4 장. 큐 (queue)

Jinseog Kim

학습목표

1. 큐 (queue) 를 이해한다
2. 배열을 이용하여 큐를 구현한다.
3. 덱 (deque) 를 이해한다.
4. 덱의 응용: 미로 탐색

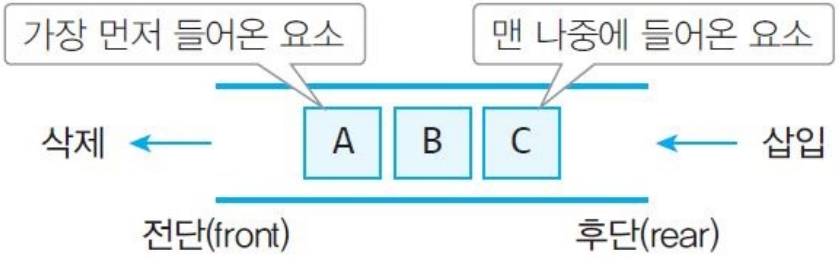
큐 (queue)

- 먼저 들어온 데이터가 먼저 나가는 자료구조
  - (예) 매표소의 대기열

  - 선입선출 (FIFO: First-In First-Out)

큐의 구조

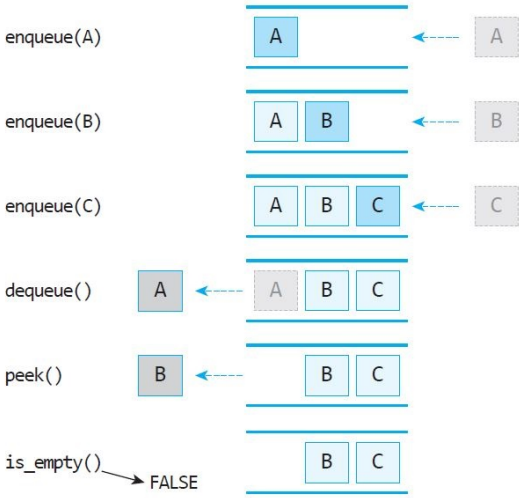
- 큐의 전단 (front)
- 큐의 후단 (rear)



큐의 추상자료형 (ADT)

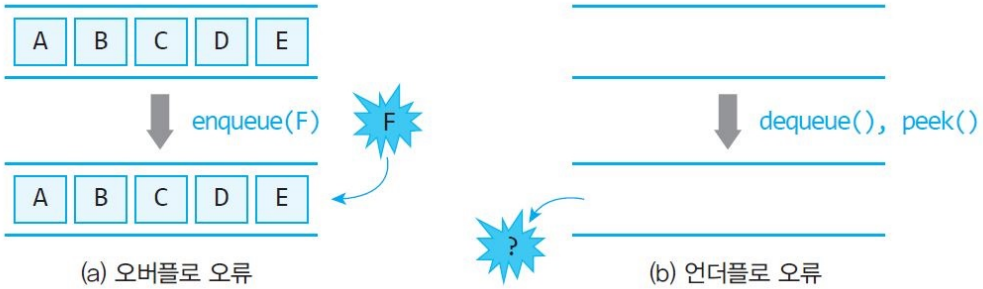
- **데이터 (객체)**: 선입선출 (FIFO) 방식을 가능하게 하는 요소의 모임
- **연산 (함수)**
  - `initialize()`: 큐의 초기화
  - `enqueue(e)`: 큐의 맨 뒤에 요소 `e` 를 추가
  - `dequeue()`: 큐의 맨 앞의 요소를 꺼내 (삭제) 반환
  - `is_full()`: 큐가 꽉 차있으면 TRUE, 아니면 FALSE
  - `is_empty()`: 큐가 비어있으면 TRUE, 아니면 FALSE
  - `peek()`: 큐의 맨 앞 요소를 삭제하지 않고 반환

큐의 연산


enqueue(A)
A
A
enqueue(B)
A
B
B
enqueue(C)
A
B
C
C
dequeue()
A
A
B
C
peek()
B
B
C
is_empty()
FALSE
B
C

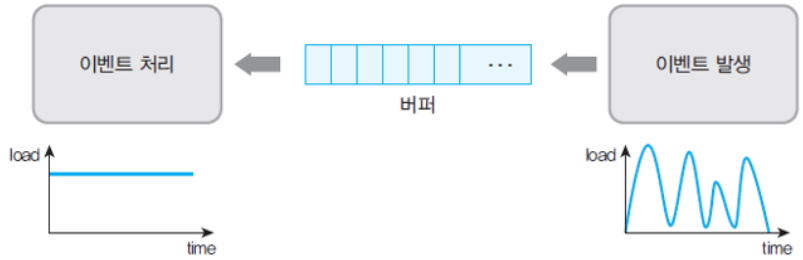
큐에서 발생할 수 있는 오류

- 오버플로 (overflow): 큐가 꽉차있어 더 이상 요소를 추가할 수 없는 경우
- 언더플로 (underflow): 큐가 비어있어 요소를 가져오거나 삭제할 수 없는 경우



