        """Left shift the stack by the other stack.
724
725       Args:
726           other: The other stack to be shifted.
727       """
728       self.stack <<= other.stack
729
730    def __irshl__(self, other):
731        """Right shift the stack by the other stack.
732
733       Args:
734           other: The other stack to be shifted.
735       """
736       self.stack >>= other.stack
737
738    def __iandl__(self, other):
739        """Left shift the stack by the other stack.
740
741       Args:
742           other: The other stack to be shifted.
743       """
744       self.stack <<= other.stack
745
746    def __irshl__(self, other):
747        """Right shift the stack by the other stack.
748
749       Args:
750           other: The other stack to be shifted.
751       """
752       self.stack >>= other.stack
753
754    def __iandl__(self, other):
755        """Left shift the stack by the other stack.
756
757       Args:
758           other: The other stack to be shifted.
759       """
760       self.stack <<= other.stack
761
762    def __irshl__(self, other):
763        """Right shift the stack by the other stack.
764
765       Args:
766           other: The other stack to be shifted.
767       """
768       self.stack >>= other.stack
769
770    def __iandl__(self, other):
771        """Left shift the stack by the other stack.
772
773       Args:
774           other: The other stack to be shifted.
775       """
776       self.stack <<= other.stack
777
778    def __irshl__(self, other):
779        """Right shift the stack by the other stack.
780
781       Args:
782           other: The other stack to be shifted.
783       """
784       self.stack >>= other.stack
785
786    def __iandl__(self, other):
787        """Left shift the stack by the other stack.
788
789       Args:
790           other: The other stack to be shifted.
791       """
792       self.stack <<= other.stack
793
794    def __irshl__(self, other):
795        """Right shift the stack by the other stack.
796
797       Args:
798           other: The other stack to be shifted.
799       """
800       self.stack >>= other.stack
801
802    def __iandl__(self, other):
803        """Left shift the stack by the other stack.
804
805       Args:
806           other: The other stack to be shifted.
807       """
808       self.stack <<= other.stack
809
810    def __irshl__(self, other):
811        """Right shift the stack by the other stack.
812
813       Args:
814           other: The other stack to be shifted.
815       """
816       self.stack >>= other.stack
817
818    def __iandl__(self, other):
819        """Left shift the stack by the other stack.
820
821       Args:
822           other: The other stack to be shifted.
823       """
824       self.stack <<= other.stack
825
826    def __irshl__(self, other):
827        """Right shift the stack by the other stack.
828
829       Args:
830           other: The other stack to be shifted.
831       """
832       self.stack >>= other.stack
833
834    def __iandl__(self, other):
835        """Left shift the stack by the other stack.
836
837       Args:
838           other: The other stack to be shifted.
839       """
840       self.stack <<= other.stack
841
842    def __irshl__(self, other):
843        """Right shift the stack by the other stack.
844
845       Args:
846           other: The other stack to be shifted.
847       """
848       self.stack >>= other.stack
849
850    def __iandl__(self, other):
851        """Left shift the stack by the other stack.
852
853       Args:
854           other: The other stack to be shifted.
855       """
856       self.stack <<= other.stack
857
858    def __irshl__(self, other):
859        """Right shift the stack by the other stack.
860
861       Args:
862           other: The other stack to be shifted.
863       """
864       self.stack >>= other.stack
865
866    def __iandl__(self, other):
867        """Left shift the stack by the other stack.
868
869       Args:
870           other: The other stack to be shifted.
871       """
872       self.stack <<= other.stack
873
874    def __irshl__(self, other):
875        """Right shift the stack by the other stack.
876
877       Args:
878           other: The other stack to be shifted.
879       """
880       self.stack >>= other.stack
881
882    def __iandl__(self, other):
883        """Left shift the stack by the other stack.
884
885       Args:
886           other: The other stack to be shifted.
887       """
888       self.stack <<= other.stack
889
890    def __irshl__(self, other):
891        """Right shift the stack by the other stack.
892
893       Args:
894           other: The other stack to be shifted.
895       """
896       self.stack >>= other.stack
897
898    def __iandl__(self, other):
899        """Left shift the stack by the other stack.
900
901       Args:
902           other: The other stack to be shifted.
903       """
904       self.stack <<= other.stack
905
906    def __irshl__(self, other):
907        """Right shift the stack by the other stack.
908
909       Args:
910           other: The other stack to be shifted.
911       """
912       self.stack >>= other.stack
913
914    def __iandl__(self, other):
915        """Left shift the stack by the other stack.
916
917       Args:
918           other: The other stack to be shifted.
919       """
920       self.stack <<= other.stack
921
922    def __irshl__(self, other):
923        """Right shift the stack by the other stack.
924
925       Args:
926           other: The other stack to be shifted.
927       """
928       self.stack >>= other.stack
929
930    def __iandl__(self, other):
931        """Left shift the stack by the other stack.
932
933       Args:
934           other: The other stack to be shifted.
935       """
936       self.stack <<= other.stack
937
938    def __irshl__(self, other):
939        """Right shift the stack by the other stack.
940
941       Args:
942           other: The other stack to be shifted.
943       """
944       self.stack >>= other.stack
945
946    def __iandl__(self, other):
947        """Left shift the stack by the other stack.
948
949       Args:
950           other: The other stack to be shifted.
951       """
952       self.stack <<= other.stack
953
954    def __irshl__(self, other):
955        """Right shift the stack by the other stack.
956
957       Args:
958           other: The other stack to be shifted.
959       """
960       self.stack >>= other.stack
961
962    def __iandl__(self, other):
963        """Left shift the stack by the other stack.
