IFN 564 Assignment

02/11/2022

Lecture & Tutor:

- Dr Dimitri Perrin
- Mr Jake Bradford

Author:

- Phan Thao Nhi Nguyen (Nina) - n11232862

1. Data structures and Algorithms

a. Data structures

To choose appropriate data structures, I analysed some main features that need to complete for this system based on problem description.

Cinema screening room: use **Array**

_ Cinema screening room has fixed capacity room. There is no reserved seating, customer buy ticket and can seat everywhere. So, we can use an array data structure for this cinema screening room. Array size is the room capacity. We can use array index to access an element in this array.

Cinema waiting line: use **Queue**

_ When customers arrive to the cinema, they wait in line to be served. The line should be implemented as First come First served strategy. Then, customers are served by the time they got into line. Queue is the appropriate data structure in this situation because it is first in first out algorithm. The first person in line is the first person served until cinema sold out tickets.

Cinema random movies: use **Stack**

_ Cinema receive movies in random time. The schedule for new arrivals is quite irregular, so always screen the most recent movies. After the movie has been shown once, it is discarded.

_ By applying Stack which is a linear data structure and follows by Last In first Out order. We can easily demonstrate the receiving movies correctly.

Cinema customer: use **Binary Search Tree**

_ Cinema has many Customers. Every Customer has their information including their first name, last name, phone number, payment method, number of screenings. Once served a customer, we must check they exist in the system or not. If not, a new profile is created.

_ Every time served a customer; we search their first name and last name in system. We can conclude this action is happened frequently in system. So, we must choose a data structure is optimize with searching action.

_ We don't know exactly how many customers we must store in this cinema system. Storing with optimize memory should be considered here.

_ Every time a customer doesn't exist in system, we must insert them into system. Moreover, staffs can delete customer information in system.

From all above information, Binary Search Tree is the most appropriate data structure in this situation. Binary Search Tree has the advantages from linked list (don't need a fixed size) and optimized in searching.

Binary Search Tree is defined by the node's key, the left subtree, the right subtree. The left subtree contains values that are less than or equal to the node's value. The right subtree only contains values that are greater than or equal to the node's value. The left and right subtree are binary search tree. So, searching is efficiency in binary search tree.

b. The algorithms and pseudo code

Adding customer into system (Binary Search Tree)

To accomplish this algorithm. We need to check customer is exist in system or not. If not, we add this customer into the system. Customer is saving as object with their information: first name, last name, phone number, payment method, number of screenings.

If customer is already in system, increase their number of screenings (+1).

In searching a customer, we assumed that searching customers by their first name, last name and their names are distinct.

For compare_to algorithm:

- Input size: 1 which is the other customer to current customer
- Basic operation: the comparison current_name < other_name
  and current_name == other_name as the basic operations.
- The efficiency: O(1) for comparing 2 objects.

For search a customer algorithm:

- Input size: n which is the size of binary search tree or n is the numbers of nodes in a tree
- Basic operation: the comparison root.compare_to(k) as the basic operations.
- The best case: O(1) which is the query search node is the root
- The average case: O(logn). We have to keep travel through nodes by nodes in between root node to deepest node.
- The worst case: O(n). In this case we must go through from root node to deepest node in binary search tree.

For insert a customer algorithm:

- Input size: n which is the size of binary search tree or n is the numbers of nodes in a tree
- Basic operation: the comparison root.compare_to(k) as the basic operations.
- The best case: O(1). Inserting the root node into empty tree.
- The average case: O(logn). Just like searching complexity, we must keep travel through nodes by nodes in between root node to deepest node.
- The worst case: O(n). In this case we must go through from root node to deepest node in binary search tree.

```
ALGORITHM compare_to (other)

// compare customer last name + first name

// current last name + first name < other last name + first name => return -1

// current last name + first name > other last name + first name => return 1

// current last name + first name = other last name + first name => return 0

current_name = last_name + full_name

other_name = other.last_name + other.full_name
```

```
if current_name < other_name

        return -1

if current_name == other_name

        return 0

else

        return 1

##########################################################################

// Implement for binary search tree from here for search, insert customers

ALGORITHM search (root, k)

// Checking k is exist in binary tree or not. Return true if exist. Return false it it's not

if root != null

        if root.compare_to(k) == 0

                return true

        else

                if root.compare_to(k) == -1

                        if root.right_child != null

                                return root.right_child.search(k)

                else

                        if root.left_child != null

                                return root.left_chid.search(k)

else

return false

################################################################

ALGORITHM insert (k, root)

If root = null

        root = k

else

        if root.compare_to(k) == 1
```

```
                if root.left_child == null
                        root.left_child = BTree(k)
                else
                        root.left_child.insert(k)
        else if root.compare_to(k) == -1
                if root.right_child == null
                        root.right_child = BTree(k)
                else
                        root.right_child.insert(k)
```

