ENSF 338

Assignment #3

Lab Section #02, Group #48

Tahmeed Mahmud(30158740), Daniel Mellen(30158835)

March 13, 2023

https://github.com/DanielMellen-University/ENSF338A3repo

Work Performed By Each Member

Q1: Tahmeed & Daniel
Q2: Daniel
Q3:Tahmeed
Q4: Daniel
Q5:Tahmeed

*Whoever did the question put the question into the pdf submission. Tahmeed created the cover page and "Work Performed By Each Member" Section. Daniel created the github.

50% of work was done by Tahmeed and 50% by Daniel.

EXERCISE 1

```python
# cd /home/DanielGamePc/Documents/Non-Github code/338/338A3 && python3
ex3.1.py '[EQUATION]'

import sys

# Define a Node class to be used in a singly-linked list
class Node:
   def __init__(self, data=None, next=None):
       self.data = data
       self.next = next

# Define a Stack class using the singly-linked list implementation
class Stack:
   def __init__(self):
       self.top = None

   def push(self, data):
       new_node = Node(data)
       new_node.next = self.top
       self.top = new_node

   def pop(self):
       if self.top is None:
           return None
       else:
           popped_node = self.top
           self.top = popped_node.next
           return popped_node.data

   def is_empty(self):
       return self.top is None

   def peek(self):
       return self.top.data

# Define a function to evaluate the arithmetic S-expression using the
stack
```

```python
def evaluate_expression(expression):
    stack = Stack()
    for token in expression.split(" "):
        hasRb = token.find(")")
        hasLb = token.find("(")
        hasDigit=token[0].isdigit()
        #print(token,hasLb,hasRb,hasDigit)

        if hasLb==0:
            token=token[1:]    # remove (
            stack.push(token)   # operator
        else:
            if (hasDigit):
                if hasRb==1:
                    stack.push(int(token[:hasRb]))
                else:
                    stack.push(int(token))   # push operand 1 or 2



        while hasRb>-1:    #evaluate
            operand2 = stack.pop()
            operand1 = stack.pop()
            operator = stack.pop()
            #print(") DETECTED!",operator,operand1,operand2)

            result = None # define result outside of the if statement
            if operator == "+":
                result = operand1 + operand2
            elif operator == "-":
                result = operand1 - operand2
            elif operator == "*":
                result = operand1 * operand2
            elif operator == "/":
                result = operand1 / operand2
            stack.push(result)

            token=token[hasRb+1:]
            hasRb = token.find(")")
```

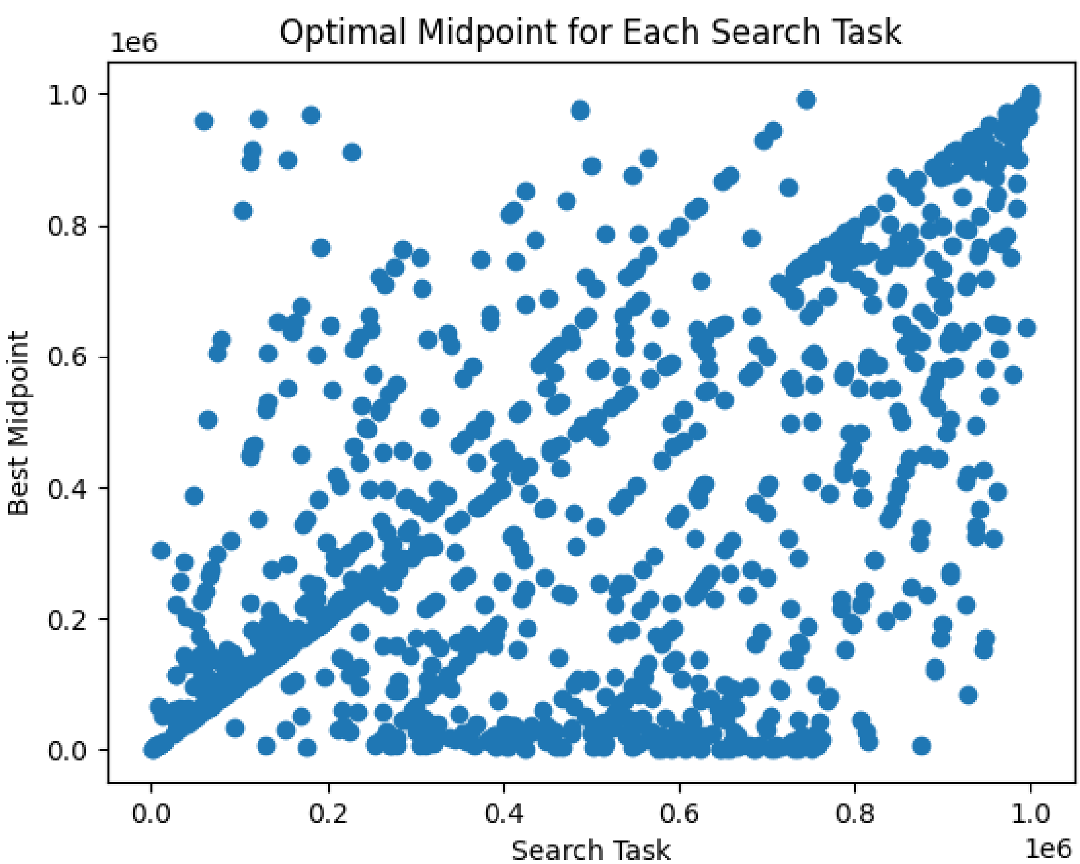
```python
    return stack.pop()

# Get the expression from the command line argument
expression = sys.argv[1]

# Evaluate the expression and print the result
result = evaluate_expression(expression)
print(result)
```