964
965       Args:
966           other: The other stack to be shifted.
967       """
968       self.stack <<= other.stack
969
970    def __irshl__(self, other):
971        """Right shift the stack by the other stack.
972
973       Args:
974           other: The other stack to be shifted.
975       """
976       self.stack >>= other.stack
977
978    def __iandl__(self, other):
979        """Left shift the stack by the other stack.
980
981       Args:
982           other: The other stack to be shifted.
983       """
984       self.stack <<= other.stack
985
986    def __irshl__(self, other):
987        """Right shift the stack by the other stack.
988
989       Args:
990           other: The other stack to be shifted.
991       """
992       self.stack >>= other.stack
993
994    def __iandl__(self, other):
995        """Left shift the stack by the other stack.
996
997       Args:
998           other: The other stack to be shifted.
999       """
1000       self.stack <<= other.stack
1001
1002    def __irshl__(self, other):
1003        """Right shift the stack by the other stack.
1004
1005       Args:
1006           other: The other stack to be shifted.
1007       """
1008       self.stack >>= other.stack
1009
1010    def __iandl__(self, other):
1011        """Left shift the stack by the other stack.
1012
1013       Args:
1014           other: The other stack to be shifted.
1015       """
1016       self.stack <<= other.stack
1017
1018    def __irshl__(self, other):
1019        """Right shift the stack by the other stack.
1020
1021       Args:
1022           other: The other stack to be shifted.
1023       """
1024       self.stack >>= other.stack
1025
1026    def __iandl__(self, other):
1027        """Left shift the stack by the other stack.
1028
1029       Args:
1030           other: The other stack to be shifted.
1031       """
1032       self.stack <<= other.stack
1033
1034    def __irshl__(self, other):
1035        """Right shift the stack by the other stack.
1036
1037       Args:
1038           other: The other stack to be shifted.
1039       """
1040       self.stack >>= other.stack
1041
1042    def __iandl__(self, other):
1043        """Left shift the stack by the other stack.
1044
1045       Args:
1046           other: The other stack to be shifted.
1047       """
1048       self.stack <<= other.stack
1049
1050    def __irshl__(self, other):
1051        """Right shift the stack by the other stack.
1052
1053       Args:
1054           other: The other stack to be shifted.
1055       """
1056       self.stack >>= other.stack
1057
1058    def __iandl__(self, other):
1059        """Left shift the stack by the other stack.
1060
1061       Args:
1062           other: The other stack to be shifted.
1063       """
1064       self.stack <<= other.stack
1065
1066    def __irshl__(self, other):
1067        """Right shift the stack by the other stack.
1068
1069       Args:
1070           other: The other stack to be shifted.
1071       """
1072       self.stack >>= other.stack
1073
1074    def __iandl__(self, other):
1075        """Left shift the stack by the other stack.
1076
1077       Args:
1078           other: The other stack to be shifted.
1079       """
1080       self.stack <<= other.stack
1081
1082    def __irshl__(self, other):
1083        """Right shift the stack by the other stack.
1084
1085       Args:
1086           other: The other stack to be shifted.
1087       """
1088       self.stack >>= other.stack
1089
1090    def __iandl__(self, other):
1091        """Left shift the stack by the other stack.
1092
1093       Args:
1094           other: The other stack to be shifted.
1095       """
1096       self.stack <<= other.stack
1097
1098    def __irshl__(self, other):
1099        """Right shift the stack by the other stack.
1100
1101       Args:
1102           other: The other stack to be shifted.
1103       """
1104       self.stack >>= other.stack
1105
1106    def __iandl__(self, other):
1107        """Left shift the stack by the other stack.
1108
1109       Args:
1110           other: The other stack to be shifted.
1111       """
1112       self.stack <<= other.stack
1113
1114    def __irshl__(self, other):
1115        """Right shift the stack by the other stack.
1116
1117       Args:
1118           other: The other stack to be shifted.
1119       """
1120       self.stack >>= other.stack
1121
1122    def __iandl__(self, other):
1123        """Left shift the stack by the other stack.
1124
1125       Args:
1126           other: The other stack to be shifted.
1127       """
1128       self.stack <<= other.stack
1129
1130    def __irshl__(self, other):
1131        """Right shift the stack by the other stack.
1132
1133       Args:
1134           other: The other stack to be shifted.
1135       """
1136       self.stack >>= other.stack
1137
1138    def __iandl__(self, other):
1139        """Left shift the stack by the other stack.
1140
1141       Args:
1142           other: The other stack to be shifted.
1143       """
1144       self.stack <<= other.stack
1145
1146    def __irshl__(self, other):
1147        """Right shift the stack by the other stack.
1148
1149       Args:
1150           other: The other stack to be shifted.
1151       """
1152       self.stack >>= other.stack
1153
1154    def __iandl__(self, other):
1155        """Left shift the stack by the other stack.
1156
1157       Args:
1158           other: The other stack to be shifted.
1159       """
1160       self.stack <<= other.stack
1161
1162    def __irshl__(self, other):
1163        """Right shift the stack by the other stack.
1164
1165       Args:
1166           other: The other stack to be shifted.
1167       """
1168       self.stack >>= other.stack
1169
1170    def __iandl__(self, other):
1171        """Left shift the stack by the other stack.
1172
1173       Args:
1174           other: The other stack to be shifted.
1175       """
1176       self.stack <<= other.stack
1177
1178    def __irshl__(self, other):
1179        """Right shift the stack by the other stack.
1180
1181       Args:
1182           other: The other stack to be shifted.
1183       """
1184       self.stack >>= other.stack
1185
1186    def __iandl__(self, other):
1187        """Left shift the stack by the other stack.
1188
1189       Args:
1190           other: The other stack to be shifted.
1191       """
1192       self.stack <<= other.stack
1193
1194    def __irshl__(self, other):
1195        """Right shift the stack by the other stack.
1196
1197       Args:
1198           other: The other stack to
```

