Linked List (ลิงก์ลิสต์)

โครงสร้างข้อมูลเชิงเส้นที่มีการเชื่อมโยงกันด้วยโหนด

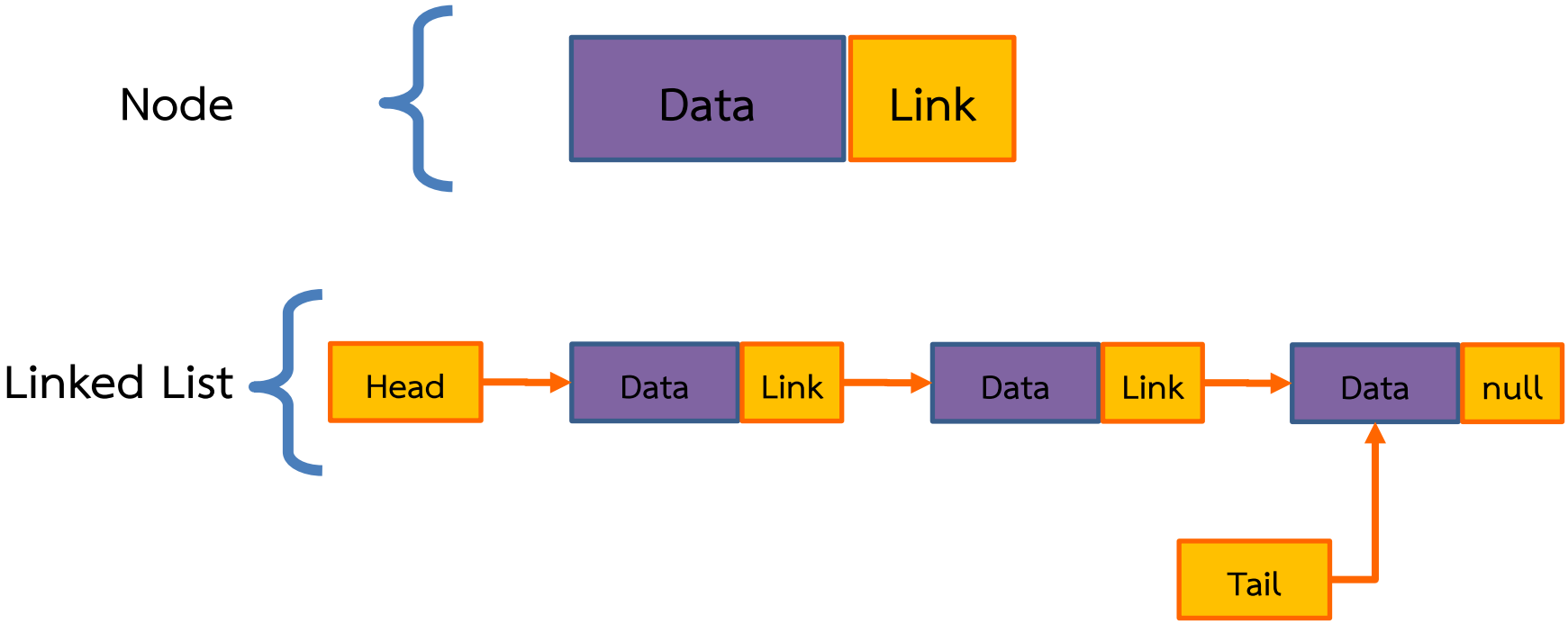
Linked List คืออะไร?

Linked List คือโครงสร้างข้อมูลชนิดหนึ่งที่ประกอบด้วยโหนด (node) หลาย ๆ ตัวเชื่อมต่อกัน โดยแต่ละโหนดจะเก็บข้อมูลและมีตัวชี้ (pointer) ที่ชี้ไปยังโหนดถัดไป ข้อดีของ Linked List คือสามารถจัดการหน่วยความจำได้ยืดหยุ่นมากกว่าการใช้ Array ซึ่งมีขนาดคงที่ จะประกอบไปด้วย 2 ส่วนหลัก

ข้อมูล (Data): เก็บข้อมูลของโหนดนั้นๆ

ตัวชี้ (Pointer): เก็บตำแหน่งหน่วยความจำของโหนดถัดไป

ภาพลิงก์ลิสต์ (Linked List)



หมายเหตุ

= ข้อมูล

= ตัวชี้ (Pointer)

ประเภทของลิงก์ลิสต์

- Single-linked list
- Double linked list
- Circular linked list

Single-linked list

- แต่ละโหนดมีข้อมูลและตัวชี้ไปยังโหนดถัดไป

ประกาศตัวแปรที่มีโครงสร้างแบบลิงค์ลิสต์เดี่ยว

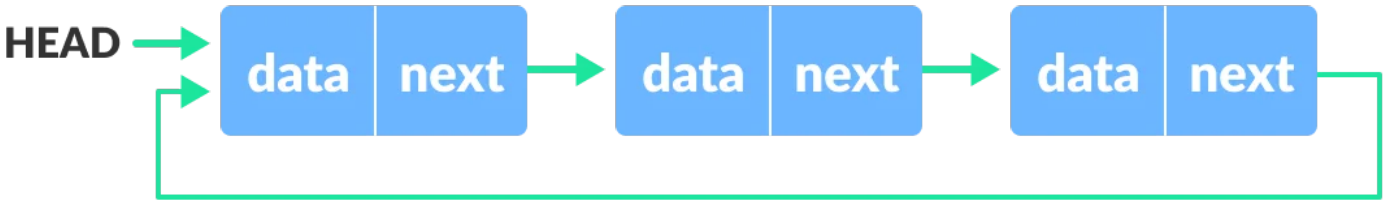
```
#include <stdio.h>
    struct node
    {
        int data;
        node *next;
    }
        node *head;
```

Double linked list

- เพิ่มตัวชี้ไปยังโหนดก่อนหน้าภายในลิงก์ลิสต์
- ดังนั้นจึงสามารถไปในทิศทางใดทิศทางหนึ่งได้ คือ เดินหน้าหรือถอยหลัง (forward or backward)

Circular linked list

- รายการลิงก์ลิสต์แบบวงกลมสามารถใช้ Single-linked list หรือ Double linked list ก็ได้
- สำหรับ Single-linked list ตัวชี้ถัดไปของโหนดสุดท้ายจะชี้ไปที่โหนดแรก
- สำหรับ Double linked list ตัวชี้ก่อนหน้าของโหนดแรกจะชี้ไปยังโหนดสุดท้ายด้วย



ข้อควรจำเกี่ยวกับลิงก์ลิสต์

- ตัวชี้ head ชี้ไปยังโหนดแรกของลิงก์ลิตส์
- ตัวชี้ next ของโหนดสุดท้ายมีค่าเป็น NULL
  - ดังนั้น ถ้า next ของโหนดปัจจุบันมีค่าเป็น NULL แปลว่าเรามาถึงโหนดสุดท้ายแล้ว

การดำเนินการเกี่ยวกับลิงก์ลิสต์

- Traversal - การเดินทางผ่านโหนดทั้งหมดในลิสต์ตั้งแต่โหนดแรก (Head) ไปจนถึงโหนดสุดท้าย (Tail)
- Insertion - การเพิ่มโหนดใหม่เข้าไปในลิสต์
- Deletion - การลบโหนดใด ๆ ออกจากลิสต์
- Search - การค้นหาโหนดที่มีข้อมูลตามค่าที่ต้องการ ใน Linked List
- Sort -เป็นการจัดเรียงโหนดในลิสต์ตามลำดับของข้อมูลภายในโหนด

Traverse a Linked List

- แสดงข้อมูลแต่ละโหนดในลิงก์ลิสต์ วิธีการคือ ใช้ตัวชี้ temp เลื่อนไปทีละโหนดจนกว่าจะสิ้นสุดลิงก์ลิสต์

1. กำหนดตัวชี้ temp เริ่มต้นจาก head
2. วนลูปจนว่า temp จะมีค่า NULL
    1. แสดงค่าข้อมูล (data)
    2. เลื่อน temp ไปยังโหนดถัดไป

```
void print(){
   Node *temp = head;
   printf("List elements \n");
   while(temp != NULL) {
      printf("%d ->",temp->data);
      temp = temp->next;
   }
   printf("NULL\n");
}
```

Insert Elements to a Linked List

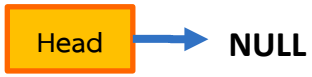
- 1. Insert at the beginning
- 2. Insert at the End
- 3. Insert at the Middle

1. Insert at the beginning

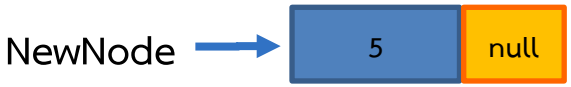
- จองหน่วยความจำสำหรับโหนดใหม่และเก็บข้อมูล
- กำหนด next ของโหนดใหม่ไปยัง head
- เปลี่ยน head ชี้ไปยังโหนดใหม่

```
void insertHead(int data){
    Node *newNode;
    newNode = malloc(sizeof(Node));
    newNode->data = data;

    newNode->next = head;
    head = newNode;
}
```

