Application of linear data structure

จัดทำโดย
นาย ภูรินทร์ จีนพวด 66070501043

เสนอ
ดร. ทวีชัย นันทวิสุทธิวงศ์
ดร. ปิยนิตย์ เวปุลานนท์

รายงานนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตร
ปริญญาวิศวกรรมศาสตร์บัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์
มหาวิทยาลุยเทคโนโลยีพระจอมเกล้าธนบุรี
ประจำปี 2566

บทที่ 1 บทนำ

1.1 ที่มาและความสำคัญ

ชิงช้าสวรรค์เป็นสถานที่ท่องเที่ยวที่ได้รับความนิยมอย่างกว้างขวางในสวนสนุกและงานรื่นเริง การจัดกลุ่มผู้นั่งบนชิงช้าสวรรค์เป็นสิ่งสำคัญอย่างยิ่งในการลดเวลารอ เพิ่มความเพลิดเพลินให้กับผู้นั่ง และรับประกันความปลอดภัย โดยปกติแล้ว การจัดสรรผู้โดยสารบนชิงช้าสวรรค์เป็นหน้าที่ความรับผิดชอบของเจ้าหน้าที่ ซึ่งนำไปสู่ปัญหาต่างๆมากมาย ความล่าช้า ข้อผิดพลาด และการขาดความชัดเจน รวมถึงความเสี่ยงที่อาจจะเกิดขึ้น

จากที่กล่าวมาข้างต้น คณะผู้จัดทำเกิดความสนใจที่จะออกแบบโครงสร้างการจัดคนขึ้นชิงช้าสวรรค์เพื่อแก้ไขปัญหาที่เกิดขึ้น โดยการประยุกต์ใช้โครงสร้างข้อมูลเชิงเส้น (Linear data structure) มาช่วยในการจำลอง และอธิบายการจัดลำดับคนขึ้นชิงช้าสววรค์อย่างรวดเร็ว มีประสิทธิภาพ เที่ยงตรง และปลอดภัย

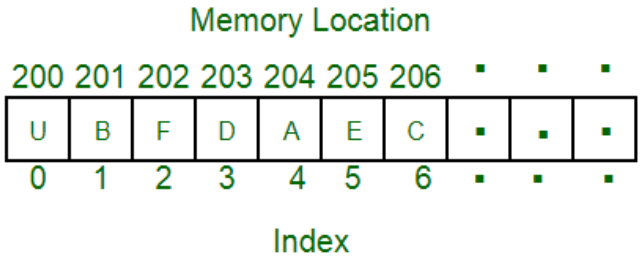
1.2 วัตถุประสงค์

1. เพื่อประยุกต์ใช้โครงสร้างข้อมูลเชิงเส้น (Linear data structure) มาช่วยในการจำลอง และอธิบายการจัดลำดับคนขึ้นชิงช้าสววรค์

บทที่ 2 ทฤษฎีความรู้

2.1 อาร์เรย์ (Array)

Array คือโครงสร้างข้อมูลที่เก็บคอลเลกชันขององค์ประกอบ โดยแต่ละองค์ประกอบระบุด้วยดัชนีหรือคีย์ องค์ประกอบในอาร์เรย์จะถูกจัดเก็บไว้ในตำแหน่งหน่วยความจำที่ต่อเนื่องกัน และเป็นประเภทข้อมูลเดียวกัน ซึ่งหมายความว่าคุณสามารถเข้าถึงองค์ประกอบในอาร์เรย์ได้โดยใช้ดัชนีหรือตำแหน่ง



(ภาพที่ 2.1 โครงสร้างข้อมูลแบบ array)

Array แบ่งออกเป็น 3 ประเภท ได้แก่1

1. อาร์เรย์หนึ่งมิติ: อาร์เรย์ประเภทที่ง่ายที่สุดคืออาร์เรย์หนึ่งมิติซึ่งมีองค์ประกอบแถวเดียว ด้วยการใช้หมายเลขดัชนีที่กำหนด ทำให้สามารถเข้าถึงแต่ละองค์ประกอบในอาร์เรย์ได้อย่างสะดวก
2. อาร์เรย์สองมิติ: อาร์เรย์ประเภทถือได้ว่าเป็นอาร์เรย์ของอาร์เรย์หรือเป็นเมทริกซ์ที่ประกอบด้วยแถวและคอลัมน์
3. อาร์เรย์หลายมิติ: อาร์เรย์ประเภทนี้ประกอบด้วยอาร์เรย์หลายตัวที่จัดเรียงตามลำดับชั้น สามารถมีมิติข้อมูลจำนวนเท่าใดก็ได้ โดยทั่วไปจะเป็นสองมิติ (แถวและคอลัมน์) แต่อาจใช้สามมิติขึ้นไปก็ได้

ข้อดีของโครงสร้างข้อมูล Array

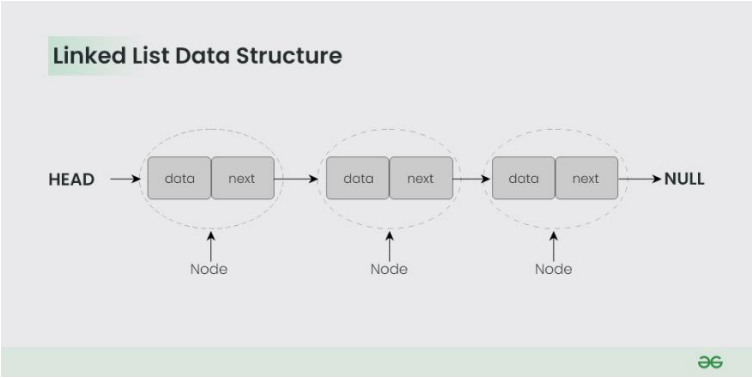
1. สามารถเข้าถึง และเรียกได้อย่างรวดเร็วเนื่องจากข้อมูลถูกจัดเก็บไว้ในตำแหน่งหน่วยความจำที่อยู่ติดกัน
2. สามารถใช้เพื่อจัดเก็บข้อมูลประเภทต่างๆ ได้หลากหลาย แม้แต่โครงสร้างข้อมูลที่ซับซ้อน เช่น วัตถุและพอยน์เตอร์
3. ใช้งานและเข้าใจได้ง่าย

ข้อเสียของโครงสร้างข้อมูลอาเรย์

1. อาร์เรย์มีขนาดคงที่ซึ่งกำหนดไว้ในขณะที่ประกาศตัวแปร ซึ่งหมายความว่าจะไม่สามารถลดหรือเพิ่มขนาดได้ขณะที่โปรแกรมทำงานอยู่
2. หากอาร์เรย์ไม่ได้รับการเติมข้อมูลจนเต็ม หน่วยความจำจะเสียพื้นที่ในการจองข้อมูลให้อาร์เรย์อย่างเสียเปล่า
3. อาร์เรย์มีการรองรับที่จำกัดสำหรับประเภทข้อมูลที่ซับซ้อน แต่องค์ประกอบของอาร์เรย์ทั้งหมดต้องเป็นประเภทข้อมูลเดียวกัน

2.2 รายการเชื่อมโยง (Linked List)

Linked list คือ โครงสร้างข้อมูลพื้นฐานที่ประกอบด้วยโหนดที่แต่ละโหนดมีข้อมูลและการอ้างอิง (ลิงก์) ไปยังโหนดถัดไปในลำดับ ช่วยให้สามารถจัดสรรหน่วยความจำแบบไดนามิกและดำเนินการแทรกและลบได้อย่างมีประสิทธิภาพเมื่อเปรียบเทียบกับอาร์เรย์



(ภาพที่ 2.2 โครงสร้างข้อมูลแบบ Linked list)

Linked list แบ่งออกเป็น 3 ประเภท ได้แก่

1. Singly linked list เป็นโครงสร้างข้อมูลเชิงเส้นซึ่งองค์ประกอบต่างๆ ไม่ได้ถูกจัดเก็บไว้ในตำแหน่งหน่วยความจำที่อยู่ติดกัน และแต่ละองค์ประกอบจะเชื่อมต่อกับองค์ประกอบถัดไปโดยใช้พอยน์เตอร์เท่านั้น
2. Doubly Linked List เป็นรายการเชื่อมโยงชนิดพิเศษซึ่งแต่ละโหนดจะมีตัวชี้ไปยังโหนดก่อนหน้าและโหนดถัดไปของรายการที่เชื่อมโยง
3. Circular Linked List เป็นรายการเชื่อมโยงที่โหนดทั้งหมดเชื่อมต่อกันเป็นรูปวงกลม โหนดแรกและโหนดสุดท้ายจะเชื่อมต่อถึงกัน