```
Assets > #You are helping me build an undo featur.py > ...
4 class Stack:
19     def pop(self):
25         Raises:
26             IndexError: If the stack is empty.
27         """
28         if self.is_empty():
29             raise IndexError("Pop from an empty stack")
30         return self.stack.pop()
31
32     def peek(self):
33         """Return the item at the top of the stack without removing it.
34
35         Returns:
36             The item at the top of the stack.
37
38         Raises:
39             IndexError: If the stack is empty.
40         """
41         if self.is_empty():
42             raise IndexError("Peek from an empty stack")
43         return self.stack[-1]
44
45     def is_empty(self):
46         """Check if the stack is empty.
```

```
4 class Stack:
5     def is_empty(self):
6         """Check if the stack is empty.
7
8         Returns:
9             True if the stack is empty, False otherwise.
10        """
11        return len(self.stack) == 0
12
13    # Test cases for the Stack class
14    if __name__ == "__main__":
15        stack = Stack()
16        stack.push("First action")
17        stack.push("Second action")
18        stack.push("Third action")
19
20        print(stack.peek()) # Output: Third action
21        print(stack.pop()) # Output: Third action
22        print(stack.peek()) # Output: Second action
23        print(stack.is_empty()) # Output: False
24        stack.pop()
25        stack.pop()
26        print(stack.is_empty()) # Output: True
```

**Output:-**