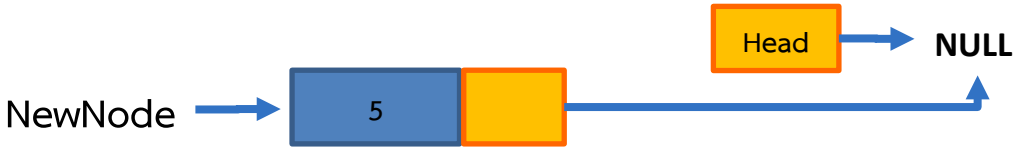
- ตัวอย่างลิงก์ลิสต์



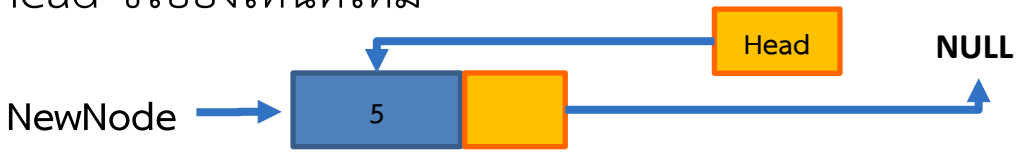
- สร้างโหนดใหม่



- กำหนด next ของโหนดไปยังตำแหน่ง head

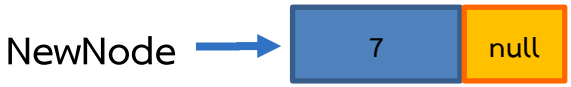


- เปลี่ยน head ชี้ไปยังโหนดใหม่



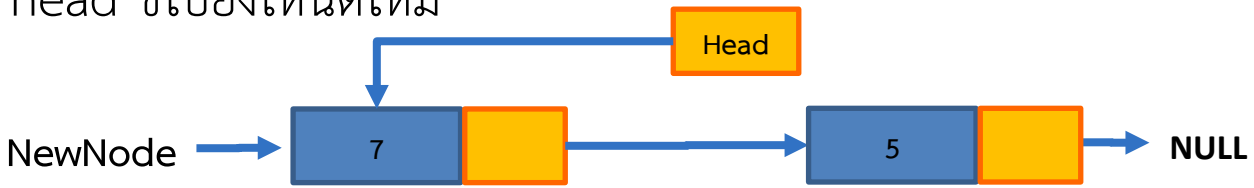
- ตัวอย่างลิงก์ลิสต์

- สร้างโหนดใหม่



- กำหนด next ของโหนดไปยังตำแหน่ง head

- เปลี่ยน head ชี้ไปยังโหนดใหม่



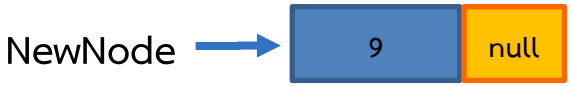
2. Insert at the End

- จองหน่วยความจำสำหรับโหนดใหม่และเก็บข้อมูล
- Traverse ไปยังโหนดสุดท้าย
- กำหนด next ของโหนดสุดท้ายให้ชี้มายังโหนดใหม่

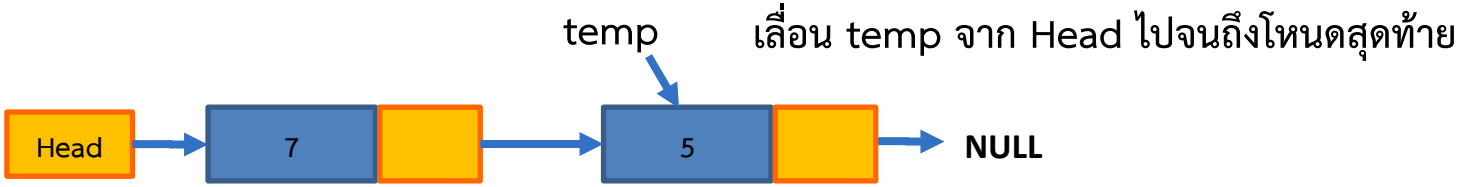
```
void insertEnd(int data){
    Node *newNode;
    newNode = malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;


    if(head == NULL){
        head = newNode;
    }else{
        Node *temp = head;
        while(temp->next != NULL){
            temp = temp->next;
        }
    temp->next = newNode;
    }
}
```

- ตัวอย่างลิงก์ลิสต์

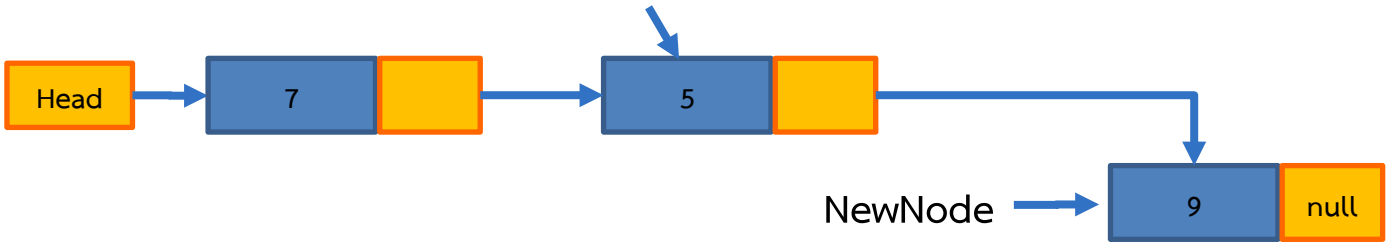
- สร้างโหนดใหม่



- Traverse ไปยังโหนดสุดท้าย



- กำหนด next ของโหนดสุดท้ายให้ชี้มายังโหนดใหม่

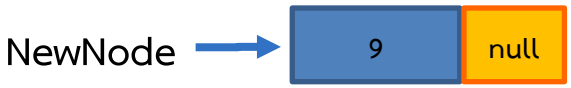


3. Insert at the Middle

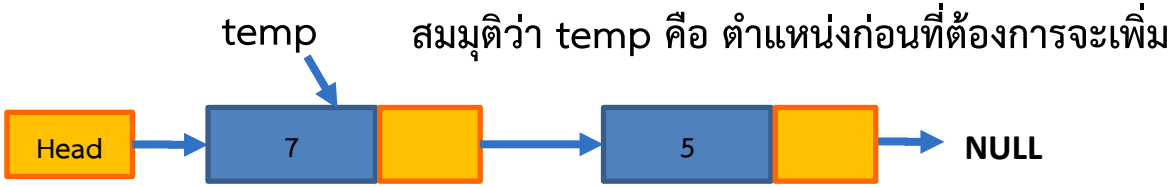
- จองหน่วยความจำสำหรับโหนดใหม่และเก็บข้อมูล
- Traverse ไปยังโหนดที่อยู่ก่อนตำแหน่งที่ต้องการเพิ่ม
- เพิ่มโหนดใหม่
  - Next ของโหนดใหม่ ชี้ไปยัง ตำแหน่งเดียวกับ next ของโหนดก่อน
  - Next ของโหนดก่อน ชี้ไปยัง โหนดใหม่

- ตัวอย่างลิงก์ลิสต์

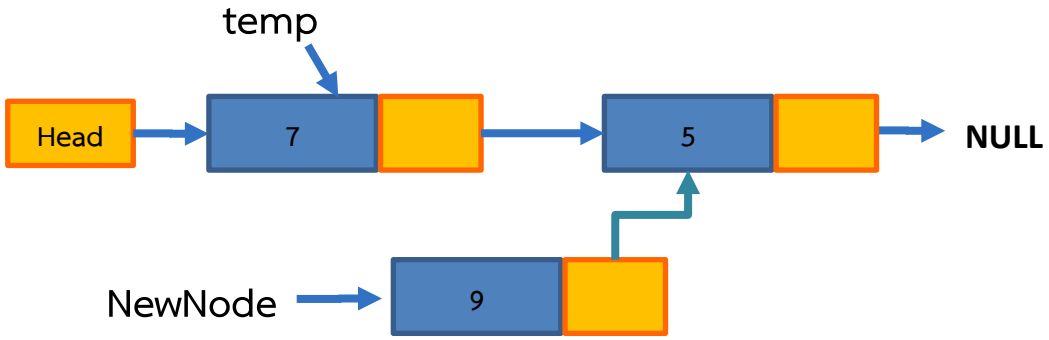
- สร้างโหนดใหม่



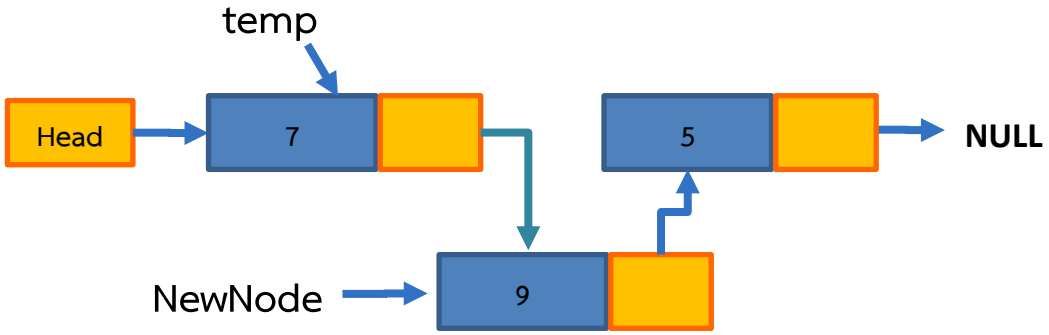
- Traverse ไปยังโหนดที่อยู่ก่อนตำแหน่งที่ต้องการเพิ่ม



