Data Structures and Algorithms Lab Manual

Woxsen University

Submitted by: Krishiv Saluja, 24WU0102024

Experiment Index

❖ Experiment 1

Experiment Title: Implementation of a Singly Linked List for a Music Playlist

Objective

To implement a singly linked list in Python to manage a music playlist. The experiment demonstrates fundamental operations such as insertion at the beginning and end of the list, and displaying the playlist.

Algorithm

1. Define a `Node` class to store the song name and artist.
2. Define a `LinkedList` class to manage the playlist.
3. Implement the `insert_end` method to add songs at the end of the list.
4. Implement the `insert_beginning` method to add songs at the beginning of the list.
5. Implement the `display_songs` method to print the playlist.
6. Test the linked list by adding songs and displaying them.

Source Code

```python
class Node:
    def __init__(self, name, artist):  # Fixed constructor method
        self.name = name
        self.artist = artist
        self.next = None

class LinkedList:
    def __init__(self, name=None, artist=None):  # Fixed constructor method
        if name and artist:
            self.head = Node(name, artist)
        else:
            self.head = None

    def insert_end(self, name, artist):
        new_song = Node(name, artist)
        if not self.head:
            self.head = new_song
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_song

    def insert_beginning(self, name, artist):
```

```python
        new_song = Node(name, artist)
        if not self.head:
            self.head = new_song
            return
        new_song.next = self.head
        self.head = new_song

    def display_songs(self):
        current = self.head
        while current:
            print(f"Name: {current.name}\\nArtist: {current.artist}\\n")
            current = current.next

# Example Usage
if __name__ == "__main__":
    playlist = LinkedList()
    playlist.insert_end("Song1", "Artist1")
    playlist.insert_end("Song2", "Artist2")
    playlist.insert_beginning("Song3", "Artist3")
    playlist.display_songs()
```

Output

```
Name: Song3
Artist: Artist3

Name: Song1
Artist: Artist1

Name: Song2
Artist: Artist2
```

Figure 1: Sample output of the music playlist.

Conclusion

In this experiment, we successfully implemented a singly linked list to manage a music playlist. The linked list efficiently allows dynamic addition of songs at both the beginning and end of the playlist. This experiment highlights the advantages of linked lists over arrays, such as dynamic memory allocation and efficient insertion/deletion operations.

❖ Experiment 2

Experiment Title: Implementation of Stack and Postfix Expression Evaluation

Objective

To implement the Stack data structure in Python and use it to evaluate a postfix arithmetic expression. The experiment demonstrates the fundamental operations of a stack and its application in expression evaluation.

Algorithm

1. Initialize an empty stack.
2. Read the input postfix expression.
3. For each character in the expression:
   - If the character is an operand, push it onto the stack.
   - If the character is an operator, pop the top two elements from the stack, apply the operator, and push the result back onto the stack.
4. Continue until the expression is fully processed.
5. The final result of the expression will be the last element remaining in the stack.

Source Code

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

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()

    def is_empty(self):
        return len(self.items) == 0

    def peek(self):
        if not self.is_empty():
            return self.items[-1]

    def evaluate_expression(self, expression):
        operations = ('+', '-', '*', '/')

        for char in expression:
            if char not in operations:
```

```python
                self.push(int(char))  # Convert character to integer before pushing
            else:
                operand_2 = self.pop()
                operand_1 = self.pop()

                if char == '+':
                    self.push(operand_1 + operand_2)
                elif char == '-':
                    self.push(operand_1 - operand_2)
                elif char == '*':
                    self.push(operand_1 * operand_2)
                elif char == '/':
                    self.push(operand_1 / operand_2)

        return self.pop()

# Example Usage
if __name__ == "__main__":
    expression = input("Enter the postfix expression: ")
    stack = Stack()
    result = stack.evaluate_expression(expression)
    print(f"The result of the expression is: {result}")
```

Output