```
PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game> & C:/Users/HARSHAVARDHAN/AppData/Local/Programs/Python/Python315/python.exe "c:/Users/HARSHAVARDHAN/Downloads/Snake Game/Snake Game/Assets/#You are helping me build an undo featur.py"
Third action
Third action
Second action
False
True
```

**Explanation:**

The code defines a `Stack` class that implements a simple stack data structure, which is commonly used for undo operations in text editors. The stack supports four main operations: `push`, `pop`, `peek`, and `is\_empty`.

## Task 2: Queue for Customer Service Requests (FIFO)

### Scenario

You are developing a customer support system where service requests must be handled in the order they arrive.

### Prompt Used:

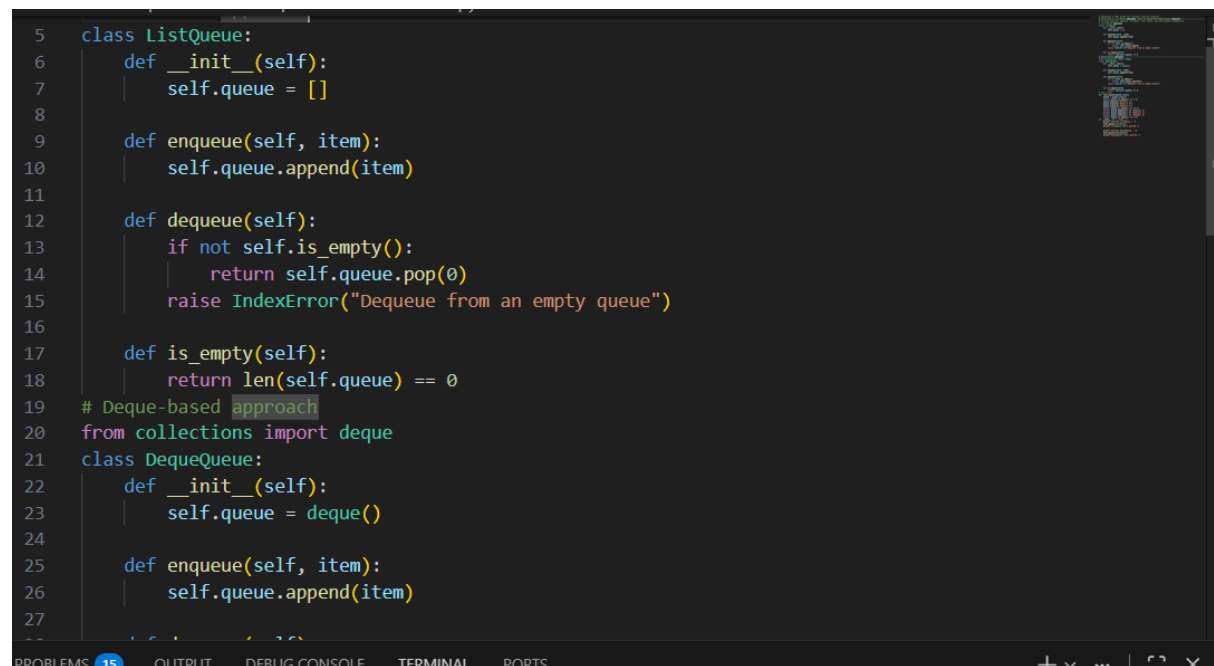
# Implement a FIFO queue for customer service requests.

# First use a list-based approach, then an optimized deque-based approach.

# Include enqueue, dequeue, is\_empty, test cases, and performance comparison.

# List-based approach

### AI-Generated Code