- เพิ่มโหนดใหม่
  - Next ของโหนดใหม่ ชี้ไปยัง ตำแหน่งเดียวกับ next ของโหนดก่อน



  - Next ของโหนดก่อน ชี้ไปยัง โหนดใหม่



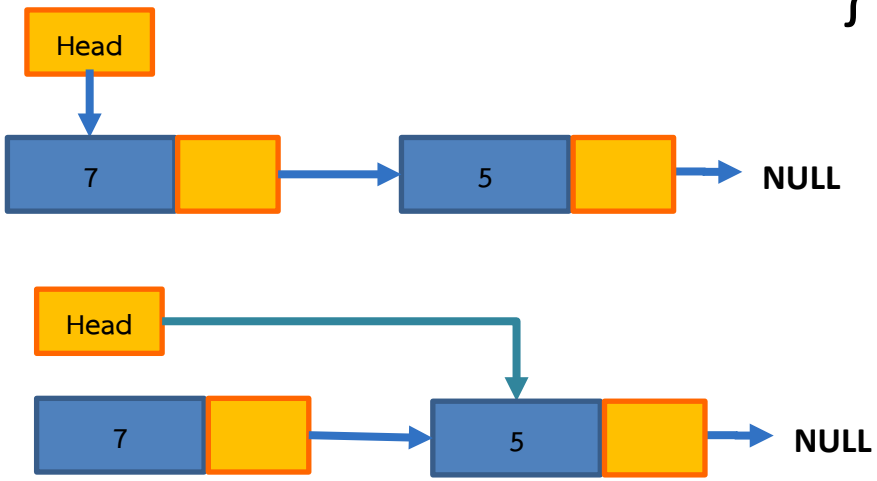
3. Insert at the Middle

- จองหน่วยความจำสำหรับโหนดใหม่และเก็บข้อมูล
- Traverse ไปยังโหนดที่อยู่ก่อนตำแหน่งที่ต้องการเพิ่ม
- เพิ่มโหนดใหม่
  - Next ของโหนดใหม่ ชี้ไปยัง ตำแหน่งเดียวกับ next ของโหนดก่อน
  - Next ของโหนดก่อน ชี้ไปยัง โหนดใหม่

```
void insertAt(Node *pre, int data){
    Node *newNode;
    newNode = malloc(sizeof(Node));
    newNode->data = data;

    newNode->next = pre->next;
    pre->next = newNode;

}
```

Delete from a Linked List

- 1. Delete from beginning
  - เปลี่ยน head ไปยังโหนดที่สอง

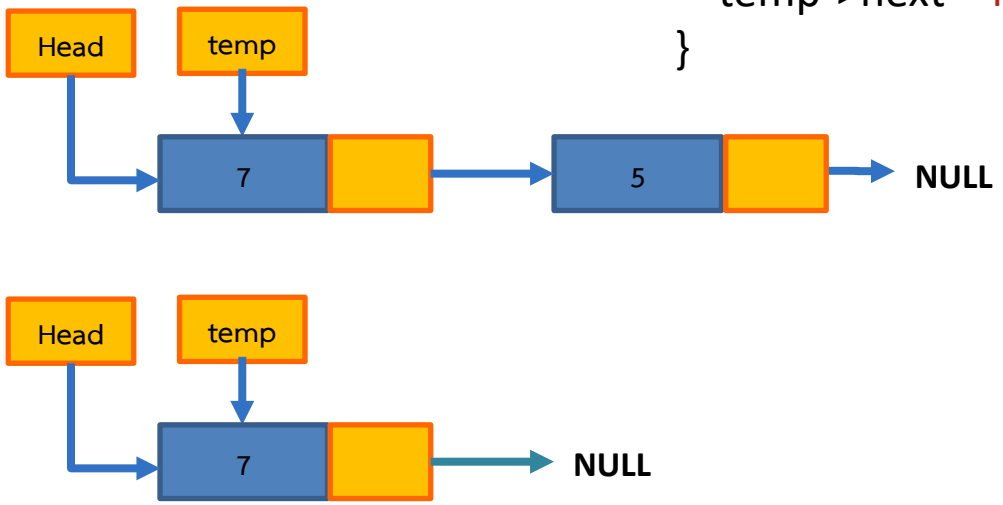



```
void deleteHead(){
    head = head->next;
}
```

Delete from a Linked List

- 2. Delete from end
  - Traverse ไปยังโหนดรองสุดท้าย
  - เปลี่ยนให้ next ของโหนดนั้นมีค่า NULL

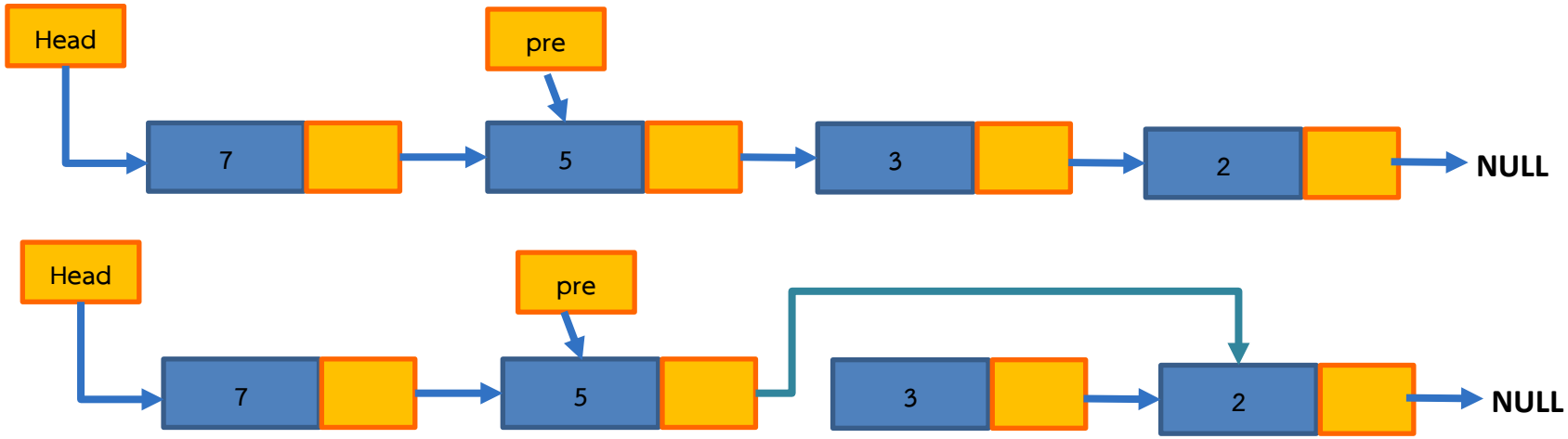
```
void deleteEnd(){
   Node* temp = head;
   while(temp->next->next != NULL){
      temp = temp->next;
   }
   temp->next = NULL;
}
```



Delete from a Linked List

- 3. Delete from middle
  - Traverse ไปยังโหนดก่อนตำแหน่งที่ต้องการลบ
  - เปลี่ยนตัวชี้ next นั้นให้ชี้ข้ามไปยังสองโหนดถัดไป

```
void deleteMiddle(Node *pre){
   pre->next = pre->next->next;
}
```



```c
#include <stdio.h>
#include <stdlib.h>

// โครงสร้างของโหนดใน Linked List
struct Node {
    int data;
    struct Node* next;
};

// ฟังก์ชันสำหรับแสดงข้อมูลใน Linked List
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// ฟังก์ชันการลบที่จุดเริ่มต้น
void deleteAtBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("ลิสต์ว่าง\n");
        return;
    }
    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}
// ฟังก์ชันการลบโหนดที่มีค่าข้อมูลตรงตามที่กำหนด
void deleteNode(struct Node** head, int key) {
    struct Node* temp = *head;
    struct Node* prev = NULL;

    if (temp != NULL && temp->data == key) {
        *head = temp->next;
        free(temp);
        return;
    }
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) return;

    prev->next = temp->next;
    free(temp);
}

// ฟังก์ชันการลบที่จุดท้ายสุด
void deleteAtEnd(struct Node** head) {
    if (*head == NULL) return;

    struct Node* temp = *head;
    if (temp->next == NULL) {
        free(temp);
        *head = NULL;
        return;
    }

    struct Node* prev = NULL;
    while (temp->next != NULL) {
        prev = temp;
        temp = temp->next;
    }
    prev->next = NULL;
    free(temp);
}

// ฟังก์ชันสร้างโหนดใหม่
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

int main() {
    struct Node* head = NULL;

    // สร้างโหนดและเชื่อมโยงกัน
    head = createNode(10);
    head->next = createNode(20);
    head->next->next = createNode(30);
    head->next->next->next = createNode(40);

    printf("ก่อนลบข้อมูล: ");
    printList(head);

    // ลบโหนดต่าง ๆ
    deleteAtBeginning(&head);  // ลบโหนดแรก
    printf("หลังลบโหนดแรก: ");
    printList(head);

    deleteNode(&head, 30);    // ลบโหนดที่มีข้อมูลเป็น 30
    printf("หลังลบโหนดที่มีค่า 30: ");
    printList(head);

    deleteAtEnd(&head);       // ลบโหนดสุดท้าย
    printf("หลังลบโหนดสุดท้าย: ");
    printList(head);

    return 0;
}
```

ก่อนลบข้อมูล: 10 -> 20 -> 30 -> 40 -> NULL

หลังลบโหนดแรก: 20 -> 30 -> 40 -> NULL

หลังลบโหนดที่มีค่า 30: 20 -> 40 -> NULL

หลังลบโหนดสุดท้าย: 20 -> NULL

Search an Element on a Linked List

- สร้างตัวชี้ current ให้มีตำแหน่งเดียวกับ head
- วนลูปจนกว่า current เป็น NULL (ไปถึงโหนดสุดท้าย)
  - ตรวจว่าข้อมูลที่ในแต่ละโหนดนั้น (current) ตรงกับสิ่งที่สนใจ
    - ถ้าใช่ พบข้อมูล (คืนค่าจริง)
  - เลื่อนตัวชี้ current ไปยังโหนดถัดไป
- คืนค่าเท็จ

```
bool search(int key) {
   Node* current = head;

   while (current != NULL) {
      if (current->data == key){
         return true;
      }
      current = current->next;
   }
   return false;
}
```

