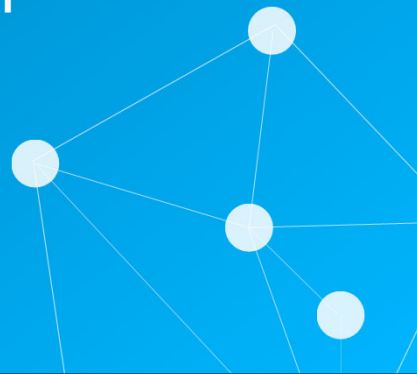


04 CHAPTER

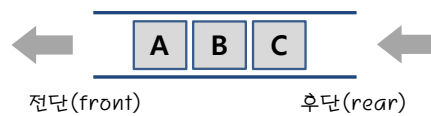
큐



큐(Queue)



- 큐: 먼저 들어온 데이터가 먼저 나가는 자료구조
- 선입선출(FIFO: First-In First-Out)
 - (예)매표소의 대기열



큐 ADT

- 삽입과 삭제는 FIFO순서를 따른다.
- 삽입은 큐의 후단에서, 삭제는 전단에서 이루어진다.

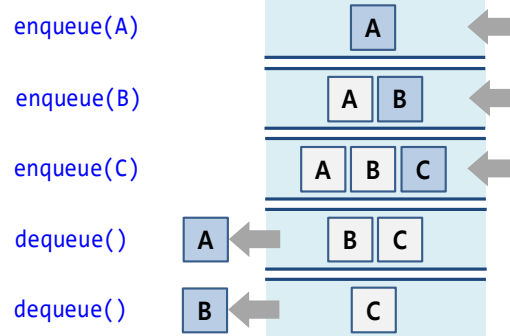
데이터: 선입선출(FIFO)의 접근 방법을 유지하는 요소들의 모임

연산:

- `init()`: 큐를 초기화한다.
- `enqueue(e)`: 주어진 요소 `e`를 큐의 맨 뒤에 추가한다.
- `dequeue()`: 큐가 비어있지 않으면 맨 앞 요소를 삭제하고 반환한다.
- `is_empty()`: 큐가 비어있으면 `true`를 아니면 `false`를 반환한다.
- `peek()`: 큐가 비어있지 않으면 맨 앞 요소를 삭제하지 않고 반환한다.
- `is_full()`: 큐가 가득 차 있으면 `true`를 아니면 `false`를 반환한다.
- `size()`: 큐의 모든 요소들의 개수를 반환한다.