```
5 class ListQueue:
6     def __init__(self):
7         self.queue = []
8
9     def enqueue(self, item):
10        self.queue.append(item)
11
12    def dequeue(self):
13        if not self.is_empty():
14            return self.queue.pop(0)
15        raise IndexError("Dequeue from an empty queue")
16
17    def is_empty(self):
18        return len(self.queue) == 0
19
20    # Deque-based approach
21    from collections import deque
22    class DequeQueue:
23        def __init__(self):
24            self.queue = deque()
25
26        def enqueue(self, item):
27            self.queue.append(item)
```

```

21 class DequeueQueue:
22     def dequeue(self):
23         if not self.is_empty():
24             return self.queue.popleft()
25         raise IndexError("Dequeue from an empty queue")
26
27     def is_empty(self):
28         return len(self.queue) == 0
29
30 # Test cases
31 def test_queue(queue_class):
32     queue = queue_class()
33     assert queue.is_empty() == True
34     queue.enqueue("Request 1")
35     queue.enqueue("Request 2")
36     queue.enqueue("Request 3")
37     assert queue.is_empty() == False
38     assert queue.dequeue() == "Request 1"
39     assert queue.dequeue() == "Request 2"
40     assert queue.dequeue() == "Request 3"
41     assert queue.is_empty() == True
42
43 if __name__ == "__main__":
44     print("Testing ListQueue...")
45     test_queue(ListQueue)

```

```

    queue.enqueue("Request 2")
    queue.enqueue("Request 3")
    assert queue.is_empty() == False
    assert queue.dequeue() == "Request 1"
    assert queue.dequeue() == "Request 2"
    assert queue.dequeue() == "Request 3"
    assert queue.is_empty() == True
if __name__ == "__main__":
    print("Testing ListQueue...")
    test_queue(ListQueue)
    print("ListQueue tests passed.")

    print("Testing DequeueQueue...")
    test_queue(DequeueQueue)
    print("DequeueQueue tests passed.")

```

```

# Performance comparison
import time
def performance_comparison():
    list_queue = ListQueue()
    deque_queue = DequeQueue()
    num_operations = 100000

    # Test ListQueue
    start_time = time.time()
    for i in range(num_operations):
        list_queue.enqueue(f"Request {i}")
    for i in range(num_operations):
        list_queue.dequeue()
    list_time = time.time() - start_time

    # Test DequeQueue
    start_time = time.time()
    for i in range(num_operations):
        deque_queue.enqueue(f"Request {i}")
    for i in range(num_operations):
        deque_queue.dequeue()
    deque_time = time.time() - start_time

```

```

list_queue.dequeue()
list_time = time.time() - start_time

# Test DequeQueue
start_time = time.time()
for i in range(num_operations):
    deque_queue.enqueue(f"Request {i}")
for i in range(num_operations):
    deque_queue.dequeue()
deque_time = time.time() - start_time

print(f"ListQueue time: {list_time:.4f} seconds")
print(f"DequeQueue time: {deque_time:.4f} seconds")
if __name__ == "__main__":
    performance_comparison()

```

## Output:-

```

PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game> & C:/Users/HARSHAVARDHAN/AppData/Local/Programs/Python/Python315/python.exe "c:/Users/HARSHAVARDHAN/Downloads/Snake Game/Snake Game/Assets/# Implement a FIFO queue for customer se.py"
Testing ListQueue...
ListQueue tests passed.
Testing DequeQueue...
DequeQueue tests passed.
ListQueue time: 9.1778 seconds
DequeQueue time: 0.0194 seconds

```

## Explanation:

The code implements two versions of a FIFO (First-In-First-Out) queue for customer service requests: one using a list and the other using a deque from the collections module.

## Task 3: Singly Linked List for Dynamic Playlist Management

## Scenario

You are designing a music playlist feature where songs can be added or removed dynamically while maintaining order.

### Prompt Used:

# Implement a singly linked list for a music playlist.

# Support insert at end, delete by value, and traversal.

# Add comments explaining pointer manipulation and edge cases.

### AI-Generated Code