Remove a customer (Binary Search Tree)
We applied deletion in binary search for removing a customer algorithm

For delete a customer algorithm:

- Input size: n which is the size of binary search tree and a customer object
- Basic operation: First the comparison ptr.key < K which move to the left child. Second, the comparison ptr.left_child != None and ptr.right_child != None to identify the parent of subtree has 2 nodes. Third, the else condition of ptr.left_child != None and ptr.right_child != None for the item has no or only one child.
- The best and average efficiency: O(logn). In this case, we must traverse from h comparisons for searching a node. Then delete and adjust their subtree (if needed).
- The worst case: O(n). In this case we must go through from root node to deepest node in binary search tree.

```
 ALGORITHM delete(root, k):
// pre: true
// post: an occurrence of item is removed from the binary search tree
//if item is in the binary search tree
    ptr ← root
    parent ← Null
    while ptr != Null and ptr.key != k:
        parent ← ptr
```

```
        if ptr.key.compare_to(k) == 1:
            ptr = ptr.left_child
        else:
            ptr = ptr.right_child

    if ptr != Null:
        if ptr.left_child != Null and ptr.right_child != Null:
            if ptr.left_child.right_child == Null:
                ptr.key ← ptr.left_child
                ptr.left_child ← ptr.left_child.left_child
            else:
                p ← ptr.left_child
                pp ← ptr
                while p.right_child != Null:
                    pp ← p
                    p ← p.right_child
                ptr.key ← p.key
                pp.right_child ← p.left_child

        else
            if ptr.left_child != Null
                c ← ptr.left_child
            else
                c ← ptr.right_child

            if ptr == self.key # need to change root
                self.key ← c
            else
```

```
            if ptr == parent.left_child
              parent.left_child ← c
            else
              parent.right_child ← c
```

Receiving a new movie (Stack)

In here, we receive new movies in random times and store them in a collection. And we will display a latest movie. Also, a favourite movie saves as backup if no new movie arrived in a while. We use stack as collection for storing movies. (We use array to implement this Stack)

For push an item to stack algorithm:

- Input size: n is the size of the array
- Basic operation: stack.items = [item] + stack.items as the basic operations.
- The efficiency: O(1). We only need to add an item at the head of an array.

For receive_movie algorithm:

- Input size: follow by input size of push function
- Basic operation: the generate for a random number which simulator for random timing in cinema.
- The efficiency: O(1) which follows with push algorithm in stack.

```
// This stack is implemented by array
ALGORITHM push (stack, item)
// push item into stack
// Implement stack with LIFO rules
top ← top + 1
stack[top] ← item
#################################################################
ALGORITHM receive_movie()
// This algorithm is used for generate random time for receiving movie
// Then add that movie to stack
favorite_movie ← "Titanic"
m1 ← "Spider men"
if generate random from 1 to 3 == 3: // demonstrate random timing
```

```
        stack.push(m1)
else
        stack.push(favorite_movie)
return stack
```

Scheduling the next movie (Stack)

We use stack to store upcoming movies. When a movie arrives in random time, it will be stored into stack. Latest movie in stack will be showed for customers by pop that movie from stack. This movie also is discarded from stack.

For pop an item from stack algorithm:

- Input size: n is the size of the stack
- Basic operation: item ← stack[top] as the basic operations. This pop an item from stack which follows by rule LIFO (Last in first out)
- The efficiency: O(1). We implement this Stack by Array. So only an arithmetic performs here and it’s a constant time function.

```
// This stack is implemented by array
ALGORITHM pop (stack)
// pop an item from stack.
// Implement stack with LIFO rules
if stack is empty
    return null
  endif
  item ← stack[top]
  top ← top - 1
  return item
```

Serving customers (Queue)

_ Customers is added to waiting line which is implemented by Queue. First come first serve rule will be applied.
_ When a ticket sells, we check if customer existed customer in system. If yes, we check for their number of screenings, with 10 screenings they receive a free ticket.

For enqueue an item from queue algorithm:

- Input size: n is the size of the queue

- Basic operation: queue[pointer] ← data as the basic operations. This inserts an item to queue which follow rule FIFO (First In First Out)
- The efficiency: O(1). We only need to insert an item at the head of array because we implemented this queue by array.