Bubble Sort

การเรียงลำดับที่ง่ายที่สุด ซึ่งทำงานโดยการเปรียบเทียบสลับตำแหน่งหากตัวซ้ายมีค่ามากกว่าตัวขวา ให้ทำการสลับตำแหน่ง

```c
#include <stdio.h>
// ฟังก์ชันสำหรับสลับค่าของตัวแปรสองตัว
void swap(int *xp, int *yp) {
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// ฟังก์ชัน Bubble Sort
void bubbleSort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n-1; i++) {
        // Traverse ลิสต์จากตำแหน่งแรกถึงตำแหน่งสุดท้าย - i
        for (j = 0; j < n-i-1; j++) {
            // ถ้าตัวที่อยู่หน้ามีค่ามากกว่า ให้สลับที่
            if (arr[j] > arr[j+1]) {
                swap(&arr[j], &arr[j+1]);
            }
        }
    }
}

// ฟังก์ชันแสดงลิสต์ข้อมูล
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("ลิสต์ก่อนการเรียงลำดับ: \n");
    printArray(arr, n);

    bubbleSort(arr, n);

    printf("ลิสต์หลังการเรียงลำดับ: \n");
    printArray(arr, n);

    return 0;
}
```

ผลลัพธ์

ลิสต์ก่อนการเรียงลำดับ:

64 34 25 12 22 11 90

ลิสต์หลังการเรียงลำดับ:

11 12 22 25 34 64 90

Sort an Element on a Linked List

- ตัวอย่างนี้ใช้ Bubble Sort ในการเรียงลำดับข้อมูลในลิงก์ลิสต์จากน้อยไปมาก

1. สร้างตัวชี้ current ให้มีตำแหน่งเดียวกับ head และสร้างตัวชี้ index = NULL
2. ถ้า head เป็น NULL
    1. คืนค่า
3. มิฉะนั้น วนลูปจนกว่าจะถึงโหนดสุดท้าย
    1. ตัวชี้ index ชี้ไปยังโหนดถัดไป
    2. ถ้าค่าในโหนดปัจจุบัน (current) มากกว่า ค่าในโหนดถัดไป
        1. สลับค่าระหว่างโหนด currrent และโหนด index

```
void sortLinkedList() {
   Node *current = head;
   Node *index = NULL;
   int temp;

   if (head == NULL) {
      return;
   } else {
      while (current != NULL) {
         // index points to the node next to current
         index = current->next;
         while (index != NULL) {
            if (current->data > index->data) {
               temp = current->data;
               current->data = index->data;
               index->data = temp;
            }
            index = index->next;
         }
         current = current->next;
      }
   }
}
```

ตัวอย่างโจทย์ในการใช้ link list

- **โจทย์ที่ 1: การจัดการรายการสินค้า**

  โจทย์: สร้างระบบสำหรับจัดการรายการสินค้าในร้านค้า โดยมีฟังก์ชันในการเพิ่มรายการสินค้า ลบรายการสินค้า และแสดงรายการสินค้าทั้งหมด

  ข้อมูลของสินค้า ประกอบด้วย ชื่อสินค้า และราคาสินค้า ให้แสดงรายการสินค้าในรูปแบบของ Linked List

- **โจทย์ที่ 2: การจัดคิว**

  โจทย์: สร้างระบบการจัดคิวสำหรับลูกค้าในร้านกาแฟ โดยแต่ละลูกค้าจะถูกเพิ่มเข้าไปในคิวเมื่อมาถึง และสามารถลบลูกค้าที่อยู่ในคิวได้เมื่อได้รับบริการ

  ข้อมูลของลูกค้า ประกอบด้วย ชื่อ และหมายเลขโทรศัพท์ ให้แสดงรายชื่อลูกค้าในคิว

- **โจทย์ที่ 3: การเก็บคะแนนสอบ**

  โจทย์: สร้างระบบสำหรับเก็บคะแนนสอบของนักเรียน โดยสามารถเพิ่มคะแนนใหม่ ลบคะแนนที่ไม่ต้องการ และแสดงคะแนนทั้งหมดของนักเรียน

  ข้อมูลของคะแนนสอบ ประกอบด้วย รหัสนักเรียน และคะแนน คำนวณคะแนนเฉลี่ยของนักเรียน

- **โจทย์ที่ 4: ระบบบันทึกประวัติการเข้าใช้**

  โจทย์: สร้างระบบบันทึกประวัติการเข้าใช้ของผู้ใช้ในแอพพลิเคชัน โดยสามารถเพิ่มข้อมูลการเข้าใช้ใหม่ ลบข้อมูลการเข้าใช้ที่ไม่ต้องการ และแสดงประวัติการเข้าใช้ทั้งหมด

  ข้อมูลการเข้าใช้ ประกอบด้วย วันและเวลาที่เข้าใช้ สามารถแสดงประวัติการเข้าใช้ในรูปแบบลำดับเวลา

- **โจทย์ที่ 5: การสร้างเกมง่าย ๆ**

  โจทย์: สร้างเกมแบบทายคำ โดยให้ผู้เล่นสามารถเพิ่มคำที่ต้องทายและแสดงคำทั้งหมดใน Linked List

  ข้อมูลของคำ ประกอบด้วย คำที่ต้องทาย ให้แสดงคำทั้งหมดที่มีอยู่ในระบบ

สแตก (Stack)

สแตก (Stack)

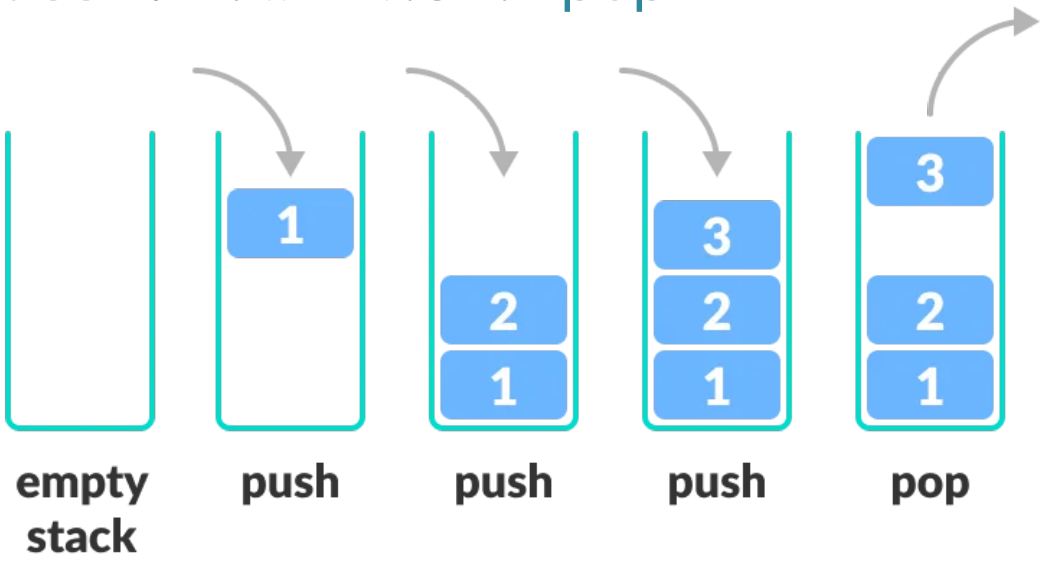
- Stack (สแตก) คือ โครงสร้างข้อมูลชนิดหนึ่งที่ออกแบบมาให้มีลักษณะการนำข้อมูลเข้าและออกจากสแตกจะมีลำดับการทำงานแบบ

"เข้าหลังออกก่อน" (Last In First Out) หรือเรียกสั้น ๆ ว่า LIFO

- เนื่องจากการนำข้อมูลเข้าและออก จะใช้ปลายด้านเดียวกันจึงทำให้ข้อมูลตัวที่นำเข้าไปเก็บก่อนถูกจัดเก็บด้านในสุด และข้อมูลตัวที่จัดเก็บตัวสุดท้ายจะอยู่บนสุด การนำข้อมูลออกจึงต้องนำข้อมูลตัวบนสุดออกก่อน

LIFO Principle of Stack

- การนำข้อมูลเข้าไปเก็บในสแต็ก เรียกว่า push
- การนำข้อมูลออกจากสแต็ก เรียกว่า pop



การดำเนินการเกี่ยวกับสแตก (Stack)

- Push: กระบวนการเพิ่มข้อมูลเข้าไปในสแตก
- Pop: กระบวนการลบหรือดึงข้อมูลออกจากสแตก
- IsEmpty: การตรวจสอบว่าสแตก (Stack) ว่างหรือไม่
- IsFull: การตรวจสอบว่าสแตก (Stack) เต็มหรือไม่
- Peek: การเข้าถึงข้อมูลที่อยู่บนสุดของสแตกโดยไม่ทำการลบข้อมูลออกจากสแตก