3

큐의 연산



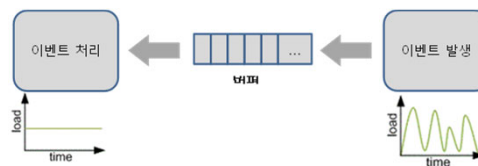
< 큐의 삽입과 삭제 >

4

큐의 응용

- 직접적인 응용

- 시뮬레이션의 대기열(공항의 비행기들, 은행에서의 대기열)
- 실시간 비디오 스트리밍
- 프린터와 컴퓨터 사이의 버퍼링



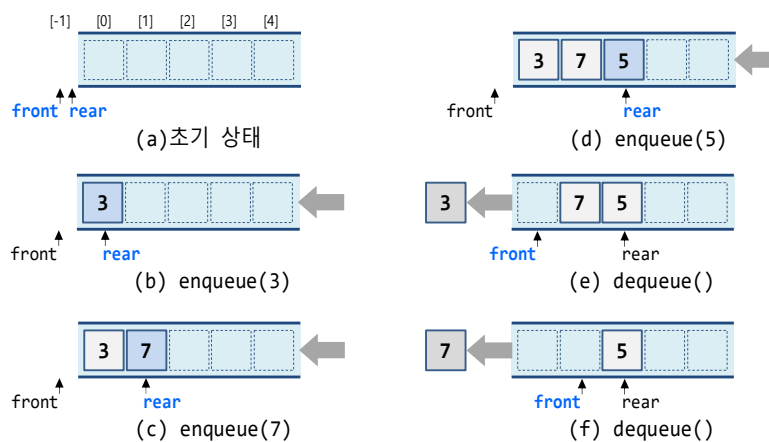
- 간접적인 응용

- 스택과 마찬가지로 프로그래머의 도구
- 많은 알고리즘에서 사용됨

5

배열을 이용한 큐 : 선형큐

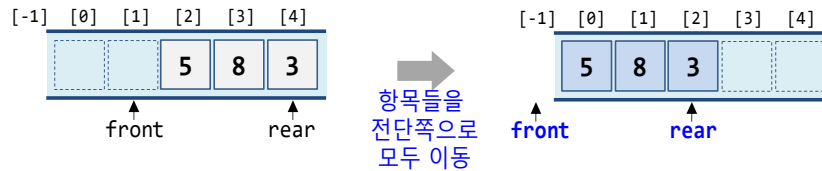
- 배열을 선형으로 사용하여 큐를 구현



6

배열을 이용한 큐 : 선형큐

- 삽입을 계속하기 위해서는 요소들을 이동시켜야 함



7

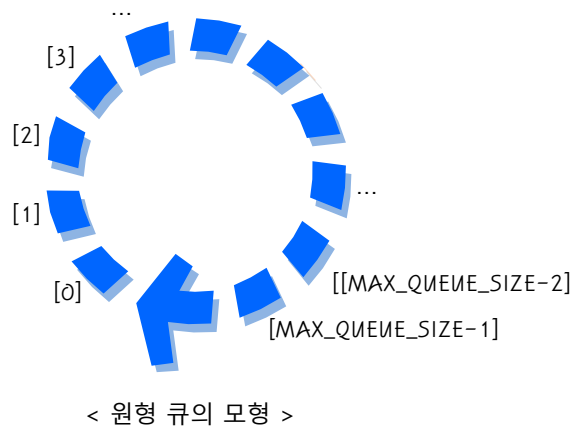
선형큐의 문제점

- 큐의 단점
 - rear가 배열의 마지막 원소를 가리키는 경우 ($\text{rear} == \text{MAXQUEUE}-1$)가 반드시 MAXQUEUE개의 항목이 차 있는 것을 의미하지 않음.
 - $\text{front} == -1$, $\text{rear} == \text{MAXQUEUE}-1$ 이면 오버플로우
 - front 가 -1보다 크면 데이터 항목이 삭제된 것이므로 큐에 빈 공간이 남아 있음.
 - 계속 항목을 삭제하면 rear pointer와 front pointer가 만나게 되고 공백 큐가 되는데도 오버플로우 현상 발생.
- 해결책
 - 삭제시마다 큐의 데이터 이동(overhead)
 - ($\text{rear} == \text{MAXQUEUE}-1$) 시에 큐 전체를 왼쪽으로 옮기고 front를 -1로 하고 rear의 값을 조정(여전히 overhead)
 - 원형 큐
 - 큐의 배열을 선형으로 표현하지 않고 원형으로 표현하는 방법.