For dequeue an item from queue algorithm:

- Input size: n is the size of the queue
- Basic operation: item = queue[front] as the basic operations. This step removes an item to queue which follow rule FIFO (First In First Out)
- The efficiency: O(1). We only need to remove the last item from array because we implemented queue by array.

For serve_customer algorithm:

- Input size: n is the size of queue and cinema capacity
- Basic operation: we choose found_cus == True and found_cus ← customers.search(query_customer) comparison as the basic operations. In general, those operations are performed often than any other.
- The efficiency: O(n) is n which n is the size of capacity. This algorithm loop follows with cinema capacity.
- This algorithm is also call search and insert customer algorithms so it will be included 2 algorithms efficiency.

```
// Implement Queue by array

ALGORITHM enqueue(queue, item)

// This algorithm receive a queue (implemented by stack), and item needs to add to queue

// Add an item and follow rule FIFO (First In First Out)

pointer ← pointer + 1

queue[pointer] ← data



ALGORITHM dequeue(queue)

// This algorithm dequeue an first in item from queue. And return that item

item = queue[front]

front ← front + 1

return item

##########################################################################

ALGORITHM prepare_waiting_line()
```

```
// This algorithm is using for adding customer into system

waiting_line ← Queue

fname = "Nina"

lname = "Nguyen"

phone_no = "99999999"

payment = "Visa"

waiting_line.enqueue(fname, lname, phone_no, payment)

##############################################################################

ALGORITHM serve_customer(capacity, query_customer, customers)

// This algorithm get customer from queue waiting line within cinema capacity (10)

// customers is Binary Search Tree

// customers is loaded from system list of customers

// query_customer is the customer want to buy ticket

for i ← 1 to capacity do

        found_cus ← customers.search(query_customer)

        if found_cus == True

                customers.insert(found_customer) // insert to Binary Search Tree

        else

                if found_cus.screens == 9 // Check if store screen is 9, this buying is 10 screens

                        Print "Get a free ticket"


queue.clear() // clear this queue line when sold all ticket of a day
```

c. Analyse these algorithms in terms of their efficiency

Array

_ *Insert customers into cinema room's seat:* we add customer index by index in fixed size array (room capacity). So:

- Insert: O(1). Inserting by index of array.

_ We use array to implement **Stack**, **Queue** in this project.

Queue: implement based on **Array**

_ *Insert/Enqueue customer into waiting line:*

- Enqueue: O(1). Items are added at the head of the array.

_ *Delete/Deque customer when a ticket is sold for them:*

- Deque: O(1). This gets the last item of array.

Stack: implement based on **Array**

_ *Insert the latest movie to movie list:*

- Push: O(1). This inserts new item to the head of array.

_ *Scheduling the next movie:*

- Pop: O(1). This gets the item at the head of array.

Binary Search Tree:

By defining the left subtree, right subtree and their values comparing with node in the case this tree is balance tree. This binary search tree with some below actions that can reach efficiency logn.

_ *Search customer by their name (first name + last name):*

- Search: O(logn).

_ *Search customer's number of screenings to check for a free ticket (10 screenings):*

- Search: O(logn).

_ *Delete a customer:*

- Deletion: O(logn).

_ *Insert customer into system:*

- Insert: O(logn).

2. Correctness testing and Testing results

a. Testing for correctness explanations

No	Algorithm	Typical Components	Test cases	Expected result
1	Adding a customer (Binary search Tree)	Adding customer into binary search tree	Adding a customer to root. (**Error! Reference source not found.**)	Add successfully

			Adding 10 customers and validate result. (Figure 2 and Figure 3)	Add successfully. Customers are in correctly order
			Adding 1000 customers. (Figure 4, Figure 5, Figure 6)	Add successfully. Customers are in correctly order. Select a customer in csv file correctly.
		Search a customer in binary search tree	Search a customer that is a root (Figure 1)	Return customer information
			Search a customer in the left side (Figure 5)	Return customer information
			Search a customer in the right side (Figure 6)	Return customer information

```
c0 = Customer('Nina', 'Nguyen','0450343022', 'Paypal', random.randint(1, 10))
tree = BTree(c0)
print(tree.key)
```

✓ 0.1s

```
Nina Nguyen, phone: 0450343022, payment: Paypal, screen number: 1
```