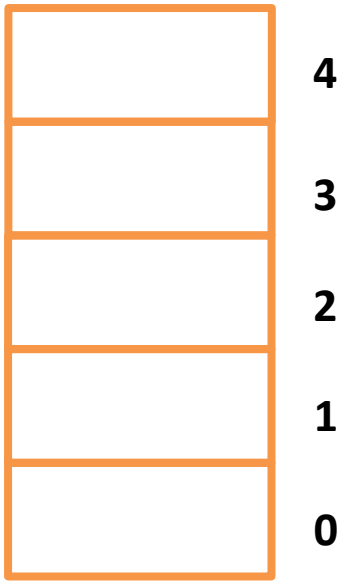
การดำเนินการเกี่ยวกับสแตกโดยใช้อาร์เรย์

- กำหนดตัวแปร TOP สำหรับชี้ element ที่อยู่ด้านบนสุด
- เมื่อเริ่มต้นสร้างสแตก กำหนดค่า top ค่าเป็น -1
- การตรวจสอบว่าสแตกว่างหรือไม่ ใช้วิธีการเปรียบเทียบ TOP == -1
- การเพิ่ม (push) element ใหม่ จะเพิ่มค่าของ TOP และเพิ่ม element ใหม่ตำแหน่ง TOP
- การนำ element ออกจากสแตก (pop) จะส่ง element ที่ตำแหน่ง TOP และลดค่าลง
- ก่อนจะเพิ่มข้อมูลใหม่ (push) ในสแตก จะต้องตรวจสอบก่อนว่าสแตกเต็มแล้วหรือไม่
- ก่อนจะนำข้อมูลออก (pop) จะต้องตรวจสอบก่อนว่า สแตกนั้นว่างหรือไม่

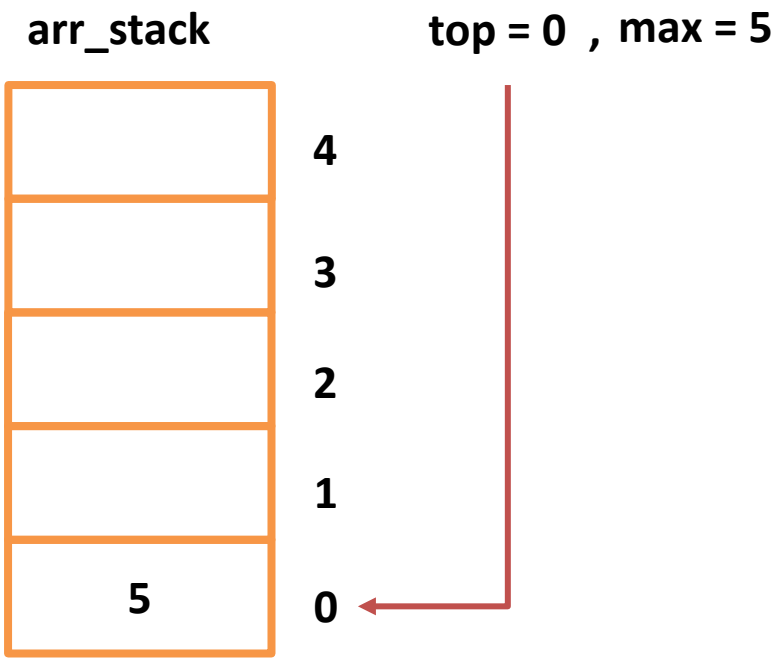
สแตก (โดยใช้อาร์เรย์)

arr_stack

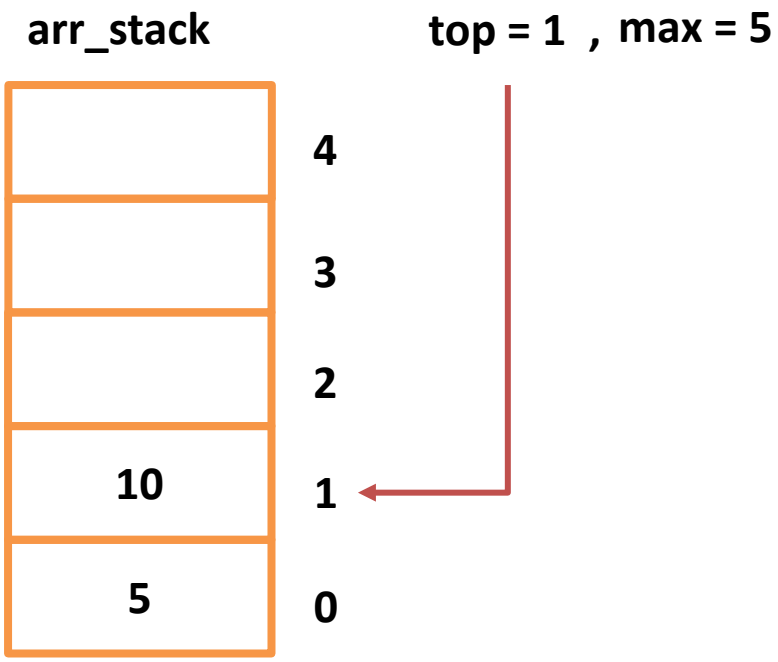
top = -1 ,max = 5



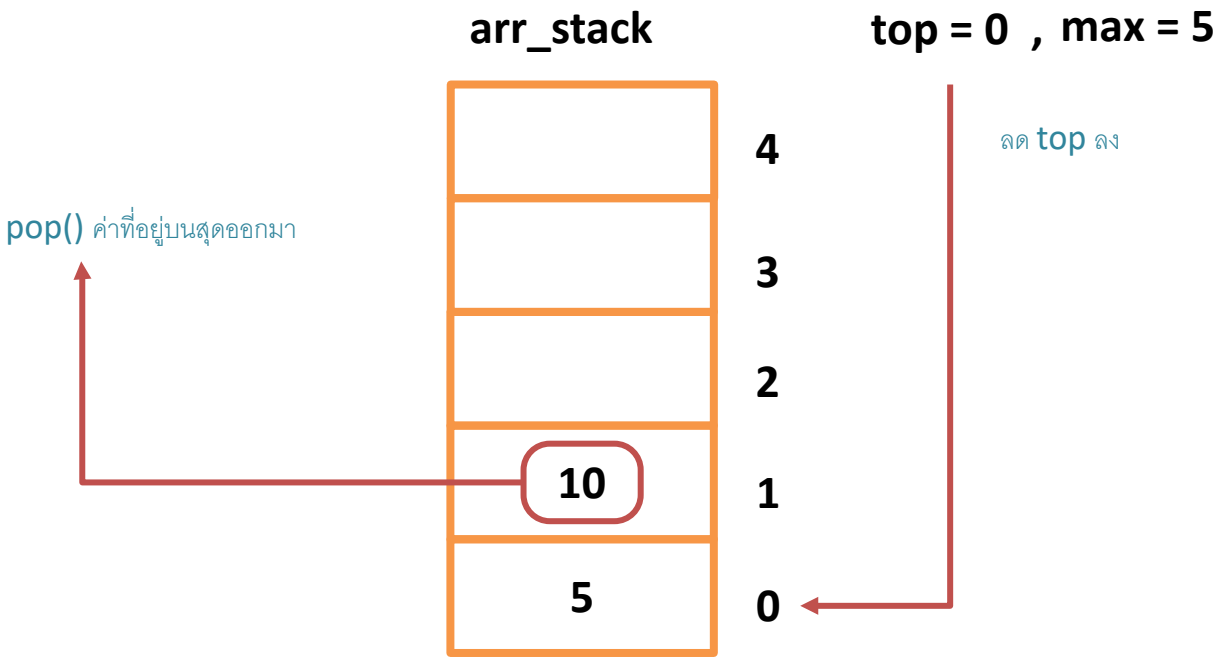
การนำข้อมูลเข้าสแตก push(5)



การนำข้อมูลเข้าสแตก push(10)



การนำข้อมูลออกจากสแตก pop()



Verifying whether the Stack is full: isFull()

- 1. START
- 2. If the size of the stack is equal to the top position of the stack, the stack is full. Return 1.
- 3. Otherwise, return 0.
- 4. END

Verifying whether the Stack is empty: isEmpty()

- 1. START
- 2. If the top value is -1, the stack is empty. Return 1.
- 3. Otherwise, return 0.
- 4. END

Stack Insertion: push()

- 1. Checks if the stack is full.
- 2. If the stack is full, produces an error and exit.
- 3. If the stack is not full, increments top to point next empty space.
- 4. Adds data element to the stack location, where top is pointing.
- 5. Returns success.

```c
#include <stdio.h>
int MAXSIZE = 8;
int stack[8];
int top = -1;


/* Check if the stack is full*/
int isfull(){
  if(top == MAXSIZE)
    return 1;
  else
    return 0;
}

/* Function to insert into the stack */
void push(int data){
  if(!isfull()) {
    top = top + 1;
    stack[top] = data;
  } else {
    printf("Could not insert data, Stack is full.\n");
  }
}
```

```c
/* Main function */
int main(){
  int i;
  push(44);
  push(10);
  push(62);
  push(123);
  push(15);
  printf("Stack Elements: \n");

  // print stack data
  for(i = 0; i < 8; i++) {
    printf("%d ", stack[i]);
  }
  return 0;
}
```

```c
#include <stdio.h>
int MAXSIZE = 8;
int stack[8];
int top = -1;


/* Check if the stack is full*/
int isfull(){
  if(top == MAXSIZE)
    return 1;
  else
    return 0;
}

/* Function to insert into the stack */
void push(int data){
  if(!isfull()) {
    top = top + 1;
    stack[top] = data;
  } else {
    printf("Could not insert data, Stack is full.\n");
  }
}
```

```c
/* Main function */
int main(){
  int i;
  push(44);
  push(10);
  push(62);
  push(123);
  push(15);
  printf("Stack Elements: \n");

  // print stack data
  for(i = 0; i < 8; i++) {
    printf("%d ", stack[i]);
  }
  return 0;
}
```

ผลลัพธ์

Stack Elements:

44 10 62 123 15 0 0 0

Stack Deletion: pop()

- 1. Checks if the stack is empty.
- 2. If the stack is empty, produces an error and exit.
- 3. If the stack is not empty, accesses the data element at which top is pointing.
- 4. Decreases the value of top by 1.
- 5. Returns success.

```c
/* Check if the stack is empty */
int isempty(){
   if(top == -1)
      return 1;
   else
      return 0;
}

/* Function to delete from the stack */
int pop(){
   int data;
   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

