Assignment 6: Medians and Order Statistics and Elementary Data Structures

Pavani Chavali

Department of Computer Science, University of The Cumberlands

MSCS-532-B01: Algorithms and Data Structures

Professor. Brandon Bass

Date: 11/23/2025

Part2: Elementary data structures Implementation and Discussion

Implementation of Arrays and Matrices

```
class dynamicarrays:
    def __init__(self):
        self.elements = []
        self.length = len(self.elements)

    def insert(self,index,data):
        self.elements.insert(index,data)
        self.length = self.length +1
        print(f"appended value:{data}")

    def delete(self,index):
        if index<len(self.elements):
            deletedvalue = self.elements.pop(index)
            self.length = self.length-1
            print(f"deleted value:{deletedvalue}")


    def traverse(self):
        print(self.elements)

array = dynamicarrays()
array.insert(0,7)
array.insert(1,52)
array.insert(2,27)
array.insert(3,34)
array.insert(4,100)

array.traverse()

array.insert(2,200)

array.traverse()

array.delete(4)

array.traverse()
```

Output:

```
● pavanichavali@Pavanis-MacBook-Pro practice % /usr/local/bin/python3 /Users/pavanichavali/Desktop/practice/practice/arrays_implementation.py
  appended value:7
  appended value:52
  appended value:27
  appended value:34
  appended value:100
  [7, 52, 27, 34, 100]
  appended value:200
  [7, 52, 200, 27, 34, 100]
  deleted value:34
  [7, 52, 200, 27, 100]
○ pavanichavali@Pavanis-MacBook-Pro practice % 
```

Implementation of matrices:

```
class Matrix:
    def __init__(self, elements):
        self.matrix = elements
        self.rows = len(elements)
        self.colmns = len(elements[0])
    def traverse(self):
        if self.rows == 0 or self.colmns == 0:
            print("Matrix is empty")
            return
        for row in self.matrix:
            print(row)

    def insertrow(self, index, rows):
        if self.colmns > 0 and len(rows) != self.colmns:
            return

        if index <= self.rows:
            self.matrix.insert(index, rows)
            self.rows = self.rows + 1
            if self.colmns == 0 and len(rows) > 0:
                self.colmns = len(rows)
            print(f"Row inserted at index {index}.")


    def insertclmn(self, index, clmns):
        if len(clmns) != self.rows:
            print(f"Error: New column must have {self.rows} rows.")
            return

        if index <= self.colmns:
            for i in range(self.rows):
                self.matrix[i].insert(index, clmns[i])
            self.colmns = self.colmns + 1
            print(f"Column inserted at index {index}.")
```

```
    def deleterow(self, index):
        if index < self.rows:
            delrow = self.matrix.pop(index)
            self.rows = self.rows - 1
            print(f"Row deleted at index {index}.")

            if self.rows == 0:
                self.colmns = 0
            return delrow


    def delclmn(self, index):
        if index < self.colmns:
            for i in range(self.rows):
                self.matrix[i].pop(index)
            self.colmns = self.colmns - 1
            print(f"Column deleted at index {index}.")




matrix = [
    [5, 6, 7],
    [8, 9, 10]
]

M = Matrix(matrix)
M.traverse()


M.insertrow(2, [234, 235, 236])

M.traverse()

M.insertclmn(1, [10, 20, 30])

M.traverse()
```

Output:

```
pavanichavali@Pavanis-MacBook-Pro practice % /usr/local/b
in/python3 /Users/pavanichavali/Desktop/practice/practice
/Matrix_Implmenetation.py
[5, 6, 7]
[8, 9, 10]
Row inserted at index 2.
[5, 6, 7]
[8, 9, 10]
[234, 235, 236]
Column inserted at index 1.
[5, 10, 6, 7]
[8, 20, 9, 10]
[234, 30, 235, 236]
pavanichavali@Pavanis-MacBook-Pro practice %
```

Implementation of Stacks:

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

    def push(self,data):
        self.elements.append(data)
        print(f"appended :{data}")

    def pop(self):
        if len(self.elements)==0:
            print("stack is empty")
            return None
        item = self.elements.pop()
        return item

    def peek(self):
        if len(self.elements)==0:
            print("stack is empty")
            return None
        return self.elements[-1]

    def traverse(self):
        if len(self.elements)==0:
            print("stack is empty")
            return None

        for i in reversed(self.elements):
            print(i)
```

```
ss = Stack()
ss.push(7)
ss.push(52)
ss.push(27)
ss.push(34)
ss.push(100)

ss.traverse()

ss.pop()

ss.traverse()

top =ss.peek()
print(top)
```

Output:

```
pavanichavali@Pavanis-MacBook-Pro practice % /usr/local/bin/python3 /Users/pavanichavali/Desktop/practice/practice/stack_implementation.py
appended :7
appended :52
appended :27
appended :34
appended :100
100
34
27
52
7
34
27
52
7
34
pavanichavali@Pavanis-MacBook-Pro practice %
```

Implementation of Queues:

```
class Queue:
    def __init__(self):
        self.elements =[]

    def enqueue(self, data):
        self.elements.append(data)
```

```
    def dequeue(self):
        if len(self.elements)==0:
            print("queue is empty")

        lastelement = self.elements.pop()
        print(f"popped element:{lastelement}")

    def traverse(self):
        print(self.elements)


q = Queue()

q.enqueue(7)
q.enqueue(52)
q.enqueue(27)
q.enqueue(34)
q.enqueue(100)

q.traverse()

q.dequeue()

q.traverse()
```

Output:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   PVS-STUDIO                Python
pavanichavali@Pavanis-MacBook-Pro practice % /usr/local/bin/python3 /Users/pavanichavali/Desktop/practice/practice/queue_implementation.py
[7, 52, 27, 34, 100]
popped element:100
[7, 52, 27, 34]
pavanichavali@Pavanis-MacBook-Pro practice %
```

Implementation of LinkedList using Arrays:

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

```
    self.head = None

def insertionatstart(self,data):
    newnode = Node(data)
    newnode.next = self.head
    self.head = newnode

def insertionatlast(self,data):
    newnode = Node(data)
    if self.head is None:
        return
    curr = self.head
    while curr.next:
        curr = curr.next
    curr.next = newnode

def deletenode(self,value):
    curr = self.head
    prev = None

    if curr is not None and curr.data == value:
        curr = None
        print(f"deleted node with value: {value}")
        return
    while curr is not None and curr.data!=value:
        prev = curr
        curr = curr.next
    if curr is None:
        print("node is not found")

    prev.next = curr.next
    curr = None
    print("deleted node {value}")

def traverse(self):
    if self.head == None:
        print("list is empty")
        return

    curr = self.head
    while curr.next:
        print(f"{curr.data}->",end="")
        curr= curr.next
    print(curr.data)
```

```
LL = Linkedlist()
LL.insertionatstart(52)
LL.insertionatlast(34)
LL.insertionatlast(27)
LL.insertionatstart(7)
LL.insertionatlast(100)

LL.traverse()

LL.deletenode(34)

LL.traverse()
```

Ouput:

```
● pavanichavali@Pavanis-MacBook-Pro practice % /usr/local/bin/python3 /Users/pavanichavali/Desktop/practice/practice/Linkedlist_implementation
  .py
  7->52->34->27->100
  deleted node {value}
  7->52->27->100
○ pavanichavali@Pavanis-MacBook-Pro practice % 
```

Array:

1. Array (I implemented Dynamic Array using Python List)

Arrays/List elements are stored in contiguous memory, which makes indexing very fast but insertion/deletion slow.

- **Traversal: O(1)** :Accessing the element at any index i is instant because the memory address can be calculated directly.
- **Insertion/Deletion(Start):** O(n) :To insert or delete an element in the middle or at the start, all subsequent n-i elements must be shifted in memory to maintain contiguity.

- **Insertion/Deletion (End):** O(1): Appending or removing the last element is fast because no shifting is required. The complexity is amortized O(1) due to occasional O(n) resizing operations when the underlying memory block runs out of space.

LinkedList:

Linked Lists are dynamic collections of nodes connected by pointers, allowing flexibility in size.

- **Traversal/Access: O(n)** To find the element at index i or the node containing a specific value, we must start at the Head and follow the pointers sequentially.
- **Insertion/Deletion (Head): O(1)** This is the best case. we only need to update the Head pointer and the next pointer of the new node.
- **Insertion/Deletion (Middle/End): O(1)** While the actual link update takes O(1),we first have to traverse the list to find the preceding node, which takes O(n) time.

Stack (which follows LIFO: Last-In, First-Out)

A Stack is typically implemented using an array/list where all operations occur only at the Top.

- **Traversal: O(n)** The traversal should be from top to bottom or the other way around to see all elements.
- **Insertion (Push): O(1)** Adding an element to the Top is equivalent to an array append, which is constant time.

- **Deletion (Pop): O(1):** Removing the element from the Top is equivalent to array popping from the end, which is constant time.

Queue (which follows FIFO: First-In, First-Out)

A Queue is implemented using a structure that allows operations at both ends Front and Rear. I implemented using Python deque.

- **Traversal**(n) Like a stack or list, the traversal will be through the data structure that is created.
- **Insertion (Enqueue): O(1)** Adding to the end of the Queue is a constant time operation.
- **Deletion (Dequeue): O(1)** Removing from the start is a constant time operation when using an efficient structure like a deque.

Githublink: https://github.com/PavaniChavali135/AlgoandDatastructures/tree/main/part2