Figure 1: insert a customer as a root

```
c1 = Customer('Jim', 'Tomkinson', '0450343234', 'Paypal', 5)
c2 = Customer('Richy', 'AAAAA', '13434343', 'Cash', 9)
c3 = Customer('Nina', 'Aguyen','0450343022', 'Paypal', 0)
c4 = Customer('Apple', 'Zruit','3243434334', 'Paypal', 6)
c5 = Customer('Apple', 'Nguyen','4324342', 'Paypal', 3)
c6 = Customer('Orange', 'Yran', '0450343234', 'Paypal', 1)
c7 = Customer('Hope', 'Pomkinson', '13438343', 'Cash', 7)
c8 = Customer('Gigi', 'Bguyen','0450342022', 'Paypal', 8)
c9 = Customer('Hadid', 'Bert','34234541454', 'Paypal', 2)
tree.insert(c1)
tree.insert(c2)
tree.insert(c3)
tree.insert(c4)
tree.insert(c5)
tree.insert(c6)
tree.insert(c7)
tree.insert(c8)
tree.insert(c9)
```

Figure 2: insert 10 customers

```
tree.display()
✓ 0.3s

      _________________________________________________Nina Nguyen____________________
     /                                                                                 \
Richy AAAAA_____                                                          ______Jim Tomkinson________________
                \                                                        /                                    \
           Nina Aguyen___________________________                 Hope Pomkinson                   _____Apple Zruit
                                                 \                                                /
                                      _____Apple Nguyen                                      Orange Yran
                                     /
                            ____Gigi Bguyen
                           /
                      Hadid Bert
```

Figure 3: validate after inserting customers (Ref code: https://stackoverflow.com/a/54074933)

```python
c0 = Customer('Nina', 'Nguyen','0450343022', 'Paypal', 9)
# inital binary search tree with root
db_customer = BTree(c0)
# Db csv file with 1000 rows
with open('db.csv', newline='') as csv_file:
    reader = csv.reader(csv_file)
    next(reader, None)  # Skip the header.
    # Unpack the row directly in the head of the for loop.
    for fname, lname, phone, payment_method, screenNo in reader:
        # Convert the numbers to floats.
        fname = fname
        lname = lname
        phone = phone
        payment_method = payment_method
        screenNo = screenNo
        # Now create the Student instance and append it to the list.
        db_customer.insert(Customer(fname, lname, phone, payment_method, screenNo))
```

Figure 4: insert 1000 customers from csv file

```python
print(db_customer.key)
print(db_customer.left_child)
print(db_customer.left_child.left_child)
print(db_customer.right_child)
print(db_customer.right_child.right_child)
```

✓ 0.4s

```
Nina Nguyen, phone: 0450343022, payment: Paypal, screen number: 9
Buck Josovitz, phone: 8099573339, payment: jcb, screen number: 1
Lanie Jenik, phone: 5761740692, payment: jcb, screen number: 9
Jayme Primrose, phone: 4906771247, payment: jcb, screen number: 4
Tito Whipple, phone: 6311814303, payment: maestro, screen number: 6
```

Figure 5: Verify print customer from binary tree

```python
## Validate search exist customer
print(db_customer.search(Customer('Tito', 'Whipple', '','',0)))
```

✓ 0.6s

```
Tito Whipple, phone: 6311814303, payment: maestro, screen number: 6
```

Figure 6: Verify a customer in csv file

2		Adding customer into Queue	Adding a customer into	Add successfully

	Adding customers to waiting line (Queue)		empty Queue. (Figure 7)	
			Adding 10 customers into Queue (Figure 8)	Add successfully
			Adding 3 customers into Queue (Figure 9)	
		Remove customer from Queue	Remove a customer from Queue with Queue order. (Figure 8 or Figure 9)	Remove successfully with correct order (First in First out)
			Remove a customer from empty Queue (Figure 10)	Queue empty

```python
# Add a customer to Queue
wc0 = Customer('Nina', 'Nguyen','0450343022', 'Paypal', 0)
line.add_to_line(wc0)
```