8

해결방법 → 원형큐

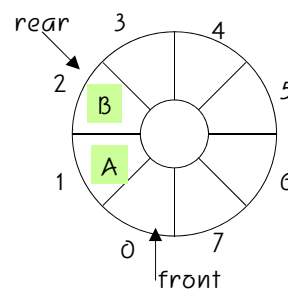
- 원형큐: 배열을 원형으로 사용하여 큐를 구현



9

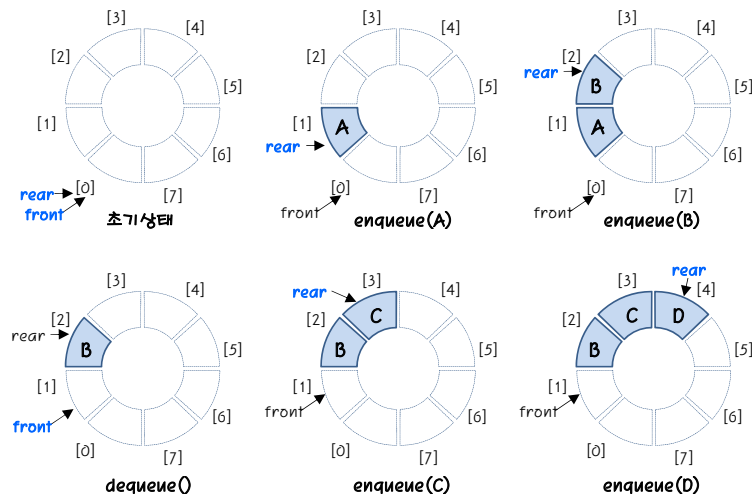
원형큐의 구조

- 전단과 후단을 관리하기 위한 2개의 변수 필요
 - front: 첫번째 요소 하나 앞의 인덱스
 - rear: 마지막 요소의 인덱스



10

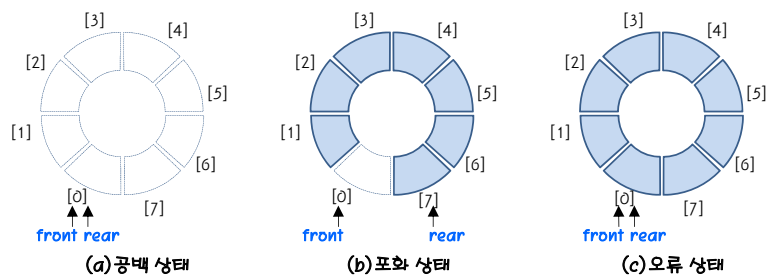
원형큐의 삽입과 삭제연산 예



11

공백상태, 포화상태

- 공백상태: $front == rear$
- 포화상태: $front \% M == (rear + 1) \% M$
- 공백상태와 포화상태를 구별 방법은?
 - 하나의 공간은 항상 비워둠(해결방안?)



12

큐의 연산



- 나머지(modulo) 연산을 사용하여 인덱스를 원형으로 회전시킨다.
- 삽입 연산

enqueue(x)

```
rear ← (rear+1) mod MAX_QUEUE_SIZE;  
data[rear] ← x;
```

- 삭제 연산

dequeue()

```
front ← (front+1) mod MAX_QUEUE_SIZE;  
return data[front];
```

13

원형 큐의 구현



- 원형 큐를 위한 데이터
 - int 큐에서 Element는 int로 지정
 - 전역변수 사용

```
#define MAX_QUEUE_SIZE 100  
#define Element int  
  
Element data[MAX_QUEUE_SIZE];  
int front;  
int rear;
```

- 원형 큐의 단순한 연산들

```
void init_queue() { front = rear = 0; ; }  
int is_empty() { return front == rear;; }  
int is_full() { return (rear+1)%MAX_QUEUE_SIZE == front; }  
int size() { return (rear-front+MAX_QUEUE_SIZE)%MAX_QUEUE_SIZE; }
```

14

원형 큐의 구현



```
void enqueue( Element val )
{
    if( is_full() )
        error(" 큐 포화 에러");
    rear = (rear+1) % MAX_QUEUE_SIZE;
    data[rear] = val;
}
Element dequeue( )
{
    if( is_empty() )
        error(" 큐 공백 에러");
    front = (front+1) % MAX_QUEUE_SIZE;
    return data[front];
}
Element peek( )
{
    if( is_empty() )
        error(" 큐 공백 에러");
    return data[(front+1) % MAX_QUEUE_SIZE];
}
```

15

사용방법



```
void main()
{
    int i;
    init_queue( );
    for( i=1 ; i<10 ; i++ )
        enqueue( i );
    print_queue("선형큐 enqueue 9회");
    printf("\tdequeue() --> %d\n", dequeue());
    printf("\tdequeue() --> %d\n", dequeue());
    printf("\tdequeue() --> %d\n", dequeue());
    print_queue("선형큐 dequeue 3회");
}
```

16

사용방법

```

선택 C:\WINDOWS\system32\cmd.exe
선행큐 enqueue 9회[ 9]= 1 2 3 4 5 6 7 8 9
dequeue() --> 1
dequeue() --> 2
dequeue() --> 3
선행큐 dequeue 3회[ 6]= 4 5 6 7 8 9
계속하려면 아무 키나 누르십시오 . . .
  