ข้อดีของโครงสร้างข้อมูล Linked list

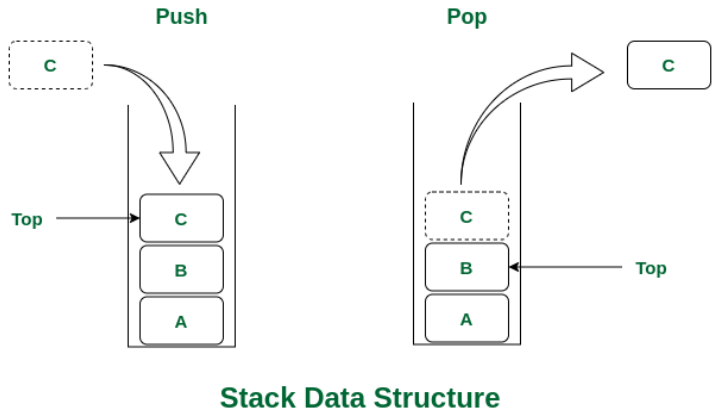
1. Linked List สามารถขยายหรือย่อขนาดได้อย่างง่ายดายระหว่างการทำงานของโปรแกรม ทำให้เป็นโครงสร้างข้อมูลที่ยืดหยุ่น และไม่ต้องจองพื้นที่ล่วงหน้า
2. การแทรกหรือการลบองค์ประกอบใน Linked List จะมีประสิทธิภาพมากกว่าในอาร์เรย์
3. Linked List อาจมีประสิทธิภาพด้านหน่วยความจำมากกว่าอาร์เรย์ เมื่อต้องรับมือกับโครงสร้างข้อมูลที่กระจัดกระจาย

ข้อเสียของโครงสร้างข้อมูล Linked list

1. แต่ละโหนดใน Linked List ต้องใช้หน่วยความจำเพิ่มเติมเพื่อจัดเก็บทั้งข้อมูลและการอ้างอิงไปยังโหนดถัดไป ส่งผลให้ใช้พื้นที่เก็บข้อมูลเยอะกว่า array
2. การใช้และการจัดการรายการที่เชื่อมโยงอาจซับซ้อนกว่าอาร์เรย์

2.3 สแต็ค (stack)

Stack คือ linear data structure ที่เป็นรูปแบบของการจัดเก็บข้อมูลโดยมีการจัดเรียงข้อมูลให้ต่อเนื่องกัน และเข้าถึงข้อมูลตามลำดับ โดย Stack จะมีรูปแบบกันจัดเก็บข้อมูลในรูปแบบข้อง Last In First Out (LIFO)



(ภาพที่ 2.3 โครงสร้างข้อมูลแบบ stack)

ข้อดีของโครงสร้างข้อมูล stack

1. Stack เป็นโครงสร้างข้อมูลที่เรียบง่ายและเข้าใจง่าย ทำให้เหมาะสำหรับการใช้งานที่หลากหลาย
2. การดำเนินการ push และ pop บนสแต็กสามารถทำได้ในเวลาคงที่ O(1) ทำให้สามารถเข้าถึงข้อมูลได้อย่างมีประสิทธิภาพ

ข้อเสียของโครงสร้างข้อมูล stack

1. องค์ประกอบใน stack สามารถเข้าถึงได้จากด้านบนเท่านั้น ทำให้ยากต่อการดึงหรือแก้ไของค์ประกอบที่อยู่ตรงกลางของ stack
2. Stack มีความจุคงที่ ซึ่งอาจเป็นข้อจำกัดได้ หากไม่ทราบจำนวนองค์ประกอบที่ต้องจัดเก็บ

2.4 คิว (queue)

Queue คือ linear data structure ที่เป็นรูปแบบของการจัดเก็บข้อมูลโดยมีการจัดเรียงข้อมูลให้ต่อเนื่องกัน โดย Queue จะมีรูปแบบกันจัดเก็บข้อมูลในรูปแบบข้อง First In First Out (FIFO)

Queue Data Structure

(ภาพที่ 2.4 โครงสร้างข้อมูลแบบ queue)

Queue แบ่งออกเป็น 4 ประเภท ได้แก่

1. Simple Queue เป็นโครงสร้างข้อมูลเชิงเส้นที่เป็นไปตามหลักการเข้าก่อนออกก่อน (FIFO) โดยองค์ประกอบต่างๆ จะถูกเพิ่มเข้าทางด้านหลัง และออกจากด้านหน้า
2. Circular Queue เป็นเวอร์ชันขยายของคิวปกติที่องค์ประกอบสุดท้ายของคิวเชื่อมต่อกับองค์ประกอบแรกของคิวที่สร้างเป็นวงกลม
3. Priority Queue เป็นประเภทของคิวที่จัดเรียงองค์ประกอบตามค่าลำดับความสำคัญ โดยทั่วไปองค์ประกอบที่มีค่าลำดับความสำคัญสูงกว่าจะถูกดึงข้อมูลก่อนองค์ประกอบที่มีค่าลำดับความสำคัญต่ำกว่า
4. Double Ended Queue คือประเภทของคิวที่สามารถทำการแทรกและถอดองค์ประกอบจากด้านหน้าหรือด้านหลังได้ ดังนั้นจึงไม่เป็นไปตามกฎ FIFO (เข้าก่อนออกก่อน)

ข้อดีของโครงสร้างข้อมูล Queue

1. สามารถจัดการข้อมูลจำนวนมากได้อย่างมีประสิทธิภาพได้อย่างง่ายดาย
2. การดำเนินการต่างๆ เช่น การแทรกและการลบสามารถดำเนินการได้อย่างง่ายดาย เนื่องจากเป็นไปตามกฎเข้าก่อนออกก่อน
3. เป็นโครงสร้างข้อมูลที่เรียบง่ายและเข้าใจง่าย ทำให้เหมาะสำหรับการใช้งานที่หลากหลาย

ข้อเสียของโครงสร้างข้อมูล Queue

1. การดำเนินการ เช่น การแทรกและการลบองค์ประกอบจากตรงกลางนั้นใช้เวลานาน
2. การค้นหาองค์ประกอบต้องใช้เวลา O(N)

บทที่ 3 วิธีการดำเนินงาน

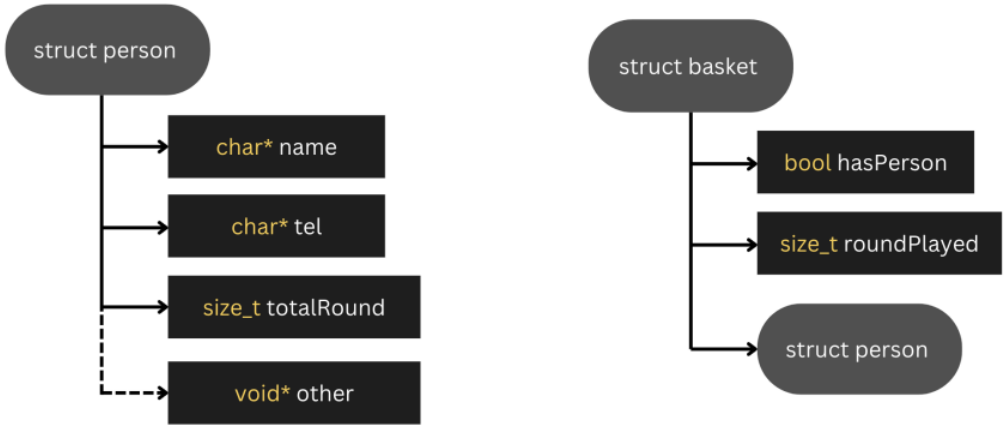
3.1 ออกแบบการประยุกต์ใช้โครงสร้างข้อมูลเชิงเส้น

3.1.1 ประเภทของโครงสร้างข้อมูลที่นำมาประยุกต์ใช้

ในการจำลองการแก้ปัญหา และอธิบายการจัดคนขึ้นเครื่องเล่นชิงช้าสวรรค์ ทางคณะผู้จัดทำเห็นเหมาะสมที่จะเลือกใช้ Simple Queue จัดการเรื่องการต่อคิวขึ้นชิงช้าสวรรค์ เนื่องจาก Simple Queue นั้นทำงานตามหลัก FIFO ทำให้ลดความล่าช้า และเพิ่มความชัดเจนของลำดับ ไม่มีการแทรกคิว และเลือก Circular Linked list มาจัดการในเรื่องของรอบการหมุนของชิงช้าสวรรค์ เนื่องจากทำการค้นหาข้อมูลได้อย่างรวดเร็ว