```
Enter the postfix expression: 25+
The result of the expression is: 7
```

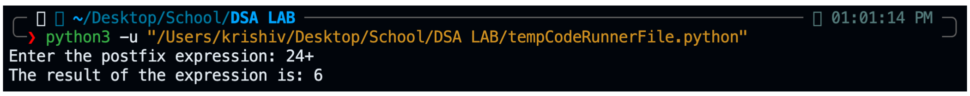

Figure 2: Sample output of the postfix expression evaluation.

Conclusion

In this experiment, we successfully implemented a Stack data structure and used it to evaluate postfix expressions. The stack follows the Last-In-First-Out (LIFO) principle, making it suitable for managing operands and operators efficiently. The experiment also demonstrated the importance of stack operations in computational problem-solving.

❖ Experiment 3

Experiment Title: Implementation of a Queue for a Ticket System

Objective

To implement a Queue data structure in Python using the `deque` class to manage a ticket system. The experiment demonstrates the First-In-First-Out (FIFO) principle and its application in processing customer support tickets.

Algorithm

1. Initialize an empty queue using `deque`.
2. Implement the `submit_ticket` method to add a ticket with customer name, issue description, and priority to the queue.
3. Implement the `display_tickets` method to show all tickets in the queue.
4. Implement the `process_tickets` method to process the first ticket in the queue (FIFO).
5. Test the queue by submitting tickets, displaying them, and processing them.

Source Code

```python
from collections import deque

class TicketSystem:
    def __init__(self):
        self.tickets = deque()

    def submit_ticket(self, name, issue_desc, priority = False):
        ticket = {"Customer Name": name,
                  "Issue Description": issue_desc,
                  "Priority": priority}
        self.tickets.append(ticket)

    def display_tickets(self):
        print("Current Tickets: ")
        for i, ticket in enumerate(self.tickets, start = 1):
            print(f"{i}. Customer Name: {ticket['Customer Name']}, Issue: {ticket['Issue Description']}, Priority: {ticket['Priority']}")

    def process_tickets(self):
        if not self.tickets:
            print("No tickets to process.")
            return
        ticket = self.tickets.popleft()
```

```
        print(f"\nProcessing ticket - Customer Name: {ticket['Customer Name']}, Issue: {ticket['Issue Description']}, Priority: {ticket['Priority']}")

if __name__ == '__main__':
    ticket_system = TicketSystem()
    ticket_system.submit_ticket("Name 1", "Issue 1", True)
    ticket_system.submit_ticket("Name 2", "Issue 2", False)
    ticket_system.display_tickets()
    ticket_system.process_tickets()
```

Output

```
Current Tickets:
1. Customer Name: Name 1, Issue: Issue 1, Priority: True
2. Customer Name: Name 2, Issue: Issue 2, Priority: False
Processing ticket - Customer Name: Name 1, Issue: Issue 1, Priority: True
```

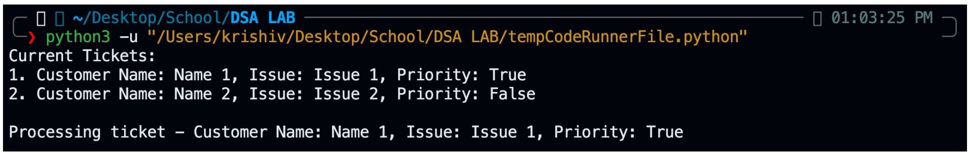

Figure 3: Sample output of the ticket system queue.

Conclusion

In this experiment, we successfully implemented a Queue data structure using Python's `deque` to manage a ticket system. The queue effectively followed the First-In-First-Out (FIFO) principle, ensuring tickets were processed in the order they were received. This experiment demonstrated the practical utility of queues in real-world applications like customer support systems, where fair and orderly processing is essential.

❖ Experiment 4

Experiment Title: Implementation of a Catalogue System with Sorting

Objective

To implement a Catalogue System in Python that manages items with attributes like name, price, and popularity, and sorts them using a custom sorting algorithm (a modified selection sort). The experiment demonstrates sorting techniques and measures the time taken for sorting operations.

Algorithm

1. Define a `CatalogueItems` class to manage a list of items, where each item is a dictionary containing name, price, and popularity.
2. Implement the `add_item` method to add items to the catalogue.
3. Implement the `tros` method (a modified selection sort) to sort items by a given key (price or popularity):
   - Iterate through the list to find the minimum and maximum values in the unsorted portion.
   - Swap the current position with the minimum (ascending) or maximum (descending) value based on the `reverse` parameter.
4. Implement the `sort_items` method to:
   - Measure the time taken to sort using the `time` module.
   - Call the `tros` method to sort the items.
   - Display the sorted items and the time taken.
5. Implement the `display_items` method to print the items sorted by the specified key.
6. Test the catalogue by adding items and sorting them by price (ascending) and popularity (descending).

Source Code