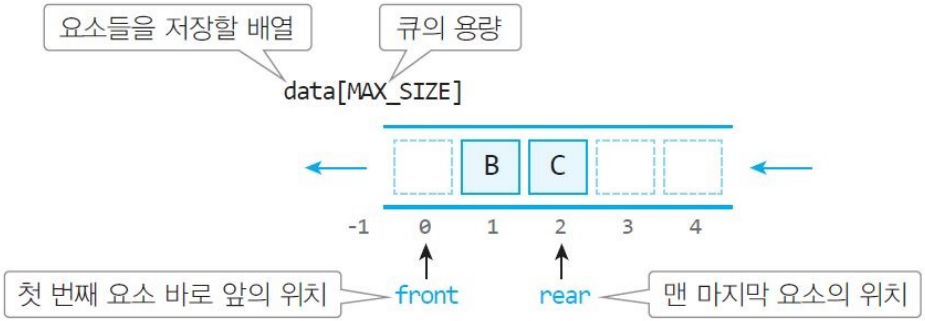
큐의 활용 분야

- 버퍼 (buffer) : 갑자기 데이터가 몰려드는 경우 이들을 잠시 보관하는 장소



- 큐의 응용
  - 빠른 CPU 와 속도가 상대적으로 느린 주변 장치 (예: 프린터) 들 사이의 시간이나 속도 차이를 극복하기 위한 버퍼
  - 컴퓨터로 현실 세계를 시뮬레이션하는 분야
  - 다양한 알고리즘: 레벨 순회, 너비우선탐색 (BFS), 기수 정렬, 등

1. **배열을 활용**

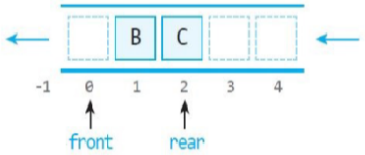


   - rear: 맨 마지막 요소의 위치 (인덱스) 를 저장
   - front: 첫 번째 요소가 아니라 그 요소 바로 앞의 위치를 저장

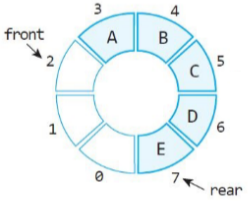
2. 연결 리스트 (linked list) 를 이용: 나중에 배울 것임

큐의 형태에 따른 분류

1. 선형 큐 (linear queue): 요소들이 일렬로 배열
2. 원형 큐 (circular queue): 마지막 요소가 첫 번째 요소와 연결된 원형 구조



선형 큐

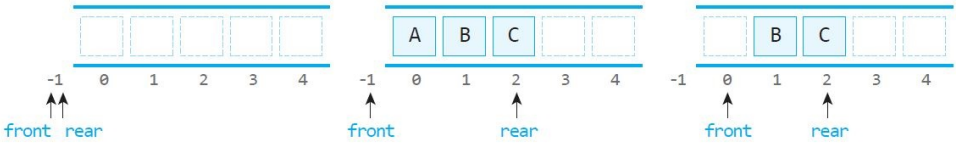


원형 큐

	선형 큐	원형 큐
장점	구현이 간단	공간을 효율적으로 사용
단점	큐가 가득 차면 공간이 낭비됨	구현이 복잡, 관리가 필요함

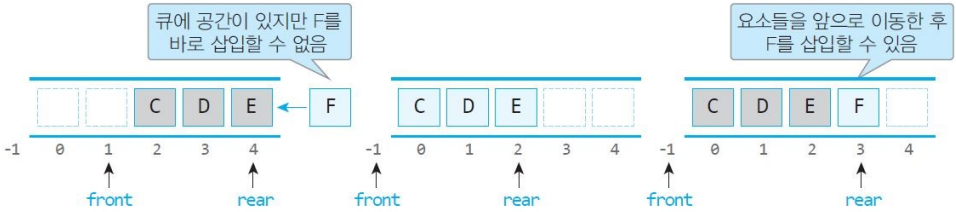
선형 큐 (linear queue)

- 선형 큐의 동작



(a) 초기 상태 (b) enqueue(A)~enqueue(C) (c) dequeue()

- 선형 큐의 문제점



(a) enqueue(F) ? (b) 기존 요소들을 이동 (c) enqueue(F) 처리

선형 큐의 구현 (데이터)

```c
typedef struct Queue {
    int data[MAX_SIZE]; // 요소의 배열
    int front;          // 전단 인덱스
    int rear;           // 후단 인덱스
    int size;           // 큐에 있는 요소의 개수
} Queue;

#define MAX_SIZE 100;
```

선형 큐의 구현 (연산)

- 생성, 상태 검사

```
1  int init(struct Queue queue){ // 초기화
2      queue.front = 0;
3      queue.rear = 0;
4      queue.size = 0;
5  }
6  int isFull(struct Queue queue){ // full?
7      return (queue.size == MAX_SIZE);
8  }
9
10 int isEmpty(struct Queue queue){ // empty?
11     return (queue.size == 0);
12 }
```

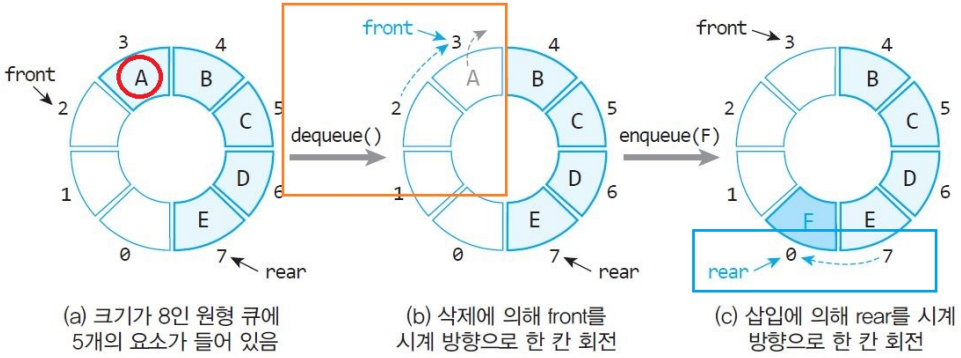
선형 큐의 구현 (연산)