3.1.2 โครงสร้างของข้อมูล

โครงสร้างข้อมูลที่สร้างขึ้นมาใช้ มีอยู่ 2 อย่าง ได้แก่ person สำหรับเก็บโครงสร้างข้อมูลผู้คน และ baseket สำหรับเก็บโครงสร้างข้อมูลกระเช้าบนชิงช้าสวรรค์ โดยในรายงานนี้กำหนดให้ 1 basket สามารถบรรจุผู้คนได้ 1 คน

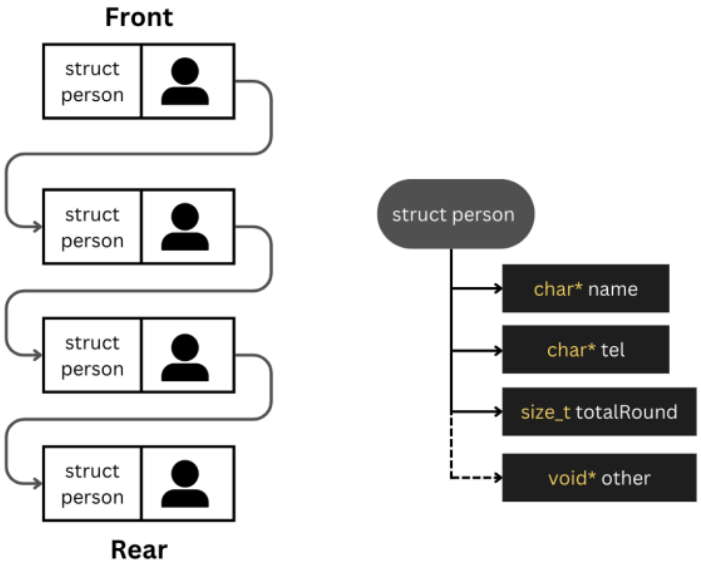


(ภาพที่ 3.1 โครงสร้างข้อมูล)

3.1.3 การนำโครงสร้างข้อมูลมาประยุกต์ใช้

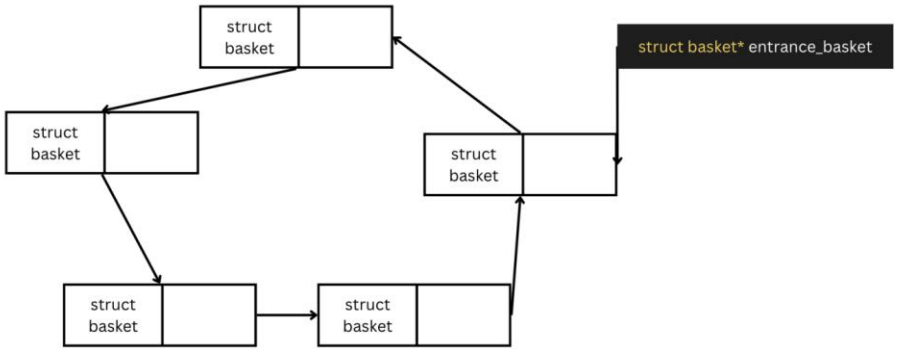
การนำโครงสร้างข้อมูลมาประยุกต์ใช้เพื่อจัดลำดับคนขึ้นชิงช้าสวรรค์ แบ่งเป็น 2 ช่วงดังนี้

1. การจัดลำดับคิวของผู้คนก่อนขึ้นชิงช้าสวรรค์ โดยการนำ person ใส่ simple queue เพื่อตัวแถวก่อนเล่นเครื่องเล่น



(ภาพที่ 3.2 โครงสร้างข้อมูลการต่อคิวก่อนขึ้นชิงช้าสวรรค์)

2. การจัดคนจากแถวเพื่อขึ้นกระเช้าของชิงช้าสวรรค์ ก่อนที่จะนำคนขึ้นกระเช้า เริ่มต้นจากการตรวจสอบสถานะว่ากระเช้ามีคนอยู่แล้วหรือไม่ จาก basket->hasPerson หากว่ามีคนทำการตรวจสอบจำนวนรอบที่กระเช้าหมุนไปว่าครบจำนวนรอบที่ซื้อหรือไม่ จาก basket->roundPlayed และ person->ToralRound หากครบแล้วให้นำคนที่อยู่ในกระเช้าออก และ pop คนที่อยู่ตำแหน่ง first ของ Simple queue หรือแถว เข้ามาแทน และทำการตรวจสอบ basket ถัดไป เปรียบเสมือนการหมุนของชิงช้าสวรรค์



(ภาพที่ 3.3 โครงสร้างข้อมูลของชิงช้าสวรรค์)

```
if entrance_basket hasPerson
    if roundPlayed >= totalRound
        hasPerson = False

if entrance_basket not
hasPerson
    if queue not empty
        basket.person = queue.top
        queue.pop

 next basket
```

initialize

```
create circular linked list length n
    init all node:
        hasPerson = False
        struct person = NULL
```

add new person to queue

```
push queue
```

(ภาพที่ 3.4 Psudocode การทำงานเบื้องต้น)

Lab & Assignment review

LAB 0 Warm Up!

- **Lab 0.1 : Sets**

```
for(int i = 0; i<length; i++){
    int check = 0;
    scanf(" %d", &number);
    for(int j=0; j<count; j++){
        if(number==elements[j]){
            check = 1;
        }
    }
    if(check == 0){
        elements[count] = number;
        count += 1;
    }
}
```

สิ่งที่สำคัญในการสร้างเซ็ท คือเซ็ทจะไม่เอาตัวเลขซ้ำ ในการวนลูปเพื่อรับตัวเลข จึงต้องนำตัวเลขที่นำเข้า ตรวจสอบกับตัวเลขทั้งหมดในเซ็ทโดยใช้ nested loop

- **Lab 0.2 : The Foundation of Set**

```c
void universal_get(int *universal , int init_number, int final_number){
    int i = 0;
    while(init_number <= final_number){
        //printf("init_number = %d\n" , init_number);
        universal[i] = init_number;
        i++;
        init_number++;
    }
}
```

ฟังก์ชัน universal_get มีไว้เพื่อหาตัวเลขทั้งหมดในช่วงด้วยการวนลูป

```c
void set_get(int* set_elements, int* set_length, int universal[], int length_universal){
    int count = 0;

    for(int i = 0; i<*set_length; i++){
        int check = 0;
        for(int j=0; j<count; j++){
            if(set_elements[i]==set_elements[j]){
                check += 1;
            }
        }
        if(check == 0){
            set_elements[count] = set_elements[i];
            count += 1;
        }
    }
```

ฟังก์ชัน set_get มีไว้เพื่อสร้างเซ็ทจากตัวเลขที่นำเข้า เช่นเดียวกับ Lab 0.1 แต่เนื่องจากในข้อนี้มีช่วงของ universal ดังนั้นการจะสร้างเซ็ทจึงต้องคำนึงถึงช่วงของ universal ด้วย โดยช่วงของ universal นั้นได้มาจากฟังก์ชัน universal_get

```c
*set_length = count;
count = 0;
for(int i=0; i<*set_length; i++){
    int check = 0;
    for(int j=0; j<length_universal; j++){
        if(set_elements[i] == universal[j]){
            check += 1;
        }
    }
    if(check > 0){
        set_elements[count] = set_elements[i];
        count += 1;
    }
}
*set_length = count;
```

จึงจะต้องมีการใช้ nested loop เพื่อตรวจสอบตัวเลขทั้งหมดที่เหมือนกัน ใน set(array) และ universal(array)

หลังจากทำการ optimize setA และ setB แล้ว จึงสามารถนำมาดำเนินการต่อได้

```c
void Union(int setA[], int setB[], int length_setA, int length_setB, int universal[], int length_universal){
    int i;
    int union_length=length_setA+length_setB;
    int union_elements[length_setA+length_setB];

    for(i=0; i<length_setA; i++){
        union_elements[i] = setA[i];
    }
    union_length = i;
    for(i=0; i<length_setB; i++){
        union_elements[union_length] = setB[i];
        union_length++;
    }
    selectionSort(union_elements, union_length);
    set_get(union_elements, &union_length, universal, length_universal);
}       You, 5 days ago • add Lab & Assignment Files
```