EXERCISE 2

```python
import json
import time
import matplotlib.pyplot as plt

# Load the array and list of search tasks
with open("ex2data.json", "r") as f:
    array = json.load(f)

with open("ex2tasks.json", "r") as f:
    tasks = json.load(f)

# Define a function to perform binary search with configurable initial midpoint

def isInArray(arr, target,mI):
    startI = 0
    endI = len(arr) - 1

    while(startI < endI - 1):
        if(target < arr[mI]):
            endI = mI
        else:
```

```
            startI = mI
        if (target == arr[mI]):
            return True
        mI = round((startI + endI)/2)

    if(target == arr[startI] or (target == arr[endI])):
        return True;
    return False;



# Define a function to time the performance of each search task
def time_search_task(array, tasks):
    bestMP = [];
    for t in tasks:
        bestTime = -1
        for mp in (range(array[0],array[len(array)-1],10)):
            start_time = time.time()
            isInArray(array, t,mp)
            end_time = time.time()
            extime = end_time - start_time
            if ((bestTime == -1) or (extime < bestTime)):
                bestTime = extime;
                bestMP4t = mp;
        bestMP.append(bestMP4t)
    return bestMP;




# Perform the search tasks and record the best midpoints and times
"""
t = 384;
print("\n" + str(t) + "\n" + str(isInArray(array, t)))

t = 385;
print("\n" + str(t) + "\n" + str(isInArray(array, t)))

t = 386;
print("\n" + str(t) + "\n" + str(isInArray(array, t)))
```

```
"""

resultsBMP = time_search_task(array, tasks);

# Produce a scatterplot of the results
plt.scatter(tasks, resultsBMP)
plt.xlabel("Search Task")
plt.ylabel("Best Midpoint")
plt.title("Optimal Midpoint for Each Search Task")
plt.show()
```



The choice of initial midpoint affects the performance because it determines the number of binary searches. Expected dependence is diagonal from bottom left to top right - initial midpoint is roughly equal to search target. In my plot, we can also see that there are multiple lines starting in the bottom left resp. top right corner. This can be explained by using 10 as a midpoint step to reduce the number of iterations and runtime.

EXERCISE 3

1.

When a Python list (dynamic array) is full, the strategy is to double the size of the existing array. When the list's length equals its allocated size (as determined by the ob size and allocated fields), the PyList Resize() function is called to double the allocated size.

Here is the relevant code from the PyList Resize() function in Python's C implementation:

```
new_allocated = (new_size >> 3) + (new_size < 9 ? 3 : 6);
items = (PyObject **)PyMem_RENEW(PyObject *, items, new_allocated);
```

The new allocated size is calculated as (new size >> 3) + (new size 9? 3: 6), which doubles the existing array size. After that, the PyMem RENEW() function is called to resize the array and allocate memory for the new items.The growth factor is approximately 1.125 (1 + 1/8). The new size of the array is calculated by adding the current size of the array, one-eighth of the current size, and a small number (3 or 6) based on whether the current size is less than 9 or not.