- 추가 (삽입)

```
1  void enqueue(struct Queue queue, int item)
2  {
3      if (isFull(queue))
4          return;
5      queue.rear = (queue.rear + 1) % queue->capacity;
6      queue.array[queue.rear] = item;
7      queue->size = queue->size + 1;
8      printf("%d enqueued to queue\n", item);
9  }
```

원형 큐 (Circular Queue)

- 인덱스 front 와 rear 를 원형으로 회전



(a) 크기가 8인 원형 큐에 5개의 요소가 들어 있음

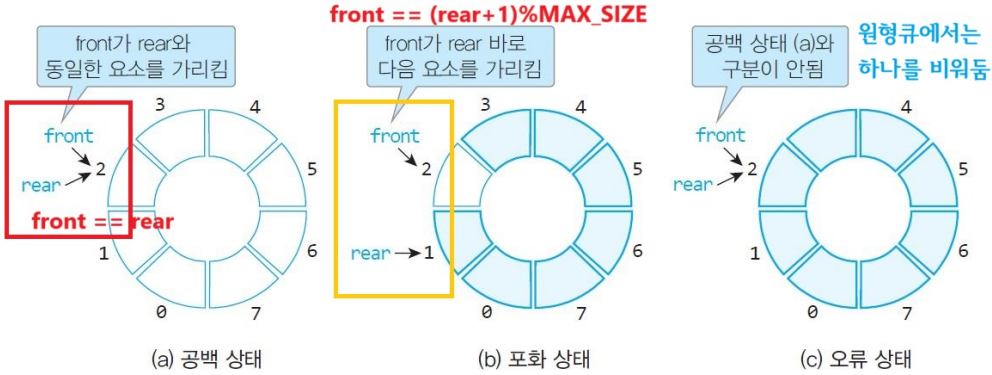
(b) 삭제에 의해 front를 시계 방향으로 한 칸 회전

(c) 삽입에 의해 rear를 시계 방향으로 한 칸 회전

- 인덱스 회전 방법 - 나머지 연산 (%) 이용
  - 전단회전: front <- (front + 1) % MAX_SIZE // dequeue() 직후
  - 후단회전: rear <- (rear + 1) % MAX_SIZE // enqueue(e) 직후

원형 큐의 연산

- 공백 상태 / 포화 상태 검사: is_empty(), is_full()




```
1  is_empty()
2    if front == rear: return TRUE
3    else: return FALSE
```

```
1  is_full()
2    if front == (rear+1)%MAX_SIZE: return TRUE
3    else: return FALSE
```

원형 큐의 구현: 데이터

- CircularQueue.h 참고

```
1 // 전역변수
2 Element data[MAX_SIZE]; // 요소의 배열
3 int front; // 전단 인덱스
4 int rear; // 후단 인덱스
```

원형 큐의 구현: 연산

```
1  void init_queue(){//초기화
2      front = rear = 0;
3  }
4  int is_empty(){//공백상태
5      return front == rear;
6  }
7  int is_full(){//포화상태
8      return front == (rear + 1)%MAX_SIZE;
9  }
```

```
1  void enqueue(Element val){//삽입연산
2      if (is_full()) error("Overflow!");
3      rear = (rear + 1) % MAX_SIZE;
4      data[rear] = val;
5  }
6  Element dequeue(){// 원형큐의 삭제 연산
7      if (is_empty()) error("Underflow!");
8      front = (front + 1) % MAX_SIZE;
9      return data[front];
10 }
11 Element peek(){// 원형큐의 탐색 (peek) 연산
12     if (is_empty()) error("Underflow!");
13     return data[(front + 1) % MAX_SIZE];
14 }
```

원형 큐의 규현: 테스트

- CircularQueue.c 참고

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 8

typedef int Element;
#include "CircularQueue.h"

void print_queue(char msg[]) { // 큐의 내용을 화면에 출력하는 함수
    printf("%s front=%d, rear=%d --> ", msg, front, rear);
    int size = (rear - front + MAX_SIZE) % MAX_SIZE;
    for (int i = front + 1; i <= front + size; i++)
        printf("%2d ", data[i % MAX_SIZE]);
    printf("\n");
}
```

원형 큐의 규현: 테스트

```
1  void main(){
2      init_queue();
3
4      for (int i = 1; i < 7; i++) enqueue(i);
5      print_queue("enqueue 1~6: ");
6
7      for (int i = 0; i < 4; i++) dequeue();
8      print_queue("dequeue 4 회: ");
9
10     for (int i = 7; i < 10; i++) enqueue(i);
11     print_queue("enqueue 7~9: ");
12 }
```

```
$ gcc CircularQueue.c -o CircularQueue
$ ./CircularQueue
enqueue 1~6:  front=0, rear=6 -->  1  2  3  4  5  6
dequeue 4회:  front=4, rear=6 -->  5  6
enqueue 7~9:  front=4, rear=1 -->  5  6  7  8  9
```

Lab: 맛집의 웨이팅 정보 저장하기

- 손님들의 대기 정보를 저장하는 큐

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100    // 배열의 크기
struct Waiting {        // 대기 정보 구조체
    int id;             // 대기번호
    int nperson;        // 인원
    char info[32];      // 전화번호
};
typedef struct Waiting Element;
#include "CircularQueue.h"
```

