Molina, Joshua Ali S.

IDB2 DSALGO1

11/22/2024

#1

main.py × PositionalList.py LinkedStack.py

```python
# Molina, Joshua Ali S.
# IDB2 DSALGO1
# Finals Activity #2

from LinkedStack import LinkedStack as Stack, into_postfix
from PositionalList import PositionalList as PositionalList


#1 Infix to Postfix
S = Stack()
print("#1")
infix_expression = input("Enter an infix expression: ")

postfix_expression = into_postfix(infix_expression)

print(f"The postfix expression is: '{postfix_expression}'")

print()

#2 Sort positional list
print("#2")
P = PositionalList()
```

```
#1
Enter an infix expression: (5 + 2) * (8 - 3) / 4
The postfix expression is: '5 2 + 8 3 - * 4 /'
```

#2

```python
#2 Sort positional list
print("#2")
P = PositionalList()

numbers = [1, 72, 81, 25, 65, 91, 11]
for number in numbers:
    P.add_last(number)
```

```python
def insertion_sort(L):
    """Sort the PositionalList in ascending order using insertion sort."""
    if len(L) > 1:  # Otherwise, no need to sort
        marker = L.first()
        while marker != L.last():
            pivot = L.after(marker)  # Next item to place
            value = pivot.element()
            if value >= marker.element():  # Pivot is already sorted
                marker = pivot  # Pivot becomes the new marker
            else:  # Must relocate pivot
                walk = marker  # Find the leftmost value greater than pivot
                while walk != L.first() and L.before(walk).element() > value:
                    walk = L.before(walk)
                L.delete(pivot)  # Remove pivot
                L.add_before(walk, value)  # Reinsert pivot

# Define the insertion sort function (descending order)
1 usage
def insertion_sort_descending(L):
    """Sort the PositionalList in descending order using insertion sort."""
    if len(L) > 1:  # Otherwise, no need to sort
        marker = L.first()
        while marker != L.last():
            pivot = L.after(marker)  # Next item to place
            value = pivot.element()
            if value <= marker.element():  # Pivot is already sorted
                marker = pivot  # Pivot becomes the new marker
            else:  # Must relocate pivot
                walk = marker  # Find the leftmost value smaller than pivot
                while walk != L.first() and L.before(walk).element() < value:
                    walk = L.before(walk)
                L.delete(pivot)  # Remove pivot
                L.add_before(walk, value)  # Reinsert pivot
```

```python
def printList():
    for x in P:
        print(x, end=" ")

# Initial List
print("Initial List:")
printList()
print()

# Ascending order
insertion_sort(P)
print("Ascending order:")
printList()
print()

# Descending order
insertion_sort_descending(P)
print("Descending order:")
printList()
print()
```

```
#2
Initial List:
1 72 81 25 65 91 11
Ascending order:
1 11 25 65 72 81 91
Descending order:
91 81 72 65 25 11 1
```

Positional List code:

```
from DoublyLinkedBase import _DoublyLinkedBase
class PositionalList(_DoublyLinkedBase):
  '''A sequential container of elements allowing positional access.'''
  #---Positional list class
  class Position:
    '''An abstraction representing the location of a single element.'''
    def __init__(self, container, node):
      '''Constructor should not be invoked by the user.'''
      self._container = container
      self._node = node
    def element(self):
      '''Return the element stored at this Position'''
      return self._node._element
    def __eq__(self, other):
      '''Return True if other is a Position representing the same location.'''
      return type(other) is type(self) and other._node is self._node
    def __ne__(self,other):
      '''Return True if other does not represent the same location.'''
      return not  (self == other) #opposite of __eq__

  #-- utility method
  def _validate(self, p):
    '''Return postiion's node or raise appropriate error if invalid'''
    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:#convention for deprecated nodes
      raise ValueError('p is no longer valid')
    return p._node
  #-- utility method
  def _make_position(self, node):
    '''Return Position instance for given node (or None if sentinel).'''
    if node is self._header or node is self._trailer:
      return None #boudnary violation
    else:
      return self.Position(self, node) #legitimate position
  #-- accessors
  def first(self):
    '''Return the first Position in the list (or None if list is empty.)'''
    return self._make_position(self._header._next)
  def last(self):
    '''Return the last Position in the list (or None if list is empty)'''
    return self._make_position(self._trailer._prev)
  def before(self, p):
    '''Return the Position just before Position P (or None if p is first)'''
    node = self._validate(p)
    return self._make_position(node._prev)
  def after(self, p):
    '''Return the Position just after Position p (or None if p is last.)'''
    node = self._validate(p)
    return self._make_position(node._next)
  def __iter__(self):
    '''Generate forward iteration of the elements of the list'''
```

```
    cursor = self.first()
    while cursor is not None:
      yield cursor.element()
      cursor = self.after(cursor)
  #--mutators
  #override inherited version to return Position, rather than Node
  def _insert_between(self, e, predecessor, successor):
    '''Add element between existing nodes and return new Position'''
    node = super()._insert_between(e, predecessor, successor)
    return self._make_position(node)
  def add_first(self, e):
    '''Insert element e at the front of the lsit and return new Position.'''
    return self._insert_between(e, self._header, self._header._next)
  def add_last(self, e):
    '''Insert element e at the back of the list and return new Position.'''
    return self._insert_between(e, self._trailer._prev, self._trailer)
  def add_before(self, p, e):
    '''Insert element e into list before Position p and return new Position'''
    original = self._validate(p)
    return self._insert_between(e, original._prev, original)
  def add_after(self, p, e):
    '''Insert element e into list after Position p and return new Position'''
    original = self._validate(p)
    return self._insert_between(e, original, original._next)
  def delete(self, p):
    '''Remove and return the element at Position p.'''
    original = self._validate(p)
    return self._delete_node(original)#inherited method returns element
  def replace(self, p, e):
    '''Replace the element at Position p with e.'''
    '''Return the element formerly at Position P.'''
    original = self._validate(p)
    old_value = original._element#temporarily store old element
    original._element = e #replace with new element
    return old_value #return the old element value
```