Figure 7: Add a customer to Queue

```python
wc0 = Customer('Nina', 'Nguyen','0450343022', 'Paypal', 0)
wc1 = Customer('Jim', 'Tran', '0450343234', 'Paypal', 0)
wc2 = Customer('Richy', 'Tomkinson', '13434343', 'Cash', 0)
wc3 = Customer('Nina', 'NNguyen','0450343022', 'Paypal', 0)
wc4 = Customer('Coco', 'Tran','34234545454', 'Paypal', 0)
wc5 = Customer('Apple', 'Nguyen','4324342', 'Paypal', 0)
wc6 = Customer('Orange', 'Tran', '0450343234', 'Paypal', 0)
wc7 = Customer('Hope', 'Tomkinson', '13438343', 'Cash', 0)
wc8 = Customer('Gigi', 'NNguyen','0450342022', 'Paypal', 0)
wc9 = Customer('Hadid', 'Bert','34234541454', 'Paypal', 0)
```

```python
line = LineCollection()
line.add_to_line(wc0)
line.add_to_line(wc1)
line.add_to_line(wc2)
line.add_to_line(wc3)
line.add_to_line(wc4)
line.add_to_line(wc5)
line.add_to_line(wc6)
line.add_to_line(wc7)
line.add_to_line(wc8)
line.add_to_line(wc9)
```

```python
# checking for line with 9 customers
print(len(line.waiting_line.items))

# Checking for dequeue with correctly order FIFO
print(line.pick_customer())
print(line.pick_customer())
print(line.pick_customer())
print(line.pick_customer())
print(line.pick_customer())
print(line.pick_customer())
print(line.pick_customer())
print(line.pick_customer())
print(line.pick_customer())

# checking for line with 1 customer
print(len(line.waiting_line.items))
line.isEmpty()
```

✓ 0.6s

```
10
Nina Nguyen, phone: 0450343022, payment: Paypal, screen number: 0
Jim Tran, phone: 0450343234, payment: Paypal, screen number: 0
Richy Tomkinson, phone: 13434343, payment: Cash, screen number: 0
Nina NNguyen, phone: 0450343022, payment: Paypal, screen number: 0
Coco Tran, phone: 34234545454, payment: Paypal, screen number: 0
Apple Nguyen, phone: 4324342, payment: Paypal, screen number: 0
Orange Tran, phone: 0450343234, payment: Paypal, screen number: 0
Hope Tomkinson, phone: 13438343, payment: Cash, screen number: 0
Gigi NNguyen, phone: 0450342022, payment: Paypal, screen number: 0
1
```

Figure 8: Add and Dequeue customers from Queue. Verify Dequeue process

```
# Add list of customers
# Dequeue all customers => Line should empty now
line.add_to_line(wc1)
line.add_to_line(wc2)
line.add_to_line(wc3)

print(line.pick_customer())
print(line.pick_customer())
print(line.pick_customer())

line.isEmpty()


```

```
✓ 0.4s

Jim Tran, phone: 0450343234, payment: Paypal, screen number: 0
Richy Tomkinson, phone: 13434343, payment: Cash, screen number: 0
Nina NNguyen, phone: 0450343022, payment: Paypal, screen number: 0

True
```

Figure 9: Add 3 customers into Queue and check empty queue after Dequeue

```
print(line.isEmpty())
print(line.pick_customer())
```

```
✓ 0.3s

True
None
```

Figure 10: Dequeue a customer from empty Queue

3	Serving customers	Search customer screens	Query customer number of screens (Figure 11 or Figure 12)	Return customer number of screens
			Checking a customer have 10 number of screens (Figure 11)	Return true. Receiving a free ticket
			Checking a customer have not enough 10 screens (Figure 12)	Return False. Not have enough number of screens
		Search non-existed customer (Figure 13)	Search a non-existed customer in system (Figure 13)	Return None
		Add non-existed customer to system (inserting to binary search tree) (Figure 13)	Add customer information to system (Figure 13)	Adding successfully
	Sell ticket for customer in waiting line	Cinema capacity is smaller or equal than number of customers in waiting line. (Capacity: 10, waiting line:15) (Figure 14)	Dequeue a customer from Queue (Figure 14)	Dequeue correctly in Queue order (FIFO)
			After dequeue, wating line still have left customers (Figure 14)	Size of waiting line is 5
			Clear waiting line (Figure 16)	Waiting line size is 0
		Cinema capacity is larger than number of customers in waiting line. (Capacity: 10, waiting line: 3) (Figure 15)	Dequeue a customer from Queue (Figure 15)	Dequeue correctly in Queue order (FIFO)
			Get all of customers in waiting line (Figure 15)	Queue is empty now

```python
c1 = Customer('Jim', 'Tomkinson', '0450343234', 'Paypal', 5)
c2 = Customer('Richy', 'AAAAA', '13434343', 'Cash', 9)
c3 = Customer('Nina', 'Aguyen','0450343022', 'Paypal', 0)
c4 = Customer('Apple', 'Zruit','3243434334', 'Paypal', 6)
c5 = Customer('Apple', 'Nguyen','4324342', 'Paypal', 3)
c6 = Customer('Orange', 'Yran', '0450343234', 'Paypal', 1)
c7 = Customer('Hope', 'Pomkinson', '13438343', 'Cash', 7)
c8 = Customer('Gigi', 'Bguyen','0450342022', 'Paypal', 8)
c9 = Customer('Hadid', 'Bert','34234541454', 'Paypal', 9)
```

```python
tree = BTree(c1)
tree.insert(c2)
tree.insert(c3)
tree.insert(c4)
tree.insert(c5)
tree.insert(c6)
tree.insert(c7)
tree.insert(c8)
tree.insert(c9)
```