```c
/* Main function */
int main(){
   int i;
   push(44);
   push(10);
   push(62);
   push(123);
   push(15);
   printf("Stack Elements: \n");

   // print stack data
   for(i = 0; i < 8; i++) {
      printf("%d ", stack[i]);
   }

   printf("\nElements popped: \n");
   // print stack data
   while(!isempty()) {
      int data = pop();
      printf("%d ",data);
   }
   return 0;
}
```

```c
/* Check if the stack is empty */
int isempty(){
   if(top == -1)
      return 1;
   else
      return 0;
}

/* Function to delete from the stack */
int pop(){
   int data;
   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

```c
/* Main function */
int main(){
   int i;
   push(44);
   push(10);
   push(62);
   push(123);
   push(15);
   printf("Stack Elements: \n");

   // print stack data
   for(i = 0; i < 8; i++) {
      printf("%d ", stack[i]);
   }

   printf("\nElements popped: \n");
   // print stack data
   while(!isempty()) {
      int data = pop();
      printf("%d ",data);
   }
   return 0;
}
```

ผลลัพธ์

```
Stack Elements:
44 10 62 123 15 0 0 0
Elements popped:
15 123 62 10 44
```

Retrieving topmost Element from Stack: peek()

- 1. START
- 2. return the element at the top of the stack
- 3. END

```
/* Function to return the topmost element in the stack */
int peek(){
   return stack[top];
}
```

ตัวอย่างโจทย์ในการใช้ สแตก

ตัวอย่างโจทย์ที่ 1: การตรวจสอบ Parentheses (วงเล็บ)

โจทย์: เขียนโปรแกรมเพื่อตรวจสอบว่าข้อความที่ให้มามีการเปิดและปิดวงเล็บอย่างถูกต้องหรือไม่ เช่น "(a + b) * (c - d)" ถือว่าถูกต้อง แต่ "(a + b) * (c - d" ถือว่าไม่ถูกต้อง

ตัวอย่างโจทย์ที่ 2: การย้อนกลับสตริง

โจทย์: เขียนโปรแกรมเพื่อย้อนกลับสตริงที่ให้มา เช่น จาก "hello" เป็น "olleh"

ตัวอย่างโจทย์ที่ 3: การจัดลำดับการทำงาน (Postfix Evaluation)

โจทย์: เขียนโปรแกรมเพื่อประมวลผลนิพจน์ในรูปแบบ Postfix เช่น "5 1 2 + 4 * + 3 -" ให้ผลลัพธ์เป็น 14

คิว (Queue)

Queue

- Queue (คิว) คือ โครงสร้างข้อมูลชนิดหนึ่งที่ออกแบบมาให้มีลักษณะการนำข้อมูลเข้าและออกจากคิวจะมีลำดับการทำงานแบบ “เข้าก่อนออกก่อน” (First In First Out) หรือเรียกสั้น ๆ ว่า FIFO
- เหมือนกับการต่อแถว โดยที่ข้อมูลลำดับแรกของแถวจะได้ออกก่อน และข้อมูลที่เข้ามาต่อแถวจะอยู่หลังสุด

ลักษณะของ Queue

- มีทางเข้าข้อมูลอยู่ด้านท้ายและออกของข้อมูลอยู่ด้านหน้า
- มีการทำงานแบบตามลำดับ
- สามารถนำข้อมูลเข้าและนำข้อมูลออกสลับกันได้
- มีลำดับการทำงานแบบเข้าก่อนออกก่อน (FIFO)

การดำเนินการของ Queue

- การนำข้อมูลเข้าไปเก็บในคิว ซึ่งเรียกว่า **Enqueue (การเอ็นคิว)** คือ การนำข้อมูลไปเก็บแบบเรียงลำดับ (ต่อท้าย) ในคิว

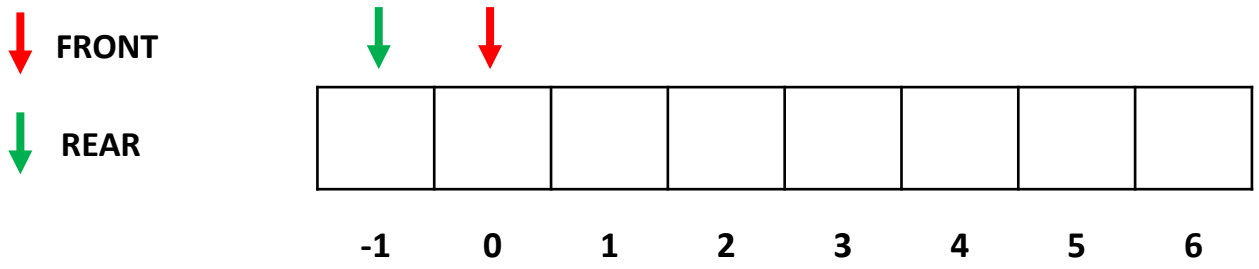
- การนำข้อมูลออกจากคิว ซึ่งเรียกว่า **Dequeue (การดีคิว)** คือ การนำข้อมูลตัวแรกของคิวออกไปใช้งาน

การดำเนินการเกี่ยวกับคิว

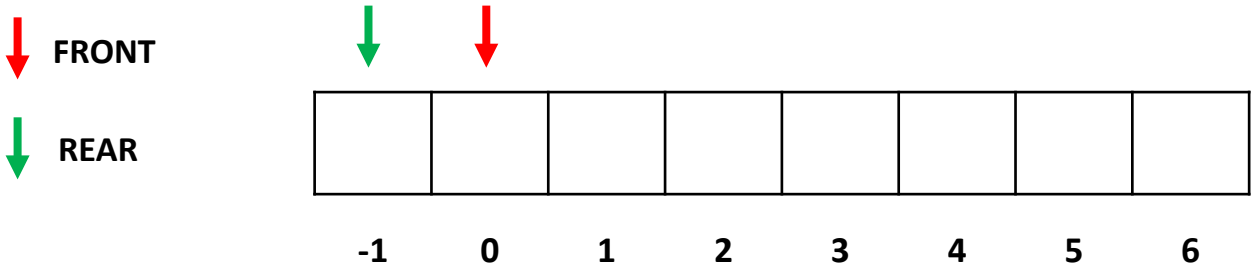
- Enqueue: การเพิ่ม (insert) รายการหรือข้อมูลใหม่ลงในคิว (queue)
- Dequeue: การนำ (remove) รายการหรือข้อมูลออกจากคิว
- IsEmpty: การตรวจสอบว่าคิว ว่างหรือไม่
- IsFull: การตรวจสอบว่าคิว เต็มหรือไม่
- Peek: การดูค่าของข้อมูลที่อยู่ด้านหน้า (front) ของคิวโดยไม่ทำการลบข้อมูลนั้นออกจากคิว

การทำงานเกี่ยวกับคิว (Queue)

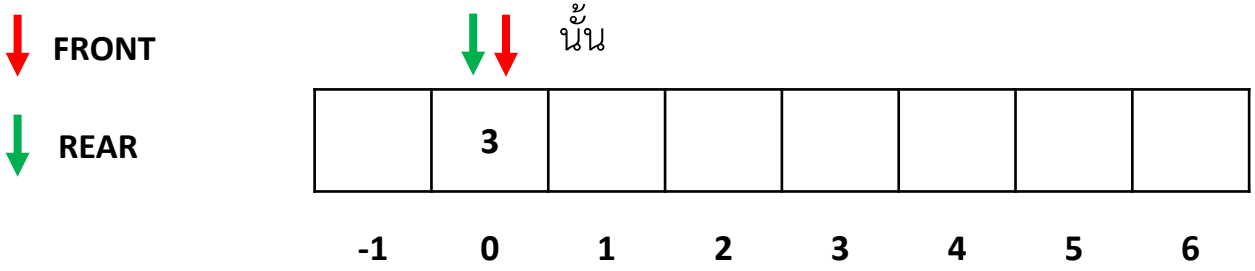
- มีตัวชี้ 2 ตัว คือ FRONT และ REAR
- FRONT ใช้ชี้ element ตัวแรกของคิว
- REAR ใช้ชี้ element ตัวสุดท้ายของคิว
- การกำหนดค่าเริ่มต้น FRONT = 0 และ REAR มีค่าเป็น -1



การเพิ่มข้อมูลในคิว (Enqueue)