Lab: 맛집의 웨이팅 정보 저장하기

```
1  void main(){
2      Element waiting[4] = {
3          { 12, 2, "010-xxxx-1234" }, { 13, 4, "010-xxxx-7809" },
4          { 14, 3, "010-xxxx-4785" }, { 15, 2, "010-xxxx-7345" } };
5      init_queue();
6      for (int i = 0; i < 4; i++) {
7          printf(" 웨이팅 신청을 완료했습니다. 대기번호: %d 번 인원:%d 명\n",
8              waiting[i].id, waiting[i].nperson);
9          enqueue(waiting[i]);
10     }
11     while (!is_empty()) {
12         Element w = dequeue();
13         printf(" 웨이팅 번호 %d 번 입장하실 차례입니다. (%d 명, %s)\n",
14             w.id, w.nperson, w.info);
15     }
16 }
```

Lab: 맛집의 웨이팅 정보 저장하기

```
$ gcc waiting_queue2.c -o waiting
$ ./waiting
```

```
웨이팅 신청. 대기번호: 12번 인원:2명
웨이팅 신청. 대기번호: 13번 인원:4명
웨이팅 신청. 대기번호: 14번 인원:3명
웨이팅 신청. 대기번호: 15번 인원:2명
front=0, rear=4 -->
       [번호=1]12, 인원=2, 연락처=010-xxxx-1234
       [번호=2]13, 인원=4, 연락처=010-xxxx-7809
       [번호=3]14, 인원=3, 연락처=010-xxxx-4785
       [번호=4]15, 인원=2, 연락처=010-xxxx-7345

웨이팅 번호 12번 입장. (2명, 010-xxxx-1234)
웨이팅 번호 13번 입장. (4명, 010-xxxx-7809)
웨이팅 번호 14번 입장. (3명, 010-xxxx-4785)
웨이팅 번호 15번 입장. (2명, 010-xxxx-7345)
front=4, rear=4 -->
```

Lab: 피보나치 수 구하기

- 피보나치 수열: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F(n-1) + F(n-2) & n \geq 2 \end{cases}$$

- 아래와 같은 절차를 이용
  1. enqueue (0): [0]
  2. enqueue (1): [0,1]
  3. dequeue() + peek() ==> enqueue(result)
     1. dequeue(): 0 <— [1]
     2. peek(): 1 <— [1]
     3. result = 0 + 1 = 2
     4. enqueue(result): [1,2]

Lab: 피보나치 수 구하기

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 5   // 배열의 크기
typedef int Element; // Element 의 자료형 정의
#include "CircularQueue.h"
int fibonacci(int n){
    if (n <= 1) return n;
    init_queue();
    enqueue(0);
    enqueue(1);
    for (int i = 2; i <= n; i++) {
        int n1 = dequeue();
        int n2 = peek();
        enqueue(n1 + n2); // F(n) = F(n-2) + F(n-1)
    }
    dequeue(); // 큐에는 F(n-1), F(n) 이 남아 있음
    return dequeue(); // F(n) 를 반환
}
```

```c
void main(){
    printf(" 피보나치 수열: ");
    for (int i = 0; i < 16; i++)
      printf("%d,", fibonacci(i));

    printf("\n\n");
}
```

구조체와 매개변수 전달을 이용한 큐

- 앞에서 정의한 큐는 데이터가 전역 변수로 선언됨
  - 여러개의 큐를 이용한 프로그램 작성이 불가
- **대안:** 큐의 데이터들을 구조체에 저장
  - 구조체의 주소를 함수의 매개변수로 전달 (포인터 사용)
  - QueueStruct.h

```
1  typedef struct Queue {
2      Element data[MAX_SIZE]; // 요소의 배열
3      int front;              // 전단 인덱스
4      int rear;               // 후단 인덱스
5  } Queue;
6
7  #define MAX_SIZE 100;
```

구조체와 매개변수 전달을 이용한 큐: 연산

```
1  void enqueue(Queue * q, Element e){
2    if (is_full(q)) error "overflow";
3    else{
4      q->rear = (q->rear+1) % MAX_SIZE;
5      q->data[q->rear] = e;
6    }
7  }
8
9  Element dequeue(Queue * q){
10   if (is_empty(q)) error "Underflow";
11   else{
12     q->front = (q->front+1) % MAX_SIZE
13     return q->data[q->front]
14   }
15 }
```

구조체와 매개변수 전달을 이용한 큐: 연산