ฟังก์ชัน Union มีไว้เพื่อหาผล union ของทั้งสองเซ็ท โดยการวนลูปเพื่อคัดลอกตัวเลขทั้งหมดจาก set A และ set B มาต่อกันใน array ใหม่ซึ่งมีชื่อว่า union_elements จากนั้นทำการ sort ข้อมูล เพื่อเรียงตัวเลขจากน้อยไปมาก และเนื่องจากตัวเลขที่คัดลอกมาต่อกันนั้น มีโอกาสซ้ำกันจึงจะต้องนำเข้าฟังก์ชัน set_get เพื่อ optimize ให้ถูกต้อง

```c
void Intersection(int setA[], int setB[], int length_setA, int length_setB, int universal[], int length_universal){
    int count=0;
    int intersection_length=length_setA;
    int intersection_elements[intersection_length];

    for(int i=0; i<length_setA; i++){
        for(int j=0; j<length_setB; j++){
            if(setA[i] == setB[j]){
                intersection_elements[count] = setA[i];
                count += 1;
            }
        }
    }
    intersection_length = count;
    print_elements(intersection_elements, intersection_length);
}
```

ฟังก์ชัน Intersection มีไว้เพื่อหาผล intersection ของทั้งสองเซ็ท โดยการใช้ nested loop เพื่อตรวจสอบตัวเลขทั้งหมดในทั้งสองเซ็ท หากว่ามีตัวเลขที่เหมือนกันในทั้งสองเซ็ท จะทำการเก็บตัวเลขนั้นลงใน array ใหม่ที่มีชื่อว่า intersection_elements

```
void Difference(int set1[], int set2[], int length_set1, int length_set2, int universal[], int length_universal){
    int count = 0;
    int difference_length = length_set1+length_set2;
    int difference_elements[difference_length];

    for(int i=0; i<length_set1; i++){
        int check = 0;
        for(int j=0; j<length_set2; j++){
            if(set1[i] == set2[j]){
                check = 1;
                break;
            }
        }
        if(check == 0){
            difference_elements[count] = set1[i];
            count += 1;
        }
    }
    difference_length = count;
    print_elements(difference_elements, difference_length);
}
```

ฟังก์ชัน Difference มีไว้เพื่อหาผล difference ของทั้งสองเซ็ท โดยการใช้ nested loop เพื่อตรวจสอบตัวเลขทั้งหมดในทั้งสองเซ็ท หากตัวเลขนั้นมีใน set1 และไม่มีใน set2 จะทำการเก็บตัวเลขลงใน array ใหม่ซึ่งมีชื่อว่า difference_elements ฟังก์ชันนี้สำคัญที่ argument จะต้องใส่ให้ถูกว่าเซ็ทไหนเป็นเซ็ทตั้ง เซ็ทไหนเป็นเซ็ทลบ

```
void Complement(int set_elements[], int set_length, int universal[], int length_universal){
    int count = 0;
    int complement_length = length_universal;
    int complement_elements[complement_length];
    for(int i=0; i<length_universal; i++){
        int check = 0;
        for(int j=0; j<set_length; j++){
            if(set_elements[j] == universal[i]){
                check = 1;
                break;
                // printf("element >%d", set1[i], set2[j]);
            }
        }
        if(check == 0){
            complement_elements[count] = universal[i];
            count += 1;
        }
    }
    complement_length = count;
    print_elements(complement_elements, complement_length);
}
```

ฟังก์ชัน Complement มีไว้เพื่อหาผล complement ของเซ็ท โดยการใช้ nested loop เพื่อตรวจสอบตัวเลขทั้งหมดใน set และช่วงของ universal หากตัวเลขนั้นมีในช่วงของ universal แต่ไม่มีใน set จะทำการเก็บตัวเลขนั้นลงใน array ใหม่ซึ่งมีชื่อว่า complement_elements

- **สิ่งที่ได้เรียนรู้จาก LAB 0**

การจองพื้นที่ของ Array ในทั้งรูปแบบ [] และคำสั่ง malloc อีกทั้งยังได้เรียนรู้การประยุกต์ใช้ array กับการวนลูป เพื่อดำเนินการต่างๆ เช่น traversal, comparison, insert, delete เป็นต้น

Lab 1 Array

- **Lab 1.1 : No Bracket**

```c
int *arr = malloc( length * sizeof(int));
for(int i=0; i<length; i++){
    scanf(" %d", arr+i);
}

int mode;
scanf(" %d", &mode);
if(mode == 1 && length<=1){
    printf("none");
}
else{
    for(int i=mode; i<length; i+=2){
        printf("%d ", *(arr+i));
    }
}

free(arr);
```

เนื่องจากโจทย์ข้อนี้ไม่ให้ใช้คำสั่งที่มี [] ดังนั้นการประกาศ array จึงต้องใช้ฟังก์ชัน malloc มาช่วยในการจองพื้นที่ในหน่วยความจำ และในการจะ access ค่าก็จะต้องคำนวณจาก address เมื่อทำการสร้าง array แล้วจึงใช้การวนลูปเพื่อแสดงค่าตัวเลข ณ ตำแหน่งนั้นๆ

- **Lab 1.2 : No Bracket No Printf**

```c
int *arr = malloc( length * sizeof(int) );
for(int i=0; i<length; i++){
    scanf(" %d", arr+i);
}
```

```c
void print(int number, int position){
    printf("%d %d\n", number, position);
}
```

โจทย์ข้อนี้ไม่ให้ใช้คำสั่งที่มี [] ดังนั้นการประกาศ array จึงต้องใช้คำสั่ง malloc มาช่วยในการจองพื้นที่ในหน่วยความจำ ในการจะ access ค่าก็จะต้องคำนวณจาก address และเนื่องจากโจทย์ไม่ให้มีคำสั่ง printf ในฟังก์ชัน main จึงต้องสร้างฟังก์ชัน print เพื่อแสดงผลทางหน้าจอ

```c
void smallest_number(int *arr , int length){
    int pos=0;
    for(int i=0; i<length; i++){
        if(*(arr+i) < *(arr+pos)){
            pos = i ;
        }
    }
    print(*(arr+pos) , pos);
}


void largest_number(int *arr , int length){
    int pos=0;
    for(int i=0; i<length; i++){
        if(*(arr+i) > *(arr+pos)){
            pos = i ;
        }
    }
    print(*(arr+pos) , pos);
}
```

ในการจะหา largest และ smallest number จะวนลูปตัวเลขทั้งหมดใน array และ comparison เพื่อหาตัวเลขมากที่สุด และน้อยที่สุด

- **Lab 1.3 : Sum of Diagonal Matrix**

```c
if(rows != columms){
    printf("ERROR");
    return 0;
}
```

```c
int matrix[rows][columms];
for(i=0; i<rows; i++){
    for(j=0; j<columms; j++){
        scanf(" %d", &matrix[i][j]);
    }
}
```

ในการหาผลรวมของ primary และ secondary diagonal matrix แถวและหลักของแมทริกซ์จะต้องเท่ากัน เมื่อตรวจสอบแล้วใช้ nested loop เพื่อวนรับค่าแมทริกซ์ในแถวและหลักนั้นๆ

```c
for(i=0; i<rows; i++){
    primary_sum += matrix[i][i];
    }
printf("Primary: %d\n", primary_sum);
```

ผลรวมของ primary diagonal matrix หาได้จากผลรวมของตัวเลขใน matrix ที่แทยงมุมจากซ้ายลงไปขวา จะสังเกตความสัมพันธ์ได้ว่า หากแถวมีค่าเท่ากับ i หลักจะมีค่าเท่ากับ i เช่นเดียวกัน จึงใช้การวนลูปเพื่อบวกค่าเข้าไปในตัวแปร primary_sum

```
while(i < rows){
    secondary_sum += matrix[i][columms-i-1];
    i++;
}
printf("Secondary: %d", secondary_sum);
```

ผลรวมของ secondary diagonal matrix หาได้จากผลรวมของตัวเลขใน matrix ที่แทยงมุมจากขวาลงไปซ้าย จะสังเกตความสัมพันธ์ได้ว่า หากแถวมีค่าเท่ากับ i หลักจะมีค่าเท่ากับจำนวนหลักของเมทริกซ์ – i – 1 จากนั้นจึงใช้การวนลูปเพื่อบวกค่าเข้าไปในตัวแปร secondary_sum