Initial Data

```python
# Test case for a free ticket
checking_customer_2 = Customer('Hadid', 'Bert', '', '', '')

cus_db = tree.search(checking_customer_2)
if cus_db.get_no_of_screen() == 9:
    print("Receive a free ticket")
else:
    print(f'Customer no of screen: {cus_db.get_no_of_screen()}')
```

✓ 0.3s

```
Receive a free ticket
```

Figure 11: Query and Validate for a free ticket customer

```python
# test case for non-free ticket
checking_customer = Customer('Jim', 'Tomkinson', '', '', '')

cus = tree.search(checking_customer)
if cus.get_no_of_screen() == 9:
    print("Receive a free ticket")
else:
    print(f'Customer no of screen: {cus.get_no_of_screen()}')
```

✓ 0.4s

```
Customer no of screen: 5
```

Figure 12: Query and Validate for a non-free ticket customer

```
non_existed_cus = Customer('Justin', 'Beiber','7234541454', 'Paypal', 1)
print(tree.search(non_existed_cus))

print("AFTER INSERT. Seerch for customer information")
tree.insert(non_existed_cus)
print(tree.search(non_existed_cus))
```
✓ 0.6s

```
None
AFTER INSERT. Seerch for customer information
Justin Beiber, phone: 7234541454, payment: Paypal, screen number: 1
```

Figure 13: Check non-existed customer and verify after inserting non-existed customer

```
wc1 = Customer('Jim', 'Tran', '0450343234', 'Paypal', 0)
wc2 = Customer('Richy', 'Tomkinson', '13434343', 'Cash', 0)
wc3 = Customer('Nina', 'NNguyen','0450343022', 'Paypal', 0)
wc4 = Customer('Coco', 'Tran','34234545454', 'Paypal', 0)
wc5 = Customer('Apple', 'Nguyen','4324342', 'Paypal', 0)
wc6 = Customer('Orange', 'Tran', '0450343234', 'Paypal', 0)
wc7 = Customer('Hope', 'Tomkinson', '13438343', 'Cash', 0)
wc8 = Customer('Gigi', 'NNguyen','0450342022', 'Paypal', 0)
wc9 = Customer('Hadid', 'Bert','34234541454', 'Paypal', 0)
wc10 = Customer('Pep', 'Homkinson', '13438343', 'Cash', 0)
wc11 = Customer('Tin', 'Albetr','0450342022', 'Paypal', 0)
wc12 = Customer('Connor', 'Mc','34234541454', 'Paypal', 0)
wc13 = Customer('Pepsi', 'Test','34234541454', 'Paypal', 0)
wc14 = Customer('Tin', 'Albetr','0450342022', 'Paypal', 0)
wc15 = Customer('Connor', 'Mc','34234541454', 'Paypal', 0)
```

```
line4 = LineCollection()
line4.add_to_line(wc1)
line4.add_to_line(wc2)
line4.add_to_line(wc3)
line4.add_to_line(wc4)
line4.add_to_line(wc5)
line4.add_to_line(wc6)
line4.add_to_line(wc7)
line4.add_to_line(wc8)
line4.add_to_line(wc9)
line4.add_to_line(wc10)
line4.add_to_line(wc11)
line4.add_to_line(wc12)
line4.add_to_line(wc13)
line4.add_to_line(wc14)
line4.add_to_line(wc15)
```

```
# Servicing customers with cinema capacity
# Test case: capacity = 10. Receive 10 customers in line (15 customers)
cinema = Cinema()

cinema.add_customer(line4)

# Line should still have 5 customers left
len(line4.waiting_line.items)
```
✓ 0.4s

```
5
```

Figure 14: Test case cinema capacity 10, waiting line 15

```
wc1 = Customer('Jim', 'Tran', '0450343234', 'Paypal', 0)
wc2 = Customer('Richy', 'Tomkinson', '13434343', 'Cash', 0)
wc3 = Customer('Nina', 'NNguyen','0450343022', 'Paypal', 0)
```
[188] ✓ 0.3s

```
line3 = LineCollection()
line3.add_to_line(wc1)
line3.add_to_line(wc2)
line3.add_to_line(wc3)
```
[189] ✓ 0.4s

```
# Servicing customers with cinema capacity
# Test case: capacity = 10. Receive all customers in line
cinema = Cinema()

cinema.add_customer(line3)

print(f'Line size: {len(line3.waiting_line.items)}')
```
[191] ✓ 0.6s

```
... Line size: 0
```