```
assets > # Implement a singly linked list for a m.py > MusicPlaylist > Insert_at_End
8 class MusicPlaylist:
20     def delete_by_value(self, title):
26         previous.next = current.next # Bypass the current node
27     else: # If it's the head node
28         self.head = current.next # Move the head to the next node
29         return # Exit after deleting the song
30     previous = current # Move the previous pointer to the current node
31     current = current.next # Move to the next node
32     def traverse(self):
33         current = self.head # Start from the head of the playlist
34         while current: # Traverse through the playlist
35             print(current.title) # Print the title of the current song
36             current = current.next # Move to the next node
37 # Example usage:
38 playlist = MusicPlaylist()
39 playlist.insert_at_end("Song A")
40 playlist.insert_at_end("Song B")
41 playlist.insert_at_end("Song C")
42 print("Playlist after inserting songs:")
43 playlist.traverse()
44 playlist.delete_by_value("Song B")
45 print("\nPlaylist after deleting 'Song B':")
46 playlist.traverse()
```

### Output:-

```
PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game> & C:/Users/HARSHAVARDHAN/AppData/Local/Programs/Python/Python315/python.exe "c:/Users/HARSHAVARDHAN/Downloads/Snake Game/Snake Game/Assets/# Implement a singly linked list for a m.py"
• Playlist after inserting songs:
Song A
Song B
Song C

Playlist after deleting 'Song B':
Song A
Song C
PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game>
```

### Explanation:

The code defines a singly linked list to represent a music playlist. Each song is represented as a node (SongNode) that contains the song title and a pointer to the next song. The MusicPlaylist class provides methods to insert songs at the end of

the playlist, delete songs by their title, and traverse the playlist to display all songs. The code also includes comments explaining the pointer manipulation and edge cases, such as handling an empty playlist or deleting the head node.

## Task 4: Binary Search Tree for Fast Record Lookup

### Scenario

You are building a student record system where quick searching by roll number is required.

### Prompt Used:

- # Implement a Binary Search Tree (BST) for a student record system.
- # Support `insert(value)`, `search(value)`, and `inorder_traversal()`.
- # Start from a partially written `Node` and `BST` class.
- # Complete the missing methods using recursion.
- # Maintain BST property:  $\text{left} < \text{root} < \text{right}$ .
- # Add meaningful docstrings and inline comments explaining left/right pointer updates.
- # Explain how BST improves search efficiency compared to linear search.
- # Include test cases for insertion, searching, and inorder traversal.
- # Discuss best-case, average-case, and worst-case time complexity.

### AI-Generated Code:

```
sets > # Implement a Binary Search Tree (BST) f.py > BinarySearchTree > insert

10 class TreeNode:
11     def __init__(self, value):
12         self.value = value
13         self.left = None # Pointer to the left child
14         self.right = None # Pointer to the right child
15 class BinarySearchTree:
16     def __init__(self):
17         self.root = None # Initialize the root of the BST
18     def insert(self, value):
19         """Insert a new value into the BST while maintaining the BST property."""
20         if self.root is None:
21             self.root = TreeNode(value) # If tree is empty, set root to new node
22         else:
23             self._insert_recursive(self.root, value) # Otherwise, insert recursively
24     def _insert_recursive(self, node, value):
25         """Helper method to insert a value recursively."""
26         if value < node.value:
27             if node.left is None:
28                 node.left = TreeNode(value) # Insert as left child
29             else:
30                 self._insert_recursive(node.left, value) # Recur left
31         else:
32             if node.right is None:
```

```
sets > # Implement a Binary Search Tree (BST) f.py > BinarySearchTree > insert