```
1  Element peek(Queue* q){
2    if (is_empty(q)) error("Underflow Error!");
3    return q->data[(q->front + 1) % MAX_SIZE];
4  }
5
6  void print_queue(Queue* q) {
7    printf("front=%d, rear=%d --> ", q->front, q->rear);
8    int size = (q->rear - q->front + MAX_SIZE) % MAX_SIZE;
9    for (int i = q->front + 1; i <= q->front + size; i++)
10       printf("[%d]%2d ", i, q->data[i % MAX_SIZE]);
11   printf("\n");
12 }
```

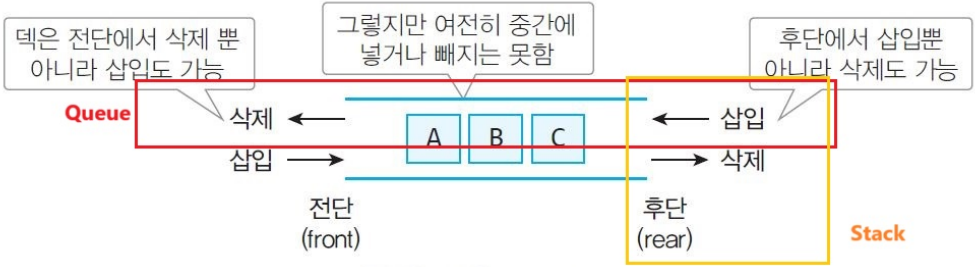
구조체와 매개변수 전달을 이용한 큐: 테스트

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 8
typedef int Element;
#include "QueueStruct.h"

void main(){
    Queue q;
    init_queue(&q);
    print_queue(&q); //front=0, rear=0 -->
    enqueue(&q, 10);
    print_queue(&q); //front=0, rear=1 --> [1]10
    enqueue(&q, 20);
    print_queue(&q); //front=0, rear=2 --> [1]10 [2]20
    dequeue(&q);
    print_queue(&q); //front=1, rear=2 --> [2]20
}
```

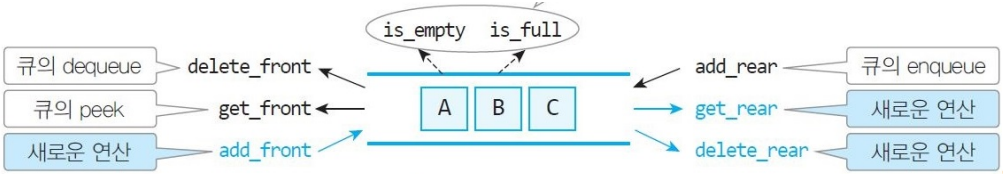
덱 (deque) 이란?

- Queue: FIFO(후단 삽입, 전단 삭제)
- 덱 (deque; double-ended queue): 전단과 후단에서 모두 삽입과 삭제가 가능한 큐
  - queue 와 stack 의 특징이 혼합된 형태

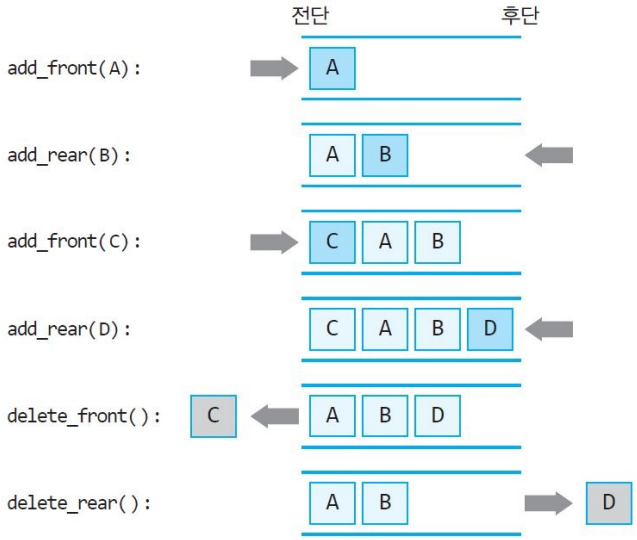


덱의 추상 자료형 (ADT)

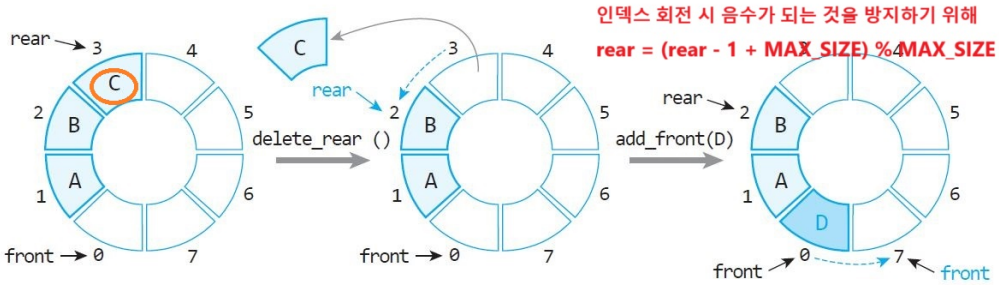
- 데이터: 전단과 후단에서 모두 접근가능한 객체
- 연산
  - `initialize()`: deque 의 초기화
  - `add_front(e)`: 맨 앞에 요소 e 를 추가
  - `add_rear(e)`: 맨 뒤 요소 e 를 추가
  - `delete_front()`: deque 의 맨 앞의 요소를 꺼내 (삭제) 반환
  - `delete_rear()`: deque 의 맨 뒤의 요소를 꺼내 (삭제) 반환
  - `get_front()`: deque 의 맨 앞 요소를 삭제하지 않고 반환
  - `get_rear()`: deque 의 맨 뒤 요소를 삭제하지 않고 반환
  - `is_full()`: deque 가 꽉 차있으면 TRUE, 아니면 FALSE
  - `is_empty()`: deque 가 비어있으면 TRUE, 아니면 FALSE



덱의 연산


전단
후단
add_front(A):
A
add_rear(B):
A
B
add_front(C):
C
A
B
add_rear(D):
C
A
B
D
delete_front():
C
A
B
D
delete_rear():
A
B
D

원형 덱의 연산



인덱스 회전 방법 (1,2 는 원형 Queue 의 연산과 동일)