Figure 15: Test case cinema capacity (10) is larger than waiting line size (3)

```
print(len(line4.waiting_line.items))
print("AFTER clear waiting line")
line4.clear_line()
print(len(line4.waiting_line.items))
```
[265] ✓ 0.7s

```
... 5
AFTER clear waiting line
0
```

Figure 16: Clear waiting line

4	Receiving a new movie (Stack)	Simulate receive a new movie test case (case: input number = random number)	Add this movie into stack (Figure 17)	Add successfully
		Simulate not receive any movie in random timing (case: input number != random number)	Add favourite movie to stack (Figure 18)	Add successfully

```
1  movies.receive_movie(1)
✓ 0.4s

######## Receive a new movie #########

<customize_collection.Stack at 0x1606ea29570>
```

Figure 17: Receive a new movie in random time

```
1  movies = MovieCollection()
2  movies.receive_movie()
✓ 0.3s
######## Waiting for too long for a new movie. So add a movie from favourite movie Titanic, duration: 200 ########
```

Figure 18: Test case get favourite movie

5	Scheduling movie (Stack)	Scheduling for movie	Print latest movie is random timing received movie (Figure 20)	Print correctly receiving movie
			Print latest movie is favourite movie (Figure 19)	Print correctly favourite movie
		Display movie description	Print movie information (Figure 19 or Figure 20)	Print correctly movie information

```
1  print(movies.display_movie())
[40]  ✓ 0.6s
···  Titanic, duration: 200
```

Figure 19: Display favourite movie

```
1  print(movies.display_movie())
✓ 0.7s
Normal people, duration: 120
```

Figure 20: Display correct latest movie which received from random time

3. Testing for efficiency

i. Binary Search Tree

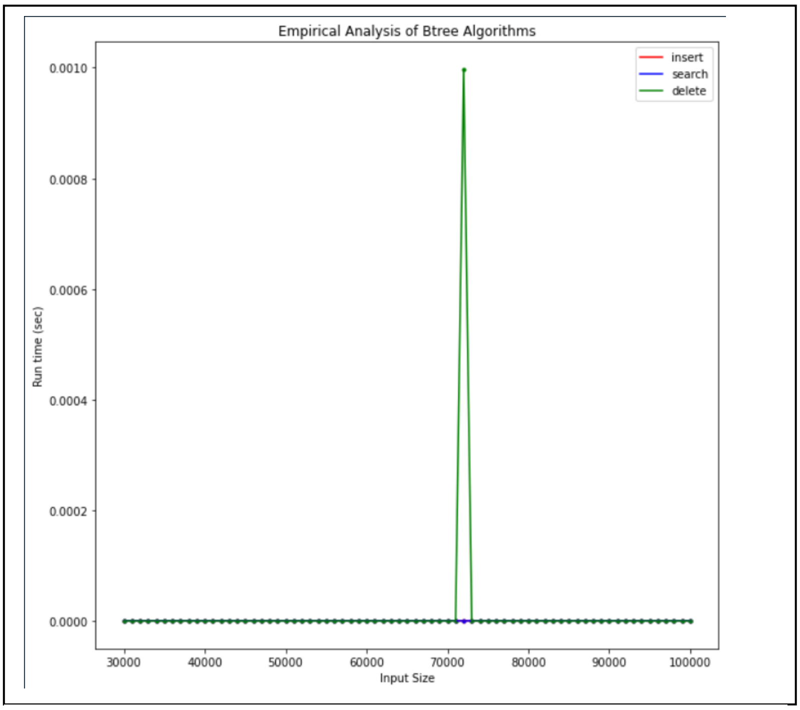

Figure 21: Empirical Analysis for Binary Search Tree

Input size for every algorithm: 80
_ start from 30_000
_ finish when 100_000
_ with step 1_000
This empirical analysis on Binary Search Tree for delete algorithm proves that it is Logarithmic. Because it has some certain decrease and conquer exploring.
For insert and delete are constant efficiency class which is independent of their input size.