15 class BinarySearchTree:
24     def _insert_recursive(self, node, value):
31         else:
32             (method) def _search_recursive(
33                 self: Self@BinarySearchTree,
34                 node: Any,
35                 value: Any
36             ) -> (Any | bool)
37             Helper method to search for a value recursively.
38             value)
39     def _search_recursive(self, node, value):
40         """Helper method to search for a value recursively."""
41         if node is None:
42             return False # Value not found
43         if node.value == value:
44             return True # Value found
45         elif value < node.value:
46             return self._search_recursive(node.left, value) # Search left subtree
47         else:
48             return self._search_recursive(node.right, value) # Search right subtree
49     def inorder_traversal(self):
50         """Perform an inorder traversal of the BST and return a list of values."""
51         result = []
```



```

1     result = []
2     self._inorder_recursive(self.root, result)
3     return result
4     def _inorder_recursive(self, node, result):
5         """Helper method to perform inorder traversal recursively."""
6         if node is not None:
7             self._inorder_recursive(node.left, result) # Traverse left subtree
8             result.append(node.value) # Visit current node
9             self._inorder_recursive(node.right, result) # Traverse right subtree
10
11 # Test cases
12 if __name__ == "__main__":
13     bst = BinarySearchTree()
14     bst.insert(10)
15     bst.insert(5)
16     bst.insert(15)
17     bst.insert(3)
18     bst.insert(7)
19     bst.insert(12)
20     bst.insert(18)
21
22 # Test search

```

```

assets > # Implement a Binary Search Tree (BST) f.py > BinarySearchTree > insert
64     bst.insert(5)
65     bst.insert(15)
66     bst.insert(3)
67     bst.insert(7)
68     bst.insert(12)
69     bst.insert(18)
70
71     # Test search
72     print(bst.search(7)) # Output: True
73     print(bst.search(4)) # Output: False
74
75     # Test inorder traversal
76     print(bst.inorder_traversal()) # Output: [3, 5, 7, 10, 12, 15, 18]
77

```

## Output:

```

PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game> & C:/Users/HARSHAVARDHAN/AppData/Local/Programs/Python/Python315/python.exe "c:/Users/HARSHAVARDHAN/Downloads/Snake Game/Snake Game/Assets/# Implement a Binary Search Tree (BST) f.py"
True
False
[3, 5, 7, 10, 12, 15, 18]
PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game>

```

## Explanation:

The code defines a Binary Search Tree (BST) data structure for managing student records. The `TreeNode` class represents each node in the tree, containing a value and pointers to the left and right children. The `BinarySearchTree` class provides methods for inserting values, searching for values, and performing an inorder traversal. The `insert` method adds new values while maintaining the BST property,

ensuring that left children are less than the parent node and right children are greater. The search method checks for the presence of a value in the tree, and the `inorder_traversal` method returns a sorted list of values in the tree. The test cases demonstrate the functionality of the BST by inserting values, searching for existing and non-existing values, and performing an inorder traversal to display the sorted order of the values. The BST improves search efficiency compared to linear search by allowing it to skip large portions of the tree based on comparisons, resulting in an average-case time complexity of  $O(\log n)$  for balanced trees, while linear search has a time complexity of  $O(n)$ . However, in the worst case (e.g., when the tree is skewed), the time complexity can degrade to  $O(n)$ .

## Task 5: Graph Traversal for Social Network Connections

### Scenario

You are modeling a social network, where users are connected to friends, and you want to explore connections.

### Prompt Used:

```
# Implement a graph to model social network connections using an adjacency list.
# Each user is a node, and friendships are edges.
# Implement graph traversal algorithms:
# - Breadth-First Search (BFS) to find nearby connections
# - Depth-First Search (DFS) to explore deep connection paths
# Add inline comments explaining each step of BFS and DFS traversal.
# Explain how queues are used in BFS and stacks/recursion are used in DFS.
# Implement DFS using:
# - Recursive approach
# - Iterative approach (using an explicit stack)
# Compare recursive vs iterative DFS in terms of memory and use cases.
# Explain practical real-world use cases of BFS vs DFS in social networks.
# Include sample test cases demonstrating:
# - BFS traversal from a given user
# - DFS traversal from a given user
# Keep the code clean, readable, and beginner-friendly.
```

### AI-Generated Code