- **Lab 1.4 : Dictionary**

```
struct dict{
    char value[100];
    char key[100];
};
```

```
int size;
scanf("%d",&size);
struct dict dic[size];
```

สร้าง structure dict ที่ประกอบไปด้วย key และ value จากนั้นประกาศ array ประเภท dict

```
void editdict(struct dict *dic, int size){ //fo
    char EditKey[100], EditValue[100];
    scanf(" %s %s", EditKey, EditValue);

    for(int i=0; i<size; i++){
        if(strcmp(dic[i].key , EditKey)==0){
            strcpy(dic[i].value, EditValue);
            return;
        }
    }

    printf("No change\n");
    return;
}
```

ฟังก์ชัน editdict จะทำการรับค่าข้อมูล key และ value ที่ต้องการจะแก้ไข จากนั้นวนลูปเพื่อหา key ที่ต้องการแก้ไขใน array เมื่อเจอจะทำการอัพเดท value เก่า ให้เป็น value ใหม่

- **Challenge: List Slicing**

เนื่องจาก list slicing สามารถใส่ start step end ได้หลายกรณี จึงต้องทำการ format ก่อน

```
if(start < 0){
    start = start + size;
}
if(end < 0){
    end = end + size;
}
```

```
// for a oversize case
if(start < end){
    if(start > size || start < 0){
        start = 0;
    }
    if(end > size || end < 0){
        end = size;
    }
}
else if (start > end)
{
    if(start > size){
        start = size - 1;
    }
    if(end > size){
        end = 0;
    }
}
```

เนื่องจากค่าที่รับมามีโอกาสเป็นตัวเลขติดลบ ซึ่งค่าติดลบนั้นหมายถึงเริ่มนับจาก index สุดท้าย แต่ c compiler นั้นไม่สามารถทำงานได้ในกรณีที่ index เป็นตัวเลขลบ จึงจะต้องทำการแปลง index ให้เป็นตัวเลขบวก และเนื่องจาก start และ end ที่รับมารวมถึงที่บวก size มา มีโอกาสที่จะมากกว่า size และน้อยกว่า 0 จึงจะต้องทำการ format เพื่อให้ compiler สามารถทำงานได้

```
// decrease or increase
if(step>=0){
    for(int i=start; i<end; i+=step){
        printf("%d ", arr[i]);
        check = 1;
    }
}
else{
    for(int i=start; i>end; i+=step){
        printf("%d ", arr[i]);
        check = 1;
    }
}
```

หลังจากทำการ format เพื่อรองรับทุกกรณีแล้ว จะใช้ step เพื่อดูว่าจะต้องแสดงผลจาก index น้อยไปมาก หรือมากไปน้อย การแสดงผลใช้การวนลูป และ+- index ตามค่า step

Assignment 1 Array

- **Assignment 1.1 : Jump Game**

```
void Jump_ways(int position, int count, int array[], int length){
    if(position == length-1){
        if (count < minJump){
            minJump = count;
        }
    }
    else{
        for(int i=position+1 ; i<=position + array[position] && i<length; i++){
            Jump_ways(i, count+1, array, length);
        }
    }
}       You, last week • add Lab & Assignment Files
```

ใช้การวนรูป และ recursive เพื่อลอง handle ทุกวิธีที่เป็นไปได้ จากนั้นจะได้จำนวนครั้งที่กระโดดของแต่ละวิธี ทำการ comparison ว่าวิธีไหนกระโดดน้อยที่สุด

- **Assignment 1.2 : Symmetric Matrix**

```
int symmetric(int rows, int cols, int matrix[rows][cols]){
    for(int i=0; i<rows; i++){
        for(int j=0; j<cols; j++){
            if(matrix[i][j] != matrix[j][i]){
                return 0;
            }
        }
    }
    return 1;
}

int skew_symmetric(int rows, int cols, int matrix[rows][cols]){
    for(int i=0; i<rows; i++){
        for(int j=0; j<cols; j++){
            if(matrix[i][j] != -1*(matrix[j][i])){
                return 0;
            }
        }
    }
    return 1;
}
```

ทั้งสองฟังก์ชันใช้ nested loop เพื่อ traversal matrix ในแถว และหลัก จากนั้นระหว่างนั้นทำ comparison ตามนิยามของ symmetric matrix และ skew symmetric matrix

- **Assignment 1.3 : Spiral Array Printer**

```c
void spiral_printing(int ST_rows, int ST_cols, int rows, int cols, int matrix[rows][cols]){

    while(ST_rows < rows && ST_cols < cols){

        // printf("--------------------TO RIGHT----------------\n");

        for(int i = ST_cols; i<cols; i++){
            printf("%d ", matrix[ST_rows][i]);
        }
        ST_rows++;

        // printf("------------------DOWn----------------\n");

        for(int i = ST_rows; i< rows; i++){
            printf("%d ", matrix[i][cols-1]);
        }
        cols--;

        // printf("----------------LEFT------------------\n");

        if(ST_cols < cols){
            for(int i = cols-1; i >= ST_cols; i--){
                printf("%d ", matrix[rows-1][i]);
            }
        }
        rows--;

        // printf("------------------UP--------------------\n");

        if(ST_rows < rows){
            for(int i = rows-1; i>= ST_rows; i--){
                printf("%d ", matrix[i][ST_cols]);
        }
        }
        ST_cols++;

        // printf("check point %d %d %d %d\n",ST_rows, ST_cols, rows, cols);
    }
}
```

หลักการคือการ traversal matrix จากทาง บนซ้ายไปบนขวา บนขวาลงล่างขวา ล่างขวาไปล่างซ้าย และล่างซ้ายขึ้นบนซ้าย วนลูปแบบนี้จนกว่าจะครบทั้ง matrix

- **Assignment 1.4 : Matrix Multiplication**

```
int product_matrix[n_rows][m_columms];
if(n_columms == m_rows){
    for(int i=0; i<n_rows; i++){
        for(int j=0; j<m_columms; j++){
            product_matrix[i][j] = 0;
            for(int k=0; k<n_columms && k<m_rows; k++){
                product_matrix[i][j] += matrix_n[i][k] * matrix_m[k][j];
            }
        }
    }
}
```

ใช้ nested loop ถึง 3 ชั้น โดยชั้นที่ 1 จะเป็นการ traversal แถวของ matrix หนึ่ง ชั้นที่ 2 เป็นการ traversal หลักของ matrix สอง และชั้น 3 ชั้นในสุดเป็นการ traversal หลักของ matrix หนึ่ง และ แถวของ matrix สอง ซึ่งมีค่าเท่ากัน เพื่อหาผลลัพธ์การคูณ matrix ตามหลักการ

- **Assignment 1.5 : Grading**

```
struct student
{
    char name[100];
    float score;
};
```

สร้าง structure student ที่ประกอบไปด้วยชื่อ และคะแนน

```
float find_MEAN(struct student *student_info, int amount){
    float sum=0.00, Mean;

    for(int i=0; i<amount; i++){
        sum += student_info[i].score;
    }

    Mean = sum / (float)amount;
    return Mean;
}
```

```
float find_SD(struct student *student_info, int amount, float Mean){
    float sum=0.00, SD;

    for(int i=0; i<amount; i++){
        sum += pow((student_info[i].score - Mean), 2);
    }

    SD = sqrt(sum/amount);
    return SD;
}
```

การหาค่า mean และค่า SD จะทำการวนลูปเพื่อหาผลรวมหารด้วยจำนวนนักเรียน ตามหลักการการหาค่า mean และค่า SD โดยเรียกไลบารี่ math.h เพื่อมาช่วยในการคำนวณ

```c
void MAXIMUM_student(struct student *student_info, int amount){
    int pos_MaxScore = 0;

    for(int i=0; i<amount; i++){
        if(student_info[i].score > student_info[pos_MaxScore].score){
            pos_MaxScore = i;
        }
    }
    printf(" %s", student_info[pos_MaxScore].name);
    return;
}

void MINIMUM_student(struct student *student_info, int amount){
    int pos_MinScore = 0;

    for(int i=0; i<amount; i++){
        if(student_info[i].score < student_info[pos_MinScore].score){
            pos_MinScore = i;
        }
    }
    printf(" %s", student_info[pos_MinScore].name);
    return;
}
```