Linked Stack code:

```
class LinkedStack:
  """LIFO Stack implementation using a singly linked list for storage."""

  class _Node:
    """Lightweight non-public class for storing a singly linked node."""
    __slots__ = '_element', '_next'  # Streamline memory usage

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

  def __init__(self):
    """Create an empty stack."""
    self._head = None
    self._size = 0

  def __len__(self):
```

```python
        """Return the number of elements in the stack."""
        return self._size

    def is_empty(self):
        """Return True if the stack is empty."""
        return self._size == 0

    def push(self, element):
        """Add element to the top of the stack."""
        new_node = self._Node(element, self._head)
        self._head = new_node
        self._size += 1

    def top(self):
        """Return but do not remove the element at the top of the stack."""
        if self.is_empty():
            raise Exception('Stack is empty')
        return self._head._element

    def pop(self):
        """Remove and return the element from the top of the stack (LIFO)."""
        if self.is_empty():
            raise Exception("The stack is empty!")
        top_element = self._head._element
        self._head = self._head._next
        self._size -= 1
        return top_element

    @staticmethod
    def evaluate_postfix(expression):
        """Evaluate a postfix expression using a linked stack."""
        stack = LinkedStack()
        tokens = expression.split()

        for token in tokens:
            if token.isdigit():
                stack.push(int(token))
            else:
                operand2 = stack.pop()
                operand1 = stack.pop()
                result = LinkedStack.perform_operation(operand1, operand2, token)
                stack.push(result)

        return stack.pop()

    @staticmethod
    def perform_operation(operand1, operand2, operator):
        """Perform arithmetic operations based on the operator."""
        if operator == '+':
            return operand1 + operand2
        elif operator == '-':
            return operand1 - operand2
        elif operator == '*':
            return operand1 * operand2
        elif operator == '/':
            if operand2 == 0:
```

```python
            raise ZeroDivisionError("Division by zero!")
        return operand1 / operand2  # Use float division for accuracy
    else:
        raise ValueError(f"Unknown operator: {operator}")


def precedence(operator):
    """Return precedence of operators."""
    if operator in ('+', '-'):
        return 1
    if operator in ('*', '/'):
        return 2
    return 0


def into_postfix(expression):
    """Convert an infix expression to postfix notation."""
    output = []
    operators = LinkedStack()

    # Remove spaces and tokenize input expression
    tokens = expression.replace(" ", "")

    current_number = ''

    for char in tokens:
        if char.isdigit():
            current_number += char  # Build multi-digit numbers
        else:
            if current_number:
                output.append(current_number)
                current_number = ''
            if char in '+-*/':  # Ensure using standard operators only
                while (not operators.is_empty() and
                       precedence(operators.top()) >= precedence(char)):
                    output.append(operators.pop())
                operators.push(char)
            elif char == '(':
                operators.push(char)
            elif char == ')':
                while not operators.is_empty() and operators.top() != '(':
                    output.append(operators.pop())
                operators.pop()

    if current_number:
        output.append(current_number)

    while not operators.is_empty():
        output.append(operators.pop())

    return ' '.join(output)
```