1. delete_front() 직후 (=dequeue()): rear <- (rear + 1)%MAX_SIZE
2. add_rear(e) 직후 (=enqueue(e)): front <- (front + 1)%MAX_SIZE
3. delete_rear() 직후: rear <- (rear - 1 + MAX_SIZE)%MAX_SIZE
4. add_front(e) 직후: front <- (front - 1 + MAX_SIZE)%MAX_SIZE

원형 덱의 데이터

- 원형큐와 동일

```
1 Element data[MAX_SIZE]; // 요소의 배열
2 int front; // 전단 인덱스
3 int rear; // 후단 인덱스
```

원형 덱의 연산 구현

- 원형 큐에 없는 연산 (전단 삽입, 후단 삭제, 후단 참조) 만 추가

```
1  void add_front(Element val){ //전단 삽입
2      if (is_full()) error("Overflow Error!");
3      data[front] = val;
4      front = (front - 1 + MAX_SIZE) % MAX_SIZE;
5  }
6  Element delete_rear(){ //후단 삭제
7      if (is_empty() error("Underflow Error!");
8      int prev = rear;
9      rear = (rear - 1 + MAX_SIZE) % MAX_SIZE;
10     return data[prev];
11 }
12 Element get_rear(){ //후단 참조
13     if (is_empty()) error("Underflow Error!");
14     return data[rear];
15 }
```

원형 덱의 연산 구현 (포인터 이용)

```
1  void add_front(Queue *q, Element e){// 전단에 삽입
2    if (is_full(q))
3      error ("overflow");
4    else{
5      q->data[front] = e;
6      q->front = (q->front - 1 + MAX_SIZE)%MAX_SIZE;
7    }
8  }
9  Element delete_rear(Queue *q){ // 후단 삭제
10   if (is_empty(q))
11     error ("underflow");
12   else{
13     Element e = q->data[rear];
14     q->rear = (q->rear - 1 + MAX_SIZE)%MAX_SIZE;
15     return e;
16   }
17 }
```

원형 덱 테스트

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 10
#define Element int
#include "CircularDeque.h"

// 덱 요소의 출력 함수. front+1 부터 size 개의 요소를 순서대로 출력
void print_deque(char msg[]) {
    printf("%s front=%d, rear=%d --> ", msg, front, rear);
    int size = (rear - front + MAX_SIZE) % MAX_SIZE;

    for (int i = front + 1; i <= front + size; i++)
        printf("%2d ", data[i % MAX_SIZE]);
    printf("\n");
}
```

원형 덱 테스트

```
1  void main(){
2    init_deque();
3    print_deque("Initialize Deque");
4    for (int i = 1; i < 10; i++) {
5      if (i % 2) {
6        add_front(i); // i 가 홀수이면 전단으로 삽입
7        print_deque(" 홀수 전단삽입");
8      }
9      else{
10       add_rear(i); // 짝수이면 후단으로 삽입
11       print_deque(" 짝수 후단삽입");
12     }
13   }
14   delete_rear();
15   print_deque("delete_rear() ");
16   delete_front();
17   print_deque("delete_front()");
18 }
```

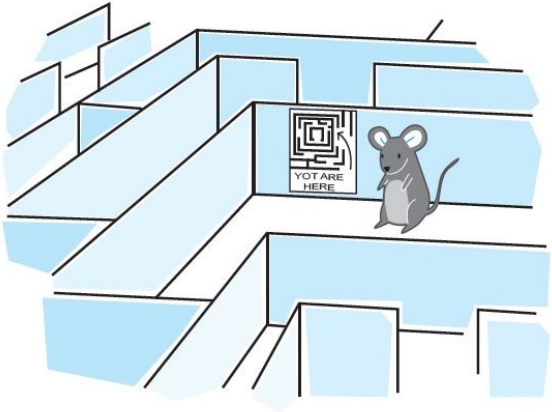
원형 덱 테스트 결과