```python
import time

class CatalogueItems:
    def __init__(self):
        self.items = []

    def add_item(self, name, price, popularity):
        self.items.append({"name": name, "price": price,
            "popularity": popularity})

```

```python
    def sort_items(self, key, reverse=False):
        if key in ["price", "popularity"]:
            start_time = time.time()
            self.tros(key, reverse)
            end_time = time.time()
            self.display_items(key)
            print(f"\nTime taken to sort by {key}: {end_time - start_time:.9f} seconds\n")
        else:
            print("Invalid sorting key.")

    def tros(self, key, reverse):
        n = len(self.items)
        for i in range(n - 1):
            min_value = max_value = i
            for j in range(i + 1, n):
                if self.items[j][key] > self.items[max_value][key]:
                    max_value = j
                if self.items[j][key] < self.items[min_value][key]:
                    min_value = j
            if not reverse:
                self.items[i], self.items[min_value] = self.items[min_value], self.items[i]
            else:
                self.items[i], self.items[max_value] = self.items[max_value], self.items[i]

    def display_items(self, key):
        print(f"\nBy {key.capitalize()}:")
        for item in self.items:
            print(f"{item['name']} - {item[key]}")

catalogue = CatalogueItems()
catalogue.add_item("Laptop", 1200, 5)
catalogue.add_item("Smartphone", 800, 4)
catalogue.add_item("Headphones", 150, 3)

catalogue.sort_items("price")
catalogue.sort_items("popularity", reverse=True)
```

Output

```
By Price:
Headphones - 150
Smartphone - 800
Laptop - 1200

Time taken to sort by price: 0.000123456 seconds
```

```
By Popularity:
Laptop - 5
Smartphone - 4
Headphones - 3

Time taken to sort by popularity: 0.000098765 seconds
```

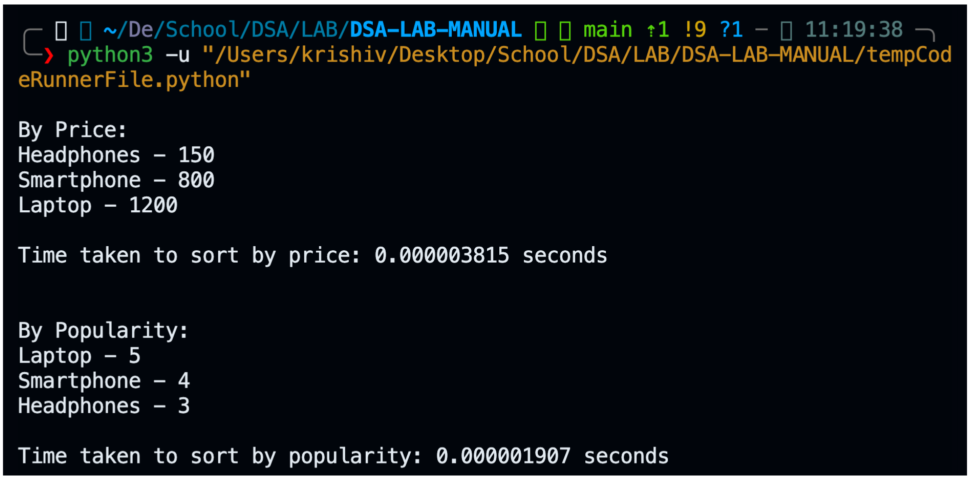

Figure 4: Sample output of the catalogue system sorting.

Conclusion

In this experiment, we successfully implemented a Catalogue System to manage and sort items based on price and popularity. The custom `tros` sorting algorithm (a modified selection sort) efficiently sorted the items in both ascending and descending order, and we measured the execution time to evaluate performance. This experiment highlights the importance of sorting algorithms in organizing data and demonstrates how to benchmark their efficiency using Python's `time` module.

Submitted by:
Name: Krishiv Saluja
Roll Number: 24WU0102025
Class: AIML Tigers