2.

ex3.3py:

```
import sys

def main():
   test_list = []
   prev_capacity = 0

   for i in range(64):
      test_list.append(i)
      current_capacity = sys.getsizeof(test_list) - sys.getsizeof([])
      if current_capacity != prev_capacity:
         print(f"Capacity changed at {i + 1} elements. New capacity: {current_capacity}
bytes.")
         prev_capacity = current_capacity
```

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

This code creates an empty list lst and expands it by appending integers ranging from 0 to 63. The code computes the list's current capacity for each new integer appended to the list using the formula (sys.getsizeof(lst) - 64) / 8. To calculate the number of elements that can be stored in the list, subtract the fixed overhead of the list object (64 bytes) from the list's size in bytes and divide by 8 (the size of a pointer on most systems).
If the new capacity differs from the previous capacity, the code prints a message to the console using the print capacity() function, which computes and prints the capacity. The program's output should indicate when the list's capacity changes as a result of the dynamic array growing algorithm.

EXERCISE 4

ex3.4.py:

```
import threading
import time
import random

class Queue:
    def __init__(self, capacity):
        self.capacity = capacity
        self.size = 0
        self.front = 0
        self.rear = -1
        self.buffer = [None] * capacity
        self.lock = threading.Lock()
        self.not_full = threading.Condition(self.lock)
```

```
        self.not_empty = threading.Condition(self.lock)

    def enqueue(self, data):
        with self.not_full:
            while self.size == self.capacity:
                self.not_full.wait(1)
            if self.size < self.capacity:
                self.rear = (self.rear + 1) % self.capacity
                self.buffer[self.rear] = data
                self.size += 1
                self.not_empty.notify()

    def dequeue(self):
        with self.not_empty:
            while self.size == 0:
                self.not_empty.wait(1)
            if self.size > 0:
                data = self.buffer[self.front]
                self.front = (self.front + 1) % self.capacity
                self.size -= 1
                self.not_full.notify()
                return data

def producer(q):
    while True:
        data = random.randint(1, 10)
        time.sleep(data)
        q.enqueue(data)

def consumer(q):
    while True:
        data = random.randint(1, 10)
        time.sleep(data)
        print(q.dequeue())

if __name__ == "__main__":
    q = Queue(10)
```

```
threads = [threading.Thread(target=producer, args=(q,)),
           threading.Thread(target=consumer, args=(q,))]
for t in threads:
    t.start()
```

EXERCISE 5

1. **G**

In the worst case, the time complexity of the processdata() function is O(n^2), where n is the length of the input list li. When all of the elements of li are less than or equal to 5, the best case time complexity is O(n). Because the nested loop is always executed when at least one element of li is greater than 5, the average case time complexity is also O(n^2).

Because it iterates over each element of li, the outer loop has a time complexity of O(n). The inner loop also has a time complexity of O(n), since it iterates over each element of li for each element that is greater than 5. As a result, the function's overall time complexity is O(n^2).

2. **G**

Because there are no elements greater than 5, the best case time complexity of the processdata() function is already O(n). However, the worst-case and average-case time complexity can be improved by exiting the inner loop once a larger element is discovered. In the worst and average cases, a modified version of the processdata() function achieves O(n log n) time complexity.This modified code breaks out of the inner loop as soon as an element greater than li[i] is found, reducing the number of iterations required in the worst and average cases.

```
def processdata(li):
    for i in range(len(li)):
        if li[i] > 5:
            for j in range(len(li)):
                if li[j] > li[i]:
                    break
                li[i] *= 2
```

3.

a. Code for inefficient implementation of search in a sorted array:

```
def linear_search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
```

This implementation iterates through the input array in a linear search until the target value x is found. This algorithm's worst-case time complexity is O(n), where n is the length of the input array and and an average-case time complexity of O(n/2).