```

# Keep the code clean, readable, and beginner-friendly.
class Graph:
    def __init__(self):
        self.adjacency_list = {}

    def add_user(self, user):
        if user not in self.adjacency_list:
            self.adjacency_list[user] = []

    def add_friendship(self, user1, user2):
        self.add_user(user1)
        self.add_user(user2)
        self.adjacency_list[user1].append(user2)
        self.adjacency_list[user2].append(user1)

    def bfs(self, start_user):
        visited = set() # To keep track of visited nodes
        queue = [start_user] # Queue for BFS
        bfs_result = []

        while queue:
            current_user = queue.pop(0) # Dequeue the first user
            if current_user not in visited:
                visited.add(current_user) # Mark the user as visited
                bfs_result.append(current_user) # Add to result

```

```

                bfs_result.append(current_user) # Add to result

                # Enqueue all unvisited friends of the current user
                for friend in self.adjacency_list[current_user]:
                    if friend not in visited:
                        queue.append(friend)

        return bfs_result

    def dfs_recursive(self, start_user, visited=None):
        if visited is None:
            visited = set() # To keep track of visited nodes
            visited.add(start_user) # Mark the user as visited
            dfs_result = [start_user] # Add to result

            # Recursively visit all unvisited friends of the current user
            for friend in self.adjacency_list[start_user]:
                if friend not in visited:
                    dfs_result.extend(self.dfs_recursive(friend, visited))

        return dfs_result

    def dfs_iterative(self, start_user):

```

```

62     def dfs_iterative(self, start_user):
63         visited = set() # To keep track of visited nodes
64         stack = [start_user] # Stack for DFS
65         dfs_result = []
66
67         while stack:
68             current_user = stack.pop() # Pop the last user from the stack
69             if current_user not in visited:
70                 visited.add(current_user) # Mark the user as visited
71                 dfs_result.append(current_user) # Add to result
72
73                 # Push all unvisited friends of the current user onto the stack
74                 for friend in reversed(self.adjacency_list[current_user]):
75                     if friend not in visited:
76                         stack.append(friend)
77
78         return dfs_result
79
80 # Sample test cases
81 if __name__ == "__main__":
82     graph = Graph()
83     graph.add_friendship("Alice", "Bob")
84     graph.add_friendship("Alice", "Charlie")
85     graph.add_friendship("Bob", "David")

```

```

f __name__ == "__main__":
    graph = Graph()
    graph.add_friendship("Alice", "Bob")
    graph.add_friendship("Alice", "Charlie")
    graph.add_friendship("Bob", "David")
    graph.add_friendship("Charlie", "Eve")
    graph.add_friendship("David", "Frank")

    print("BFS from Alice:", graph.bfs("Alice")) # Output: ['Alice', 'Bob', 'Charlie', 'David']
    print("DFS (recursive) from Alice:", graph.dfs_recursive("Alice")) # Output: ['Alice', 'Bob', 'David', 'Frank', 'Charlie', 'Eve']
    print("DFS (iterative) from Alice:", graph.dfs_iterative("Alice")) # Output: ['Alice', 'Bob', 'David', 'Frank', 'Charlie', 'Eve']

```

## OUTPUT:

```

PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game> & C:/Users/HARSHAVARDHAN/AppData/Local/Programs/Python/Python315/python.exe "c:/Users/HARSHAVARDHAN/Downloads/Snake Game/Snake Game/Assets/# Implement a graph to model social network.py"
BFS from Alice: ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Frank']
DFS (recursive) from Alice: ['Alice', 'Bob', 'David', 'Frank', 'Charlie', 'Eve']
DFS (iterative) from Alice: ['Alice', 'Bob', 'David', 'Frank', 'Charlie', 'Eve']
PS C:\Users\HARSHAVARDHAN\Downloads\Snake Game\Snake Game>

```

## Explanation:

The code defines a Graph class to model social network connections using an adjacency list. Each user is represented as a node, and friendships are represented as edges between nodes. The Graph class provides methods to add users, add friendships, and perform breadth-first search (BFS) and depth-first search (DFS) traversals.