การหานักเรียนที่มีคะแนนสูงที่สุด และต่ำที่สุดก็สามารถโดยการวนลูป และ comparison คะแนนของนักเรียน และเนื่องจากมีโครงสร้างข้อมูลของนักเรียน ที่ประกอบไปด้วยชื่อ และคะแนนอยู่ ทำให้สามารถแสดงชื่อนักเรียนจากคะแนนที่สูงที่สุด และต่ำที่สุดได้

- **สิ่งที่ได้เรียนรู้จาก Lab 1 & Assignment 1**

การจองพื้นที่ของ Array ในทั้งรูปแบบ [] และคำสั่ง malloc อีกทั้งยังได้เรียนรู้การประยุกต์ใช้ array กับการ loop และ nested loop เพื่อดำเนินการต่างๆ เช่น traversal, comparison, insert, delete เป็นต้น นอกจากนี้ยังได้สร้าง structure ที่เก็บ method ต่างๆ เพื่อประยุกต์ใช้ในการเขียนโปรแกรม และได้ฝึกการเขียนโปรแกรมแบบ recursive

Lab 2 Linked List

- **Lab 2.1 : Linked List Insertion**

```c
void InsertNodeEnd(struct node **start){
    struct node *ptr;
    int val;

    scanf(" %d", &val);
    struct node *newNode = (struct node*)malloc(sizeof(struct node));
    newNode->data = val;
    newNode->next = NULL;

    // printf("%p\n" ,*start);
    ptr = *start;

    if (*start == NULL) {
        *start = newNode;
    }
    else{
        while(ptr->next != NULL){
            ptr = ptr->next;
        }
        ptr->next = newNode;
    }
}
```

```c
void InsertNodeBegin(struct node **start){
    int val;

    scanf(" %d", &val);
    struct node *newNode = (struct node*)malloc(sizeof(struct node));
    newNode->data = val;
    newNode->next = *start;

    *start = newNode;
}
```

```c
// free allocated memories
struct node *ptr, *ptrNext;
ptr = LL;

while(ptr != NULL){
    ptrNext = ptr->next;
    free(ptr);
    ptr = ptrNext;
}
```

```c
void display(struct node *start){
    struct node *ptr;
    ptr = start;

    if(ptr == NULL){
        printf("Invalid");
    }

    while(ptr != NULL){
        printf("%d ", ptr->data);
        ptr = ptr->next;
    }

}
```

- **Lab 2.2 : Before or After Insertion**

```c
void InsertNodeAfter(struct node **start , int length){
    struct node *ptr , *prePtr;
    int REFnumber, INSERTnumber, count = 0;
    scanf(" %d %d", &REFnumber, &INSERTnumber);

    struct node *newNode = (struct node*)malloc(sizeof(struct node));
    newNode->data = INSERTnumber;

    ptr = *start;
    prePtr = ptr;
    while(prePtr->data != REFnumber){
        // printf("%p:%d , count = %d\n",prePtr, prePtr->data, count);
        if (ptr == NULL){
            return;
        }
        prePtr = ptr;
        ptr = ptr->next;
        count++;
    }

    if(!count){
        newNode->next = prePtr->next;
        prePtr->next = newNode;
        return;
    }

    prePtr->next = newNode;      You, 6 days ago • add Lab & Assignment Fi
    newNode->next = ptr;
}
```

```c
void InsertNodeBefore(struct node **start , int length){
    struct node *ptr , *prePtr;
    int REFnumber, INSERTnumber, count = 0;
    scanf(" %d %d", &REFnumber, &INSERTnumber);

    struct node *newNode = (struct node*)malloc(sizeof(struct node));
    newNode->data = INSERTnumber;

    ptr = *start;
    prePtr = ptr;
    while(ptr->data != REFnumber){
        prePtr = ptr;
        ptr = ptr->next;
        count++;
        if (ptr == NULL){
            return;
        }
    }

    if(!count){
        newNode->next = *start;
        *start = newNode;
        return;
    }

    prePtr->next = newNode;
    newNode->next = ptr;
}
```

- **Lab 2.3 : Where to Delete?**

```c
void DeleteLastNode(struct node **start){
    struct node *ptr , *prePtr;
    int count = 0;

    if(*start == NULL){
        return;
    }
    ptr = *start;
    while(ptr->next != NULL){
        prePtr = ptr;
        ptr = ptr->next;
        count++;
    }

    if(!count){
        *start = NULL;
        free(ptr);
        return;
    }
    prePtr->next = NULL;
    free(ptr);
    return;
}
```

```c
void DeleteFirstNode(struct node **start){
    struct node *ptr;

    if(*start == NULL){
        return;
    }
    ptr = *start;
    *start = (*start)->next;
    free(ptr);
    return;
}
```

```c
void DeleteValueNode(struct node **start, int val){
    struct node *ptr , *prePtr;
    int count = 0;

    if(*start == NULL){
        return;
    }
    ptr = *start;
    while(ptr->data != val){
        prePtr = ptr;
        ptr = ptr->next;
        if(ptr == NULL){
            return;
        }
        count++;
        // printf("%d " , prePtr->data);
    }
    if(!count){
        DeleteFirstNode(start);
        return;
    }
    prePtr->next = ptr->next;
    free(ptr);
    return;
}
```

```c
struct node
{
    int data;
    struct node* next;
};
```

- **Lab 2.4 : Linked list that can go back**

```c
void InsertNodeEnd(struct node **start, int val){
    struct node *ptr;

    struct node *newnode = (struct node*)malloc(sizeof(struct node));
    newnode->data = val;
    newnode->prev = NULL;
    newnode->next = NULL;

    if(*start == NULL){
        *start = newnode;
        return;
    }
    ptr = *start;
    while(ptr->next != NULL){
        ptr = ptr->next;
    }
    ptr->next = newnode;
    newnode->prev = ptr;

}       You, 6 days ago • add Lab & Assignment Files
```

```c
struct node
{
    struct node* prev;
    int data;
    struct node* next;
};
```

```c
void DeleteValueNode(struct node **start, int val){
    struct node *ptr , *prePtr;
    int count = 0;

    if(*start == NULL){
        return;
    }
    ptr = *start;
    while(ptr->data != val){
        prePtr = ptr;
        ptr = ptr->next;
        if(ptr == NULL){
            return;
        }
        count++;
        // printf("%d " , prePtr->data);
    }
    if(!count){
        *start = (*start)->next;
        (*start)->prev = NULL;
        // printf("Previous %p and Next %p !!\n", (*st
        return;
    }
    if(ptr->next == NULL){
        prePtr->next = NULL;
        free(ptr);
        return;
    }
    prePtr->next = ptr->next;
    ptr->next->prev = prePtr;
    free(ptr);
    return;
}
```

```c
void SearchAroundNode(struct node **start, int val){
    struct node *ptr;
    if(*start == NULL){
        return;
    }

    ptr = *start;
    while(ptr->data != val){
        ptr = ptr->next;
        if(ptr == NULL){
            printf("none\n");
            return;
        }
    }

    if(ptr->prev != NULL || ptr->next != NULL){
        if(ptr->prev == NULL){
            printf("NULL ");
        }
        else if ((*ptr->prev).data)
        {
            printf("%d ", (*ptr->prev).data);
        }

        if(ptr->next == NULL){
            printf("NULL\n");
        }
        else if((*ptr->next).data){
            printf("%d\n", (*ptr->next).data);
        }

        return;
    }

    return;
}
```

- **Lab 2.5: Circular Linked List**

```c
void InsertLastNode(struct node **start, int val){
    struct node *ptr;

    struct node *newnode = (struct node*)malloc(sizeof(struct node));
    newnode->data = val;

    if(*start == NULL){
        *start = newnode;
        newnode->next = *start;
        return;
    }
    ptr = *start;
    while(ptr->next != *start){
        ptr = ptr->next;
    }

    ptr->next = newnode;
    newnode->next = *start;
}
```

```c
void DeleteValueNode(struct node **start, int val){
    struct node *ptr , *Preptr, *temp;

    if(*start == NULL){
        return;
    }

    ptr = *start;
    Preptr = NULL;
    while(ptr->data != val){
        Preptr = ptr;
        ptr = ptr->next;
        if(ptr == *start){
            return;
        }
    }

    // only one node deleted
    if(ptr->next == *start && Preptr == NULL){      You, last
        free(ptr);
        *start = NULL;
        return;
    }

    // first node deleted
    if(Preptr == NULL){
        ptr = *start;
        while (ptr->next != *start){
            ptr = ptr->next;
        }
        ptr->next = (*start)->next;
        ptr = *start;
        *start = (*start)->next;
        free(ptr);
        return;
    }

    // other node deleted
    Preptr->next = ptr->next;
    free(ptr);
    return;
```