Code for implementing search in a sorted array efficiently:.

```
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    while low <= high:
```

```
        mid = (low + high) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

This implementation conducts a binary search through the input array, dividing the search space in half with each iteration. This algorithm's worst-case time complexity is O(log n), where n is the length of the input array.

Experiment code (search in a sorted array):

```
import numpy as np
 import matplotlib.pyplot as plt
import random
import time


def linear_search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1

def binary_search(arr, x):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
```

```
        high = mid - 1
    return -1

arr = [i for i in range(1000)]
x = random.randint(0, 999)

linear_times = []
binary_times = []

for i in range(100):
    start_time = time.time()
    linear_search(arr, x)
    end_time = time.time()
    linear_times.append(end_time - start_time)

    start_time = time.time()
    binary_search(arr, x)
    end_time = time.time()
    binary_times.append(end_time - start_time)

print("Linear search times:")
print(f"min={min(linear_times):.6f} avg={sum(linear_times)/len(linear_times):.6f}")

print("Binary search times:")
print(f"min={min(binary_times):.6f}
avg={sum(binary_times)/len(binary_times):.6f}")
```

This code creates an array of 1000 integers and a search target value x. It then runs 100 searches using both the linear and binary search algorithms, timing each one and saving the elapsed time in a list. Finally, the code displays the shortest and longest search times for each algorithm.

The program's output should show the shortest and longest search times for both algorithms over 100 trials. On large inputs, the binary search algorithm (which has worst-case time complexity O(log n)) should be much faster than the linear search algorithm (which has worst-case time complexity O(n)).

b. Code for inefficient implementation of priority queue insertion and extraction:

```
class PriorityQueue:
    def __init__(self):
        self.queue = []

    def insert(self, priority, item):
        self.queue.append((priority, item))

    def extract_min(self):
        if len(self.queue) == 0:
            return None
        min_index = 0
        for i in range(len(self.queue)):
            if self.queue[i][0] < self.queue[min_index][0]:
```

```
            min_index = i
        return self.queue.pop(min_index)[1]
```

Efficient implementation for insertion in and extraction from priority queue:

```
import heapq

class PriorityQueue:
    def __init__(self):
        self.queue = []
        self.index = 0

    def insert(self, priority, item):
        heapq.heappush(self.queue, (priority, self.index, item))
        self.index += 1

    def extract_min(self):
        if len(self.queue) == 0:
            return None
        return heapq.heappop(self.queue)[2]
```

ex3.5.b.py:

```
import random
import time
import heapq
import matplotlib.pyplot as plt

def inefficient_insert(queue, element):
    queue.append(element)
    queue.sort(reverse=True)
```

```
def inefficient_extract(queue):
    return queue.pop()

def efficient_insert(queue, element):
    heapq.heappush(queue, element)

def efficient_extract(queue):
    return heapq.heappop(queue)

def time_experiment(num_elements, num_trials):
    inefficient_times = []
    efficient_times = []

    for _ in range(num_trials):
        data = [random.randint(0, 1000) for _ in range(num_elements)]

        # Inefficient implementation
        inefficient_queue = []
        start_time = time.time()
        for item in data:
            inefficient_insert(inefficient_queue, item)
        for _ in range(num_elements):
            inefficient_extract(inefficient_queue)
        inefficient_times.append(time.time() - start_time)

        # Efficient implementation
        efficient_queue = []
        start_time = time.time()
        for item in data:
            efficient_insert(efficient_queue, item)
        for _ in range(num_elements):
            efficient_extract(efficient_queue)
        efficient_times.append(time.time() - start_time)

    return inefficient_times, efficient_times

def main():
```

```
    num_elements = 1000
    num_trials = 100

    inefficient_times, efficient_times = time_experiment(num_elements,
num_trials)

    plt.hist(inefficient_times, bins=20, alpha=0.5, label="Inefficient")
    plt.hist(efficient_times, bins=20, alpha=0.5, label="Efficient")
    plt.xlabel("Time (s)")
    plt.ylabel("Frequency")
    plt.legend(loc="upper right")
    plt.title("Priority Queue Insertion and Extraction: Inefficient vs Efficient")
    plt.show()

    print(f"Inefficient avg: {sum(inefficient_times) / num_trials:.6f}s")
    print(f"Efficient avg: {sum(efficient_times) / num_trials:.6f}s")

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

In this code, the inefficient implementation has a time complexity of O(n^2) for insertion and extraction combined (O(n*log(n)) for insertion and O(n) for extraction), while the efficient implementation has a time complexity of O(n*log(n)) for insertion and extraction combined (O(log(n)) for each operation).The experiment times the execution of both implementations on large inputs (1000 elements) and plots the distribution of measured values across multiple measurements (100 measurements per task). It also prints the average time for each implementation.