```
$ gcc CircularDeque.c -o deque
$ ./deque
```

```
Initialize Deque front=0, rear=0 -->
홀수 전단삽입 front=9, rear=0 -->  1
짝수 후단삽입 front=9, rear=1 -->  1  2
홀수 전단삽입 front=8, rear=1 -->  3  1  2
짝수 후단삽입 front=8, rear=2 -->  3  1  2  4
홀수 전단삽입 front=7, rear=2 -->  5  3  1  2  4
짝수 후단삽입 front=7, rear=3 -->  5  3  1  2  4  6
홀수 전단삽입 front=6, rear=3 -->  7  5  3  1  2  4  6
짝수 후단삽입 front=6, rear=4 -->  7  5  3  1  2  4  6  8
홀수 전단삽입 front=5, rear=4 -->  9  7  5  3  1  2  4  6  8
delete_rear() front=5, rear=3 -->  9  7  5  3  1  2  4  6
delete_front() front=6, rear=3 -->  7  5  3  1  2  4  6
```

LAB 덱의 응용: 미로 탐색 (Maze problem)

- 미로에 갇힌 생쥐가 출구를 찾는 문제



LAB 덱의 응용: 미로 탐색

- 시행 착오법 (trial and error)
  - 하나의 경로를 선택해 시도하고 막히면 다시 다른 경로를 시도
  - 저장된 경로를 모두 선택할 수만 있다면 어떤 자료구조에 저장하든지 출구를 찾을 수 있음
  - 가장 대표적인 방법: DFS, BFS
- 깊이 우선 탐색 (DFS, Depth First Search)
  - 가장 **최근에 저장한 경로**를 선택하여 다시 시도
  - 스택 (stack) 을 이용해 구현
- 너비 우선 탐색 (BFS, Breadth First Search)
  - 가장 **먼저 저장된 경로**를 선택하여 다시 시도
  - 큐 (queue) 를 이용해 구현
- DFS 와 BFS 는 stack 또는 queue 를 사용, 이 두가지 방법을 모두 구현하려면 **deque**를 사용하여 구현 가능함

dongguk UNIVERSITY WISE

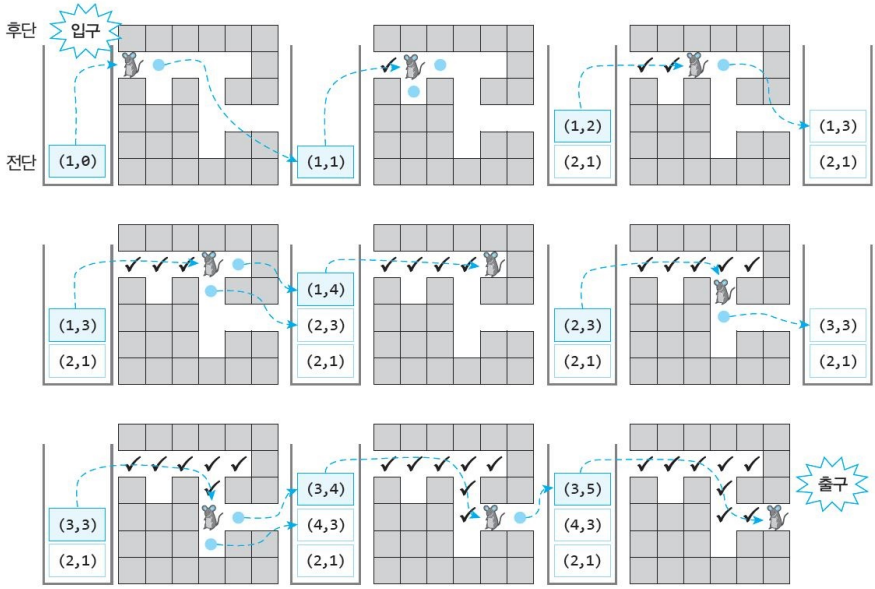
LAB 덱의 응용: DFS & BFS

```
1  DFS()
2      init_deque()
3      add_rear(입구좌표)
4      while is_empty() == FALSE :
5        현재위치 = delete_rear() // 스택에서 후단 위치를 꺼냄
6        if 현재위치 == 출구위치 :
7          return "Success" // 출구탐색 성공
8        for 이웃위치에 대하여(LEFT, RIGHT, UP, DOWN) :
9          if (이웃위치 방문하지 않은 갈 수 있는 위치) :
10           add_rear(이웃위치) // 후단 삽입
11     return "Fail" // 출구탐색 실패
```

BFS 는 queue 를 사용하므로 위의 DFS 에서 5 번 행을 아래로 바꾸면 됨

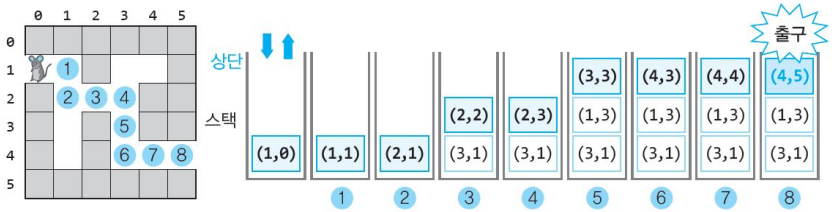
```
현재위치 = delete_front() // 큐에서 전단 위치를 꺼냄
```

LAB 덱의 응용: DFS

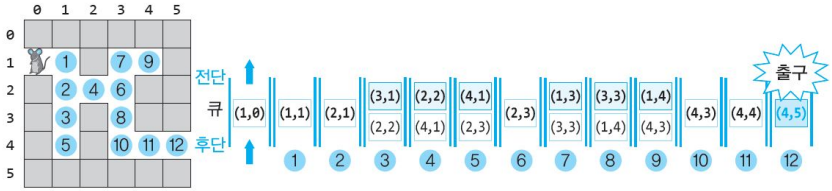

후단
입구
전단
(1,0)
(1,1)
(1,2)
(2,1)
(1,3)
(2,1)
(1,3)
(2,1)
(1,4)
(2,3)
(2,1)
(2,3)
(2,1)
(3,3)
(2,1)
(3,3)
(2,1)
(3,4)
(4,3)
(2,1)
(3,5)
(4,3)
(2,1)
출구

LAB 덱의 응용: DFS & BFS

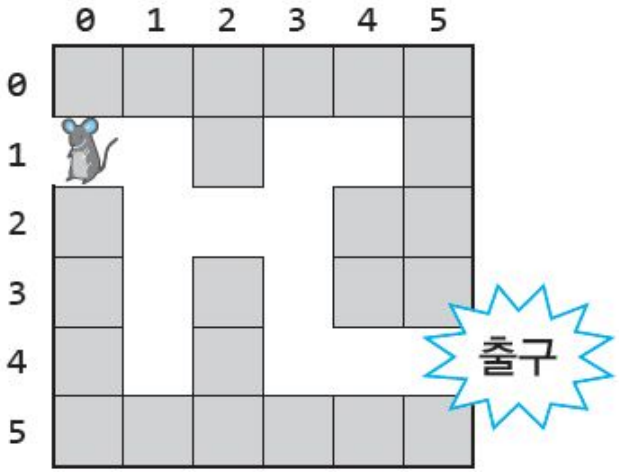
- DFS with stack

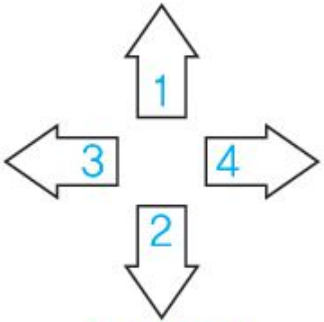


- BFS with queue



LAB 미로탐색





LAB DFS 미로 탐색

- stack size : MAX_SIZE
- 위치 (position) 정보: 행과 컬럼 인덱스 (Element)
- 미로 맵 (map) 2 차원 배열로'0' 은 접근가능, '1' 은 접근불가

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
struct Pos2D { int x; int y; };
typedef struct Pos2D Element;
#include "CircularDeque.h"
#define MAZE_DIM 6
char map[MAZE_DIM][MAZE_DIM] = {
    { '1', '1', '1', '1', '1', '1' },
    { '0', '0', '1', '0', '0', '1' },
    { '1', '0', '0', '0', '1', '1' },
    { '1', '0', '1', '0', '1', '1' },
    { '1', '0', '1', '0', '0', 'x' }, // 'x' (탈출위치)
    { '1', '1', '1', '1', '1', '1' },
};
```