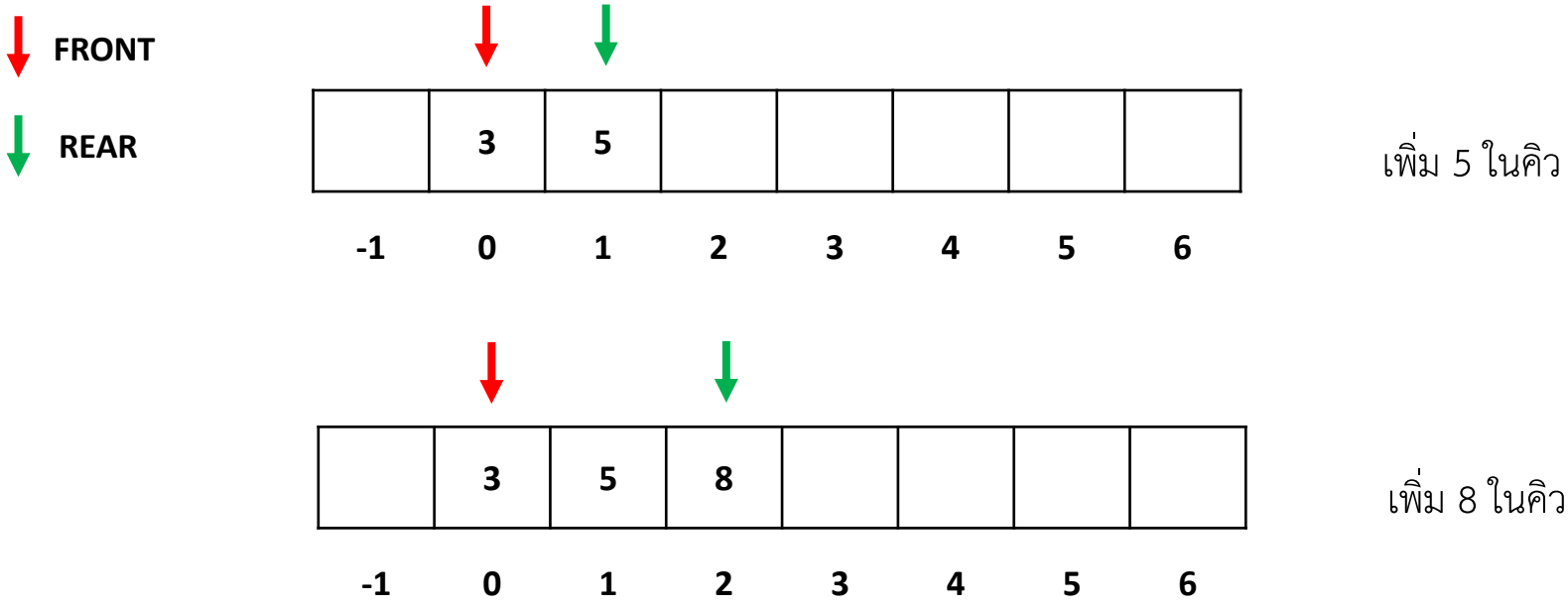
ปรับตัวชี้ rear ไปยังตำแหน่งถัดไป แล้วเพิ่มข้อมูลในตำแหน่งนั้น



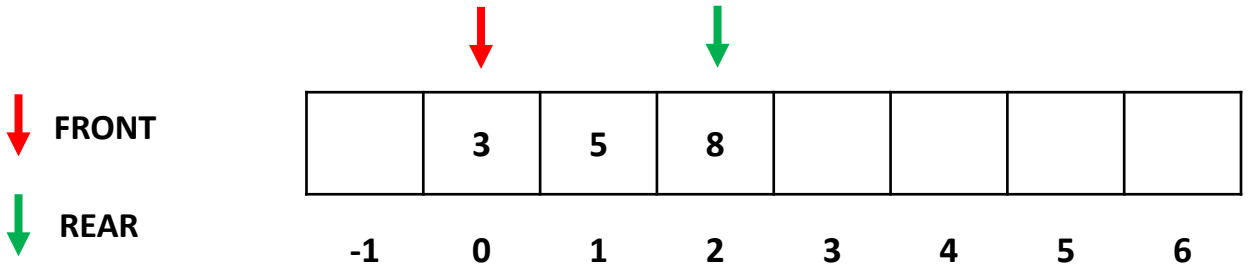
เพิ่ม 3 ในคิว

การเพิ่มข้อมูลในคิว (Enqueue)

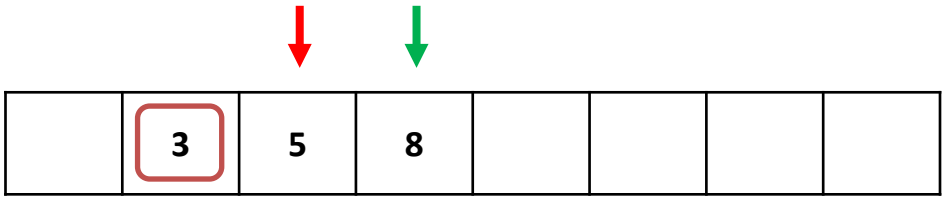
ปรับตัวชี้ rear ไปยังตำแหน่งถัดไป แล้วเพิ่มข้อมูลในตำแหน่งนั้น



การนำข้อมูลออกจากคิว (Dequeue)



นำข้อมูลที่ตำแหน่งตัวชี้ front แล้วเลื่อนตัวชี้ไปยังตำแหน่งถัดไป



Queue Insertion Operation: Enqueue()

- 1. START
- 2. Check if the queue is full.
- 3. If the queue is full, produce overflow error and exit.
- 4. If the queue is not full, increment rear pointer to point the next empty space.
- 5. Add data element to the queue location, where the rear is pointing.
- 6. return success.
- 7. END

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 6
int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;
bool isFull(){
   return itemCount == MAX;
}
bool isEmpty(){
   return itemCount == 0;
}
int removeData(){
   int data = intArray[front++];
   if(front == MAX) {
      front = 0;
   }
   itemCount--;
   return data;
}
```

```c
void insert(int data){
   if(!isFull()) {
      if(rear == MAX-1) {
         rear = -1;
      }
      intArray[++rear] = data;
      itemCount++;
   }
}

   int main(){
      insert(3);
      insert(5);
      insert(9);
      insert(1);
      insert(12);
      insert(15);
      printf("Queue: ");
      while(!isEmpty()) {
         int n = removeData();
         printf("%d ",n);
      }
   }
```

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 6
int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;
bool isFull(){
   return itemCount == MAX;
}
bool isEmpty(){
   return itemCount == 0;
}
int removeData(){
   int data = intArray[front++];
   if(front == MAX) {
      front = 0;
   }
   itemCount--;
   return data;
}
```

```c
void insert(int data){
   if(!isFull()) {
      if(rear == MAX-1) {
         rear = -1;
      }
      intArray[++rear] = data;
      itemCount++;
   }
}

   int main(){
      insert(3);
      insert(5);
      insert(9);
      insert(1);
      insert(12);
      insert(15);
      printf("Queue: ");
      while(!isEmpty()) {
         int n = removeData();
         printf("%d ",n);
      }
   }
```

ผลลัพธ์ Queue: 3 5 9 1 12 15

Queue Deletion Operation: dequeue()

- 1. START
- 2. Check if the queue is empty.
- 3. If the queue is empty, produce underflow error and exit.
- 4. If the queue is not empty, access the data where front is pointing.
- 5. Increment front pointer to point to the next available data element.
- 6. Return success.
- 7. END

```
void insert(int data){
   if(!isFull()) {
      if(rear == MAX-1) {
         rear = -1;
      }
      intArray[++rear] = data;
      itemCount++;
   }
}
int removeData(){
   int data = intArray[front++];
   if(front == MAX) {
      front = 0;
   }
   itemCount--;
   return data;
}
```

```
int main(){
   int i;
   /* insert 5 items */
   insert(3);
   insert(5);
   insert(9);
   insert(1);
   insert(12);
   insert(15);
   printf("Queue: ");
   for(i = 0; i < MAX; i++)
      printf("%d ", intArray[i]);
   // remove one item
   int num = removeData();
   printf("\nElement removed: %d\n",num);
   printf("Updated Queue: ");
   while(!isEmpty()) {
      int n = removeData();
      printf("%d ",n);
   }
}
```

```
void insert(int data){
   if(!isFull()) {
      if(rear == MAX-1) {
         rear = -1;
      }
      intArray[++rear] = data;
      itemCount++;
   }
}
int removeData(){
   int data = intArray[front++];
   if(front == MAX) {
      front = 0;
   }
   itemCount--;
   return data;
}
```

ผลลัพธ์

Queue: 3 5 9 1 12 15

Element removed: 3

Updated Queue: 5 9 1 12 15

```
int main(){
   int i;
   /* insert 5 items */
   insert(3);
   insert(5);
   insert(9);
   insert(1);
   insert(12);
   insert(15);
   printf("Queue: ");
   for(i = 0; i < MAX; i++)
      printf("%d ", intArray[i]);
   // remove one item
   int num = removeData();
   printf("\nElement removed: %d\n",num);
   printf("Updated Queue: ");
   while(!isEmpty()) {
      int n = removeData();
      printf("%d ",n);
   }
}
```

Queue - The peek() Operation

- 1. START
- 2. Return the element at the front of the queue
- 3. END

```
int peek() {
   return intArray[front];
}
```

Queue - The isFull() Operation

- 1. START
- 2. If the count of queue elements equals the queue size, return true
- 3. Otherwise, return false
- 4. END

```
bool isFull() {
   return itemCount == MAX;
}
```

Queue - The isEmpty() operation

- 1. START
- 2. If the count of queue elements equals zero, return true
- 3. Otherwise, return false
- 4. END

```
bool isEmpty() {
   return itemCount == 0;
}
```

Reference

- https://www.programiz.com/dsa/linked-list-operations
- https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm

ตัวอย่างโจทย์ในการใช้ คิว

ตัวอย่างโจทย์ที่ 1: การจัดการคำขอในเซิร์ฟเวอร์

โจทย์:

สร้างระบบจัดการคำขอ (request) ที่เข้ามายังเซิร์ฟเวอร์ โดยใช้โครงสร้างข้อมูลแบบคิว เซิร์ฟเวอร์จะต้องจัดการกับคำขออย่างมีระเบียบ โดยคำขอที่เข้ามาก่อนจะถูกประมวลผลก่อน (First-In-First-Out - FIFO)

รายละเอียด:

- เมื่อมีคำขอเข้ามาใหม่ ให้เพิ่มคำขอลงในคิว
- เมื่อเซิร์ฟเวอร์พร้อมจะประมวลผลคำขอ ให้ดึงคำขอออกจากคิว
- แสดงผลลัพธ์ว่าเซิร์ฟเวอร์ได้ประมวลผลคำขอใดไปแล้ว