(ต่อ)

(ต่อ) อธิบายรวม LAB 2 Node ที่นำมาสร้างโครงสร้างข้อมูลแบบ Singly และ Circular Linked List จะประกอบไปด้วยข้อมูล และที่อยู่ของ Node ถัดไป แต่หากเป็น Doubly Linked List จะมีที่อยู่ของ Node ก่อนหน้าด้วย

ในการดำเนินการต่างๆ ผ่าน data structure ประเภท linked list ไม่ว่าจะเป็นการ Insert , Delete หรือ Free จะมีทั้งการใช้วนลูป และไม่ใช้วนลูป ส่วนใหญ่ที่ใช้การวนลูป ก็เพื่อ traversal ไปยัง node อ้างอิง เพื่อทำการ Insert, Delete, Free หรือแสดงผล โดยการอัพเดทที่อยู่ของ node ต่างๆ ใน Linked list ซึ่งที่อยู่ที่ต้องอัพเดทก็จะต่างกันไปขึ้นอยู่กับประเภทของ Linked list

Assignment 2 Linked List

- **Assignment 2.1 : It's Sorting Time**

```c
void SortByScore(struct node **start, int size){
    struct node *ptr;
    int tempScore , tempID;

    for (int i = 0; i < size - 1; i++) {
        ptr = *start;

        while (ptr->next != NULL) {
            if (ptr->score > ptr->next->score) {
                tempID = ptr->ID;
                tempScore = ptr->score;

                ptr->ID = ptr->next->ID;
                ptr->score = ptr->next->score;

                ptr->next->ID = tempID;
                ptr->next->score = tempScore;
            }
            ptr = ptr->next;
        }
    }

    return;
}
```

```c
void SortByID(struct node **start, int size){
    struct node *ptr;
    int tempScore , tempID;

    for (int i = 0; i < size - 1; i++){
        ptr = *start;

        while (ptr->next != NULL) {
            if (ptr->ID > ptr->next->ID) {
                tempID = ptr->ID;
                tempScore = ptr->score;

                ptr->ID = ptr->next->ID;
                ptr->score = ptr->next->score;

                ptr->next->ID = tempID;
                ptr->next->score = tempScore;
            }
            ptr = ptr->next;
        }

    }

    return;
}
```

ประยุกต์ใช้ traversal Linked list และ comparison เพื่อสลับข้อมูลระหว่าง node

- **Assignment 2.2 : Circular table**

```c
int DeleteNodeByStep(struct node **start, int step){
    struct node *ptr , *Preptr, *temp;
    int count = 0;

    if(*start == NULL){
        return 0;
    }

    ptr = *start;
    Preptr = NULL;
    while(ptr != ptr->next){
        count++;


        if(count % step == 0){
            Preptr->next = ptr->next;
            temp = ptr;
            ptr = ptr->next;
            free(temp);
        }
        else{
            Preptr = ptr;
            ptr = ptr->next;
        }
    }

    return ptr->data;
}
```

ประยุกต์ใช้ traversal Circular Linked list มาดำเนินการลบ node ที่ตำแหน่งต่างๆ ตามค่า step จนเหลือ node ตัวเดียว

- **Assignment 2.3 : Reverse Linked List**

```c
void ReverseLL(struct node **start, int init, int last){
    struct node *ptr, *PrePtr, *nextnode, *tmp;

    ptr = *start;
    for(int i = 0; i<init-1; i++){
        PrePtr = ptr;
        ptr = ptr->next;
        if(ptr->next == NULL){
            ptr = *start;
            break;
        }
    }
    // printf("%d ", ptr->data);
    nextnode = ptr->next;
    // printf("%d\n", nextnode->data);

    for(int i=init; i<last; i++){
        tmp = nextnode->next;
        nextnode->next = ptr;
        ptr = nextnode;
        nextnode = tmp;
    }
    if(init == 1){
        (*start)->next = nextnode;
        // printf("%d\n", ptr->data);
        *start = ptr;
    }
    else{
        PrePtr->next->next = nextnode;
        PrePtr->next = ptr;
    }

}
```

ประยุกต์ใช้ traverasal Doubly Linked list จากนั้นทำการอัพเดทที่อยู่ของ node แต่ละ node เพื่อให้ได้ค่าที่ย้อนกลับ

- **สิ่งที่ได้เรียนรู้จาก Lab 2 & Assignment 2**

ได้เรียนรู้หลักการ และประเภท ว่าแต่ละประเภทของ Linked list นั้นมี method ใดบ้าง นอกจากนี้ยังได้เรียนรู้เกี่ยวกับการดำเนินการต่างๆของ Linked list ไม่ว่าจะเป็น traversal, insert, delete, free, display หรือ update ล้วนแล้วแต่เป็นการประยุกต์การวนลูป และใน assignment ได้ประยุกต์ใช้ Linked list กับการเก็บข้อมูล ID, SCORE และการดำเนินการอื่นอย่าง reverse และ swap data

Lab 3 Stack

- **Lab 3.1 : Stack Array**

```c
void push(int arr[], int val, int* TOP){
    (*TOP)++;
    arr[*TOP] = val;
    return;
}

void pop(int arr[], int* TOP){
    (*TOP)--;
    return;
}

void show(int arr[], int TOP){
    for(int i = TOP; i>-1; i--){
        printf("%d\n", arr[i]);
    }
    return;
}

bool isFull(int SIZE, int TOP){
    if(TOP == SIZE - 1){return true;}
    else{return false;}
}

bool isEmpty(int TOP){
    if(TOP == -1){return true;}
    else{return false;}
}
```

การสร้างโครงสร้างข้อมูลแบบ stack ด้วย array จะมี pointer ที่จะคอยชี้ตำแหน่งใน array เพื่อดำเนินการต่างๆ

- **Lab 3.2 : Stack as linked list**

```c
void push(struct node **top , int val){
    struct node *newnode = (struct node*)malloc(sizeof(struct node));

    newnode->data = val;
    if(isEmpty(*top)){
        newnode->next = NULL;
        *top = newnode;
    }
    else{
        newnode->next = *top;
        *top = newnode;
    }
}
```

```c
int pop(struct node **top){
    struct node *ptr;
    int tmp_data;

    if(isEmpty(*top)){
        return 0;
    }
    else{
        ptr = *top;
        *top = (*top)->next;
        tmp_data = ptr->data;
        free(ptr);
        return tmp_data;
    }
}
```

การสร้างโครงสร้างข้อมูลแบบ stack ด้วย linked list จะมีการใช้ head หรือ top ที่จะคอยชี้ตำแหน่งใน เพื่อดำเนินการต่างๆ เช่น การ push ให้เทียบปกติก็คือการ InsertNodeBegin ของ linked list และ การ pop คือการ DeleteFirstNode ของ linked list ตามนิยาม First In Last Out

- **Lab 3.3 : Tower of Hanoi**

```c
void TowerOfHanoI(int N, char source, char dest, char aux){
    if( N==1 ){
        printf("Move disk %d from %c to %c\n", N, source, dest);
        return ;
    }

    TowerOfHanoI(N-1, source, aux, dest);
    printf("Move disk %d from %c to %c\n", N, source, dest);
    TowerOfHanoI(N-1, aux, dest, source);
}
```

ประยุกต์ใช้ recursive ในรูปแบบของ stack เพื่อทำการย้าย disk ที่ n และ n-1

Assignment 3 Stack

Assignment 3.1 : Ten to X

```c
if(base < 2 || base > 36){ printf("invalid\n"); }
else{
    while(num != 0){
        ConvertBASE(&LL ,num, base);
        num /= base;
    }
    display(LL);
}
```

```c
void ConvertBASE(struct node **top, int num, int base){
    struct node ptr;

    struct node *newnode = (struct node*)malloc(sizeof(struct node));
    newnode->data = num%base;

    if(*top == NULL){
        newnode->next = NULL;
        *top = newnode;
    }
    else{
        newnode->next = *top;
        *top = newnode;
    }
}
```