LAB DFS 미로 탐색

```
int jump_pos[4][2] = {{-1, 0}, // Left
                      {1, 0}, // Right
                      {0, 1}, // Up
                      {0, -1} // Down
  };
```

LAB DFS 미로 탐색

- stack 연산: 삽입 (push_loc), 꺼내기 (pop_loc)

```
1  void push_loc(int x, int y) {
2      if (x < 0 || y < 0 || x >= MAZE_DIM || y >= MAZE_DIM)
3          return; // 미로 범위 밖의 위치
4      if (map[x][y] != '0' && map[x][y] != 'x')
5          return; // 접근불가 셀
6      Element pos = {x, y};
7      add_rear(pos);
8  }
9  Element pop_loc() {
10     return delete_rear();       // 후단 삭제 (스택의 pop)
11 }
12 int is_valid(int x, int y) {
13     return ((x >= 0) && (x < MAZE_DIM) && (y >= 0) && (y < MAZE_DIM) && (map[x][y] != '.'));
14 }
15 int is_dest(int x, int y) {
16     return (map[x][y] == 'x');
17 }
```

LAB DFS 미로 탐색

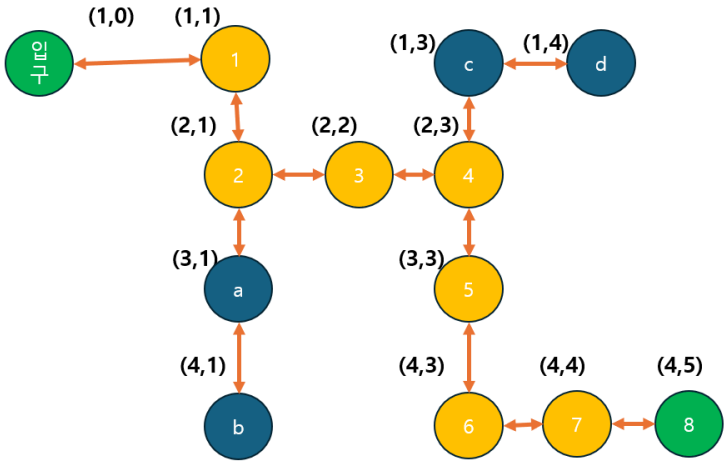
```
int DFS(){
    while (is_empty() == 0 ){
      Element hear = pop_loc(); // 스택에서 후단 위치를 꺼냄
      if (is_dest(hear.x, hear.y)) { // 출구확인
        return(1); // 출구탐색 성공
      }
      map[hear.x][hear.y] = '.';
      for (int i = 0; i < 4; i++) { //이웃위치 (LEFT, RIGHT, UP, DOWN)
        int x = hear.x + jump_pos[i][0];
        int y = hear.y + jump_pos[i][1];
        if(is_valid(x, y) && !is_dest(x, y)){
          push_loc(x, y);  // 스택의 후단 삽입
        }
      }
    }
    return (0); // 출구탐색 실패
}
```

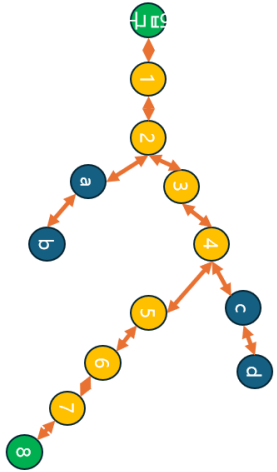
LAB DFS 미로 탐색 (main 함수)

```
void main()
{
    init_deque();
    push_loc(1, 0); //입구좌표
    if(DFS()){
      printf("\n미로 탈출 성공\n");
    }else{
      printf("\n미로 탈출 실패\n");
    }
}
```

LAB DFS 미로 탐색 그래프

- 미로 탐색은 그래프/트리에서 순회 또는 탐색 문제와 유사함





연습문제

- 연습문제: 4.1 ~ 4.17
-