วิธีการแก้ปัญหา:

สร้างโครงสร้างข้อมูลคิวที่สามารถเก็บคำขอ

ฟังก์ชันสำหรับเพิ่มคำขอ (enqueue) ลงในคิว

ฟังก์ชันสำหรับประมวลผลคำขอ (dequeue) ออกจากคิว

แสดงผลคำขอที่ถูกประมวลผล

ตัวอย่างโจทย์ในการใช้ คิว (ต่อ)

ตัวอย่างโจทย์ที่ 2: ระบบจัดการงาน (Job Scheduling)

โจทย์:

สร้างระบบจัดการงานที่มีการรอคอยในคิว โดยงานที่เข้ามาก่อนจะต้องถูกประมวลผลก่อน เช่น การพิมพ์เอกสาร การคำนวณ เป็นต้น

รายละเอียด:

- งานแต่ละงานจะมี ID และเวลาที่ใช้ในการประมวลผล
- ให้ทำการ Enqueue งานเข้ามาในคิว
- เมื่อเครื่องพร้อม จะประมวลผลงานที่อยู่ในด้านหน้า
- แสดงผลว่ามีงานใดถูกประมวลผล

วิธีการแก้ปัญหา:

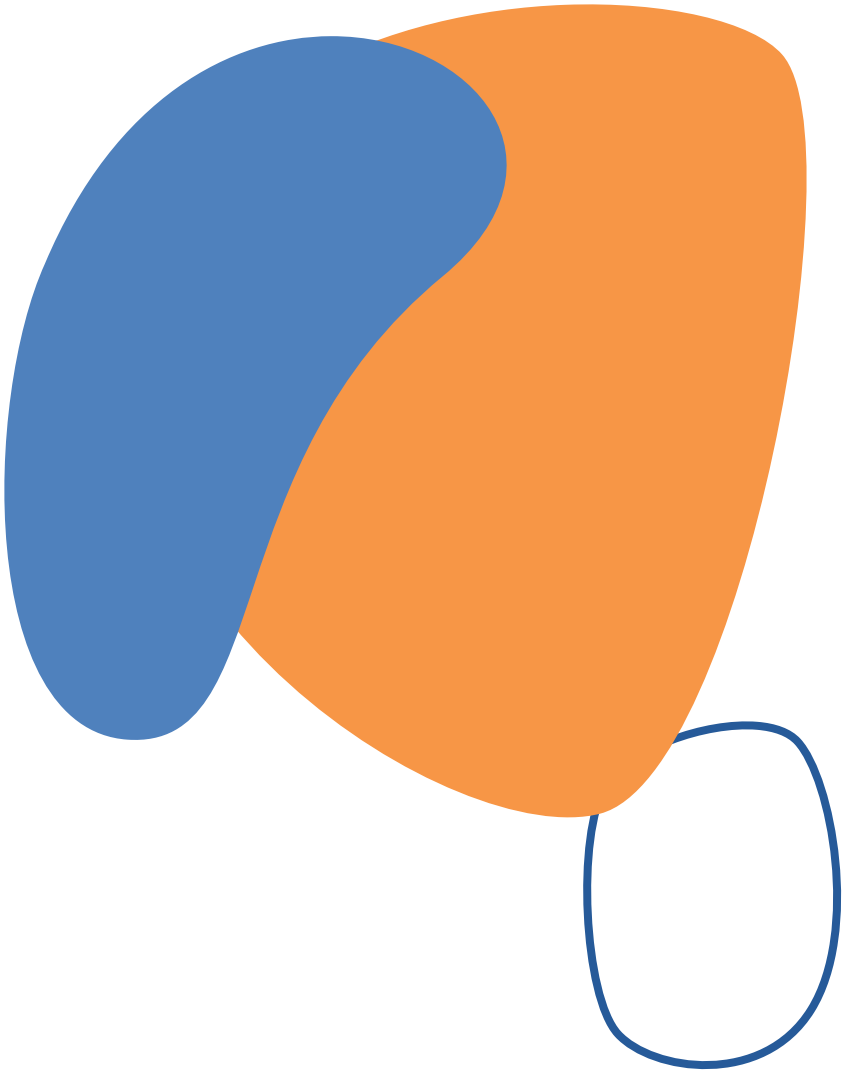
สร้างโครงสร้างข้อมูลคิวที่เก็บงาน

ฟังก์ชันสำหรับเพิ่มงานลงในคิว

ฟังก์ชันสำหรับประมวลผลงานออกจากคิว

แสดงผลงานที่ถูกประมวลผล

โจทย์ปัญหา



แบบฝึกหัดลิงก์ลิสต์ 1

- สร้างโปรแกรมที่รับข้อมูลแล้วเก็บไว้ใน List โดยโจทย์จะให้ตัวเลขมาทั้งหมด 4 ตัวเลข ให้ผู้เรียนแสดงผลตัวเลขทั้ง 4 ตัวเลขโดยคั่นด้วยเครื่องหมาย > ดังตัวอย่าง

Input	Output
5 6 7 8	5>6>7>8

แบบฝึกหัดลิงก์ลิสต์ 2

- ต่อยอดจากข้อที่แล้วโจทย์จะให้ตัวเลขเพิ่มอีก 1 ตัวเลข โดยทำการสร้างโปรแกรมที่รับข้อมูลเก็บไว้ใน List ทั้งหมด 4 ค่า แล้วให้ลบข้อมูลที่ตรงกับข้อมูลตัวที่ 5 ออกจาก List

Input	Output
5 6 7 8 6	5>7>8

แบบฝึกหัดลิงก์ลิสต์ 3

- เขียนโปรแกรมสำหรับรับข้อมูล โดยข้อมูลตัวแรกเป็นตัวเลขสำหรับบอกจำนวนทั้งหมด แล้วรับข้อมูลตามจำนวนตัวเลขที่รับมา แล้วให้แสดงผลข้อมูลที่รับเข้ามาทั้งหมด โดยที่ข้อมูลตัวสุดท้ายจะถูกแทรกเข้ามาอยู่เป็นตัวที่ 2 เสมอ

Input	Output
3 7 5 2	7>2>5

แบบฝึกหัดสแตก 1

- เขียนโปรแกรมสำหรับรับข้อมูล โดยข้อมูลตัวแรกเป็นตัวเลขสำหรับบอกจำนวนทั้งหมด แล้วรับข้อมูลตามจำนวนตัวเลขที่รับมา เพื่อเก็บไว้ใน Stack และให้แสดงผลลัพธ์เมื่อใช้คำสั่ง Peek กับ Stack ที่เก็บเอาไว้

Input	Output
3 7 5 2	2

แบบฝึกหัดสแตก 2

- เขียนโปรแกรมสำหรับรับข้อมูล โดยข้อมูลตัวแรกเป็นตัวเลขสำหรับบอกจำนวนทั้งหมด แล้วรับข้อมูลตามจำนวนตัวเลขที่รับมา เพื่อเก็บไว้ใน Stack และให้แสดงผลตัวเลขย้อนกลับ (reverse)

Input	Output
3 7 5 2	2 5 7

แบบฝึกหัดสแตก 3

- เขียนโปรแกรมเพื่อรับข้อมูล
- บรรทัดที่ 1 จะบอกจำนวนคำสั่ง
- บรรทัดที่ 2 จะบอกว่าให้ทำ Push หรือ Pop ถ้าเป็นการ Push บรรทัดถัดไปจะเป็นตัวเลขจำนวนเต็มที่จะให้ Push เข้าไปใน Stack
- คำสั่งแรกจะเป็น Push เสมอ และจะไม่มีการสั่ง Pop ในขณะที่ Stack ไม่มีข้อมูลข้างใน
- เมื่อรับคำสั่งและข้อมูลครบแล้วให้แสดงข้อมูลตัวเลขที่ถูก Pop ออกมาทั้งหมดโดยเรียงตามลำดับที่โปรแกรมทำงานจริง

Input	Output
4 push 1 push 5 pop pop	5 1

แบบฝึกหัดคิว 1

- เขียนโปรแกรมเพื่อรับค่าตัวเลขจำนวนเต็มจำนวน 5 ตัวเก็บไว้ใน Queue และทำการแสดงผลข้อมูลทั้งหมดใน Queue ดังกล่าว

Input	Output
3 7 5 2 9	3, 7, 5, 2, 9

แบบฝึกหัดคิว 2

- เขียนโปรแกรมเพื่อรับค่าเข้ามาเก็บใน Queue จนกว่าจะเจอข้อความว่า stop แล้วจึงแสดงผลข้อมูลทั้งหมดใน Queue และจำนวนข้อมูลใน Queue

Input	Output
ant bird cat dog eagle stop	1 ant 2 bird 3 cat 4 dog 5 eagle Total 5

แบบฝึกหัดคิว 3

- เขียนโปรแกรมเพื่อเพื่อรับข้อมูล
- บรรทัดที่ 1 ขนาดของ Queue
- บรรทัดที่ 2 เป็นต้นไป เป็นตัวเลขจำนวนเต็มที่ต้องการให้ใส่เข้าไปใน Queue
- บรรทัดสุดท้ายเป็นเลข 0 เพื่อบอกการสิ้นสุด input
- เมื่อรับข้อมูลใส่เข้าไปใน Queue จนครบตามขนาดที่กำหนดแล้ว ถ้ามีข้อมูลเพิ่มเข้ามาอีก ให้ทำการ dequeue แล้วจึง enqueue ข้อมูลตัวนั้น ทำไปจนกว่าจะได้รับเลข 0 จึงหยุดการทำงาน

Input	Output
3 7 5 2 9 0	5, 2, 9