หารเอาเศษตัวเลขที่รับเข้าไป ด้วยฐานที่ต้องการจะแปลงเป็น จากนั้นเก็บเศษลง stack เมื่อหารจนหารต่อไม่ได้ ให้แสดงผลข้อมูลที่อยู่ใน stack ตามหลักการ First In Last Out

- **Assignment 3.2 : Palindrome checker**

```
bool IsPalindrome(char input[]){
    struct Stack String;
    String.TOP = -1;
    int length = strlen(input);

    for(int i=0; i<length; i++){
        push(&String, input[i]);
    }

    for(int i=0; i<length; i++){
        if(input[i] != String.data[String.TOP]){
            return false;
        }
        String.TOP--;
    }
    return true;
}
```

สร้าง stack จาก array ของข้อมูลที่นำเข้า จากนั้น วนลูป comparison ระหว่าง array และ stack โดยตามหลักการ First In Last Out หมายความว่า ตัวแรกของ array จะตรวจสอบกับ ตัวสุดท้ายของ stack

- **Assignment 3.3 : Parenthesis Checker**

```
int IsBalance(char input[]){
    struct Stack OpenBlk;
    OpenBlk.TOP = -1;
    int length = strlen(input);

    for(int i = 0; i<length; i++){
        if(input[i] == '[' || input[i] == '(' || input[i] == '{'){
            push(&OpenBlk, input[i]);
        }
        else if (input[i] == ']' || input[i] == ')' || input[i] == '}')
        {
            if(!IsDual(OpenBlk.data[OpenBlk.TOP],input[i]) || IsEmpty(OpenBlk)){
                return 0;
            }
            pop(&OpenBlk);
        }
    }

    return IsEmpty(OpenBlk);
}
```

วนลูปตรวจสอบข้อมูลนำเข้า หากเป็นวงเล็บเปิด push ค่าเข้า stack หากเป็นวงเล็บปิดจะทำการ pop ค่าออกจาก stack และสุดท้ายจะดูว่า stack ว่างหรือไม่

- **Assignment 3.4 : Infix to Postfix**

```
for(int iIF = 0; iIF < strlen(Infix); iIF++){
    char buff = Infix[iIF];

    // printf("%c\n", buff);

    if(isOperand(buff)){
        Postfix[iPF++] = buff;
    }
    else if(buff == '('){
        push(&operator, buff);
    }
    else if(buff == ')'){
        while(operator.data[operator.TOP] != '(' && !IsEmpty(operator)){
            Postfix[iPF++] = pop(&operator);
        }
        pop(&operator); //pop '('
    }
    else{
        while(ItsPrecudure(buff) <= ItsPrecudure(operator.data[operator.TOP]) && !IsEmpty(operator)){
            if(buff == '^' && operator.data[operator.TOP] == '^'){
                break;
            }
            else{
                Postfix[iPF++] = pop(&operator);
            }
        }
        push(&operator , buff);
        // printf("Operator push\n");
    }
}
```

สร้าง stack เพื่อประยุกต์ใช้เก็บตัวดำเนินการ และตัวถูกดำเนินการ จากนั้นแบ่งออกเป็นกรณี วนลูปไปจนรับข้อมูลครบทุกตัว

- **สิ่งที่ได้เรียนรู้จาก Lab 3 & Assignment 3**

ได้เรียนรู้หลักการ และการ implement โครงสร้างข้อมูลแบบ stack ทั้งในรูปแบบ array และ Linked list อีกทั้งยังได้เรียนรู้การดำเนินการณ์ต่างๆ เช่น push, pop หรือ show เป็นต้น และได้ประยุกต์การนำ stack มาใช้ทั้งการเขียนโปรแกรมในรูปแบบ recursive หรือการนำ stack มาใช้เพื่อจัดเก็บข้อมูลที่จะต้องอาศัยหลักการ First In Last Out

Lab 4 Queue

- **Lab 4.1 : Spotify**

```
struct Queue
{
    struct MSnode *front;
    struct MSnode *rear;
};
```

โครงสร้างข้อมูลแบบ queue จะมี pointer สองตัวเพื่อมาชี้หัวและท้ายของ queue เพื่อให้สามารถทำงานได้ตามหลัก First In First Out

```
void addTo(struct Queue **playlist){

    struct MSnode *newsong;
    newsong = (struct MSnode*)malloc(sizeof(struct MSnode));
    scanf(" %[^\n]s", newsong->name);
    scanf(" %[^\n]s", newsong->artist);
    scanf(" %d", &newsong->time);
    newsong->next = NULL;

    if ((*playlist)->front == NULL)
    {
        (*playlist)->front = newsong;
        (*playlist)->rear = newsong;
        (*playlist)->front->next = (*playlist)->rear->next = NULL;
        return ;
    }
    else
    {
        (*playlist)->rear->next = newsong;
        (*playlist)->rear = newsong;
        (*playlist)->rear->next = NULL;
    }
}
```

addTo หรือ enqueue ถ้าให้เปรียบเทียบปกติก็คือการ InsertNodeEnd ของ Linked list ที่ไม่ต้อง traversal ไปที่ node สุดท้าย เนื่องจากมี pointer rear ที่จะขยับเสมอ เมื่อมี node ใหม่เข้ามา

```
(*playlist)->front = (*playlist)->front->next;
printf("Now playing: %s by %s\n", ptr->name, ptr->artist);
free(ptr);
```

เป็นการลบข้อมูลออกมาจาก queue เพื่อมาแสดงตามหลัก First In First Out

```
while(ptr != (*playlist)->rear->next)
{
    printf("%s by %s %d\n", ptr->name, ptr->artist, ptr->time);
    TotalTime += ptr->time;
    ptr = ptr->next;
}
printf("Remaining Time: %d\n", TotalTime);
```

เป็นการ display ข้อมูลที่อยู่ใน queue หากเป็นข้อนี้จะรวมเวลาทั้งหมดจากทุกเพลงเพื่อแสดง

- **Lab 4.2 : เมื่อไหร่จะถึงคิวฉันบ้าง???**

```c
void insert(int q[] , int size){

    int val;
    scanf(" %d", &val);

    if(IsFull(q, size))
    {
        printf("Queue is full!!\n");
        return ;
    }
    else if(IsEmpty(q))
    {
        front = rear = 0;
        q[rear] = val;
    }
    else if(rear == size-1 && front != 0)
    {
        rear = 0;
        q[rear] = val;
    }
    else{
        q[++rear] = val;
    }

    return ;
}
```

```c
void del(int q[], int size){

    int val;
    if(IsEmpty(q))
    {
        printf("Queue is empty!!\n");
        return ;
    }
    printf("%d\n",q[front]);
    if(front == rear)
    {
        front = rear = -1;
    }
    else
    {
        if(front == size -1){ front = 0; }
        else{ front++; }
    }

    return ;
}
```

Circular queue การ enqueue และ dequeue คิวประเภทนี้จะมีเงื่อนไขที่เยอะกว่าคิวธรรมดา แต่ยังคงใช้ concept เดิมคือ มี pointer สองตัวเพื่อมาชี้หัวและท้ายของ queue เพื่อให้สามารถทำงานได้ตามหลัก First In First Out

- **Lab 4.3 : Secret Code Only You and I Know**

```c
void enqueue(struct node **start , char input){

    struct node *ptr;
    struct node *newnode = (struct node*)malloc(sizeof(struct node));
    newnode->data = input;
    newnode->priority = GetPriorityOF(input);

    if(*start == NULL || newnode->priority > (*start)->priority)
    {
        newnode->next = *start;
        *start = newnode;
    }
    else
    {
        ptr = *start;
        while(ptr->next != NULL && ptr->next->priority >= newnode->priority)
        {
            ptr = ptr->next;
        }
        newnode->next = ptr->next;
        ptr->next = newnode;
    }

    return ;
}
```

Priority queue ในการจะ enqueue จะต้องตรวจสอบ priority ของข้อมูลใหม่เทียบกับ ข้อมูลเก่าเพื่อจะหาตำแหน่งที่จะ enqueue เข้าไป

- **สิ่งที่ได้เรียนรู้จาก Lab 4**

ได้เรียนรู้หลักการ และการ implement โครงสร้างข้อมูลแบบ queue ทั้งในรูปแบบ array และ Linked list อีกทั้งยังได้เรียนรู้การดำเนินการณ์ต่างๆ เช่น enqueue, dequeue หรือ display เป็นต้น และได้ประยุกต์การนำเอา circular queue และ priority queue มาใช้กับโจทย์ในการแปลงรหัส ตามหลักการ First In First Out