ii. Stack Movies

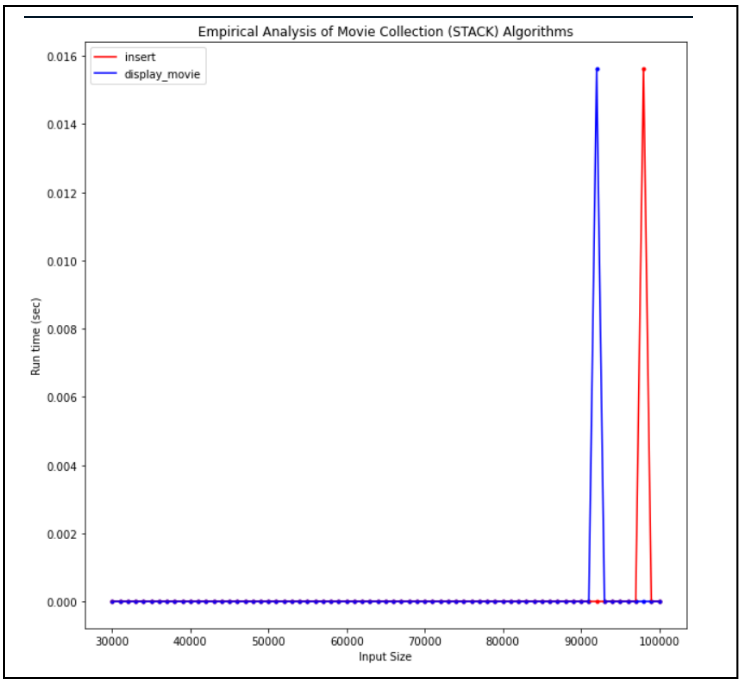

Figure 22: Empirical Analysis for Stack

Input size for every algorithm: 80
_start from 30_000
_finish when 100_000
_with step 1_000
This empirical analysis on insert and display_movie (push and pop) algorithms prove that they are Logarithmic efficiency class. They have certain decrease-and-conquer exploring based in this plot. This may happen because at the executed time computer runs other background tasks.

iii. Queue Waiting Line

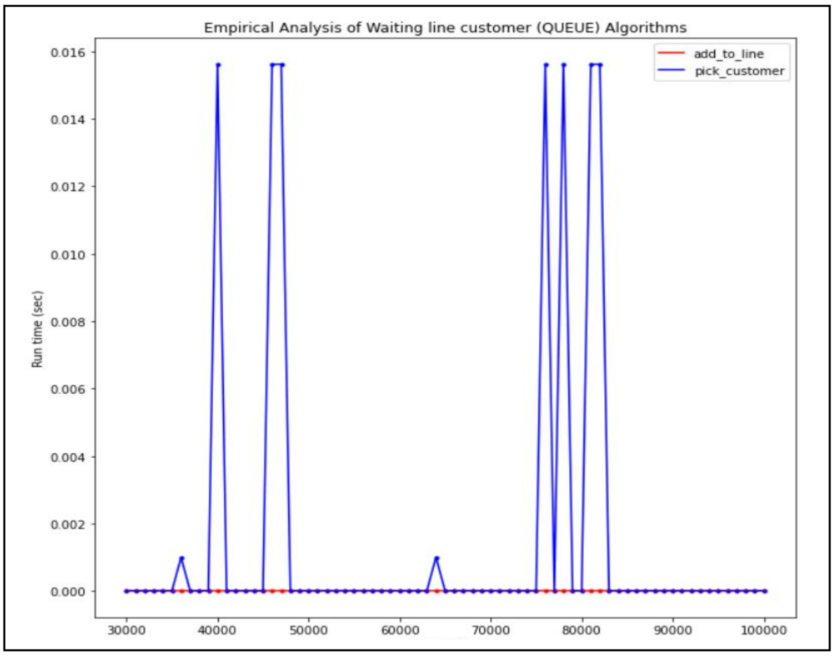

Figure 23: Empirical Analysis for Queue – Waiting Line

Input size for every algorithm: 80
_ start from 30_000
_ finish when 100_000
_ with step 1_000
This empirical analysis on pick_customer (dequeue) algorithm prove that it is Logarithmic efficiency class. By seeing the plot, we can see certain decrease-and-conquer exploring.
With the add_to_line (enqueue), it is Constant efficiency class. Because it performs algorithm independent of their input size. This may happen because at the executed time computer runs other background tasks.

4. Guideline for running Python code, automation testing and references

To execute the cinema: cinema_execute.py

testing.ipynb: This file is using for running automation testing

virtual.ipynb: This file is using for running empirical analysis

In binary_search_tree.py, we have two functions _display_aux() and display() that referenced from https://stackoverflow.com/a/54074933 post to print out binary tree.