Name: Jhelo D. Palco

Section: IDB2

Activity.py × LinkedStack.py

add_first × Cc W .* 2/5

```
class LinkedStack:
    class _Node:
        __slots__ = '_element', '_next'

        def __init__(self, element, next):
            self._element = element
            self._next = next

    def __init__(self):
        self._head = None
        self._size = 0

    def __len__(self):
        return self._size

    def is_empty(self):
        return self._size == 0

    def push(self, e):
        self._head = self._Node(e, self._head)
        self._size += 1

    def top(self):
        if self.is_empty():
            raise Exception('Stack is empty')
        return self._head._element

    def pop(self):
        if self.is_empty():
            raise Exception("The stack is empty!")
        answer = self._head._element
        self._head = self._head._next
        self._size -= 1
        return answer

```

Activity.py × LinkedStack.py

add_first × Cc W .* 2/5

```
class PositionalList(LinkedStack):
    class Position:
        def __init__(self, container, node):
            self._container = container
            self._node = node

        def element(self):
            return self._node._element

        def __eq__(self, other):
            return type(other) is type(self) and other._node is self._node

        def __ne__(self, other):
            return not (self == other)

    def __init__(self):
        super().__init__()
        self._header = self._Node( element: None,  next: None)
        self._trailer = self._Node( element: None,  next: None)
        self._header._next = self._trailer
        self._size = 0

    def _validate(self, p):
        if not isinstance(p, self.Position):
            raise TypeError('p must be proper Position type')
        if p._container is not self:
            raise ValueError('p does not belong to this container')
        if p._node._next is None:
            raise ValueError('p is no longer valid')
        return p._node

    def _make_position(self, node):
        if node is self._header or node is self._trailer:
            return None
        return self.Position(self, node)

    def first(self):
        return self._make_position(self._header._next)
```

Activity.py × LinkedStack.py

add_first Cc W .* 2/5

```
76      def last(self):
77          cursor = self._header
78          while cursor._next != self._trailer:
79              cursor = cursor._next
80          return self._make_position(cursor)
81
82      def after(self, p):
83          node = self._validate(p)
84          if node._next is self._trailer:
85              return None
86          return self._make_position(node._next)
87
88      def __iter__(self):
89          cursor = self.first()
90          while cursor is not None:
91              yield cursor.element()
92              cursor = self.after(cursor)
93
94      def _insert_after(self, node, element):
95          newest = self._Node(element, node._next)
96          node._next = newest
97          self._size += 1
98          return newest
99
100     def add_first(self, e):
101         return self._make_position(self._insert_after(self._header, e))
102
103     def add_last(self, e):
104         cursor = self._header
105         while cursor._next != self._trailer:
106             cursor = cursor._next
107         return self._make_position(self._insert_after(cursor, e))
108
109
110 def evaluate_postfix(expression):
111     stack = LinkedStack()
112     tokens = expression.strip().split()
113
114     for token in tokens:
```

```python

def evaluate_postfix(expression):
    stack = LinkedStack()
    tokens = expression.strip().split()

    for token in tokens:
        if token in '+-*/':
            b = stack.pop()
            a = stack.pop()

            if token == '+':
                stack.push(a + b)
            elif token == '-':
                stack.push(a - b)
            elif token == '*':
                stack.push(a * b)
            elif token == '/':
                stack.push(a / b)
        else:
            stack.push(float(token))

    return stack.pop()


def sort_positional_list(numbers):
    asc_list = PositionalList()
    desc_list = PositionalList()

    for num in numbers:
        if asc_list.is_empty():
            asc_list.add_first(num)
            continue

        current = asc_list.first()
        inserted = False

        if num < current.element():
            asc_list.add_first(num)
            inserted = True
```

```
        while not inserted and current is not None:
            if current.element() <= num:
                next_pos = asc_list.after(current)
                if next_pos is None or next_pos.element() > num:
                    new_pos = asc_list.add_last(num)
                    inserted = True
                    break
            current = asc_list.after(current)

        if not inserted:
            asc_list.add_last(num)

    for num in numbers:
        if desc_list.is_empty():
            desc_list.add_first(num)
            continue

        current = desc_list.first()
        inserted = False

        if num > current.element():
            desc_list.add_first(num)
            inserted = True

        while not inserted and current is not None:
            if current.element() >= num:
                next_pos = desc_list.after(current)
                if next_pos is None or next_pos.element() < num:
                    new_pos = desc_list.add_last(num)
                    inserted = True
                    break
            current = desc_list.after(current)

        if not inserted:
            desc_list.add_last(num)

    return asc_list, desc_list
```

```
1 usage
def main():
    expr = "5 2 + 8 3 - * 4 /"
    result = evaluate_postfix(expr)
    print(f"Postfix expression: {expr}")
    print(f"Result: {result}")

    numbers = [1, 72, 81, 25, 65, 91, 11]
    print("\nOriginal numbers:", numbers)

    asc_list, desc_list = sort_positional_list(numbers)

    print("Ascending order:", end=" ")
    for num in asc_list:
        print(num, end=" ")

    print("\nDescending order:", end=" ")
    for num in desc_list:
        print(num, end=" ")
    print()


if __name__ == "__main__":
    main()
```