```

9번 enqueue() 연산 후

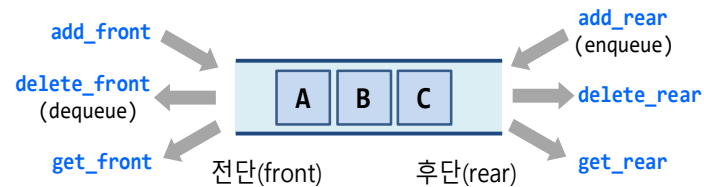
3번 dequeue() 연산 후

설명 출력 현재 항목 수 큐 내용

17

덱(deque)

- 덱(deque)은 double-ended queue의 줄임말
 - 전단(front)과 후단(rear)에서 모두 삽입과 삭제가 가능한 큐



< 덱의 구조 >

18

덱 ADT

- 큐와 데이터는 동일
 - 연산은 추가됨

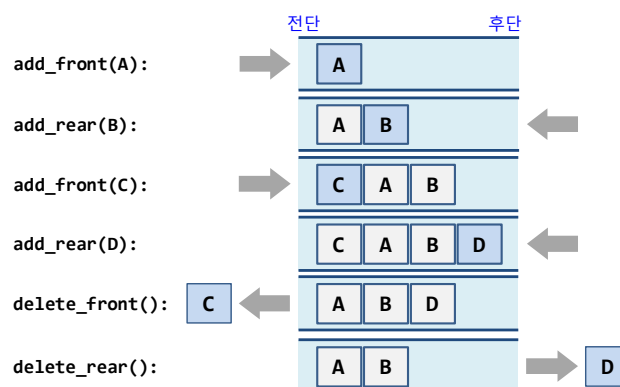
데이터: 전단과 후단을 통한 접근을 허용하는 요소들의 모임

연산:

- init(): 덱을 초기화한다.
- add_front(e): 주어진 요소 e를 덱의 맨 앞에 추가한다.
- delete_front(): 전단 요소를 삭제하고 반환한다.
- add_rear(e): 주어진 요소 e를 덱의 맨 뒤에 추가한다.
- delete_rear(): 후단 요소를 삭제하고 반환한다.
- is_empty(): 공백 상태이면 TRUE를 아니면 FALSE를 반환한다.
- get_front(): 전단 요소를 삭제하지 않고 반환한다.
- get_rear(): 후단 요소를 삭제하지 않고 반환한다.
- is_full(): 덱이 가득 차 있으면 TRUE를 아니면 FALSE를 반환한다.
- size(): 덱 내의 모든 요소들의 개수를 반환한다.

19

덱의 연산



<선형 덱의 연산 >

- 선형 덱의 문제점
 - : 선형 큐의 문제와 동일한 문제 발생(overflow)

20

원형 덱의 연산

- 큐와 데이터는 동일함
- 연산은 유사함.
- 큐와 알고리즘이 동일한 연산
 - `init_deque()` : 원형큐의 `init_queue()`
 - `print_deque()`: 원형큐의 `print_queue()`
 - `add_rear()`: 원형큐의 `enqueue()`
 - `delete_front()`: 원형큐의 `dequeue()`
 - `get_front()`: 원형큐의 `peek()`

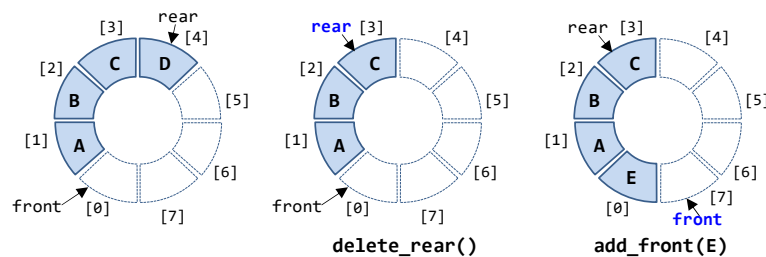
21

원형 덱의 연산

- 덱에서 추가된 연산
 - `delete_rear()`, `add_front()`, `get_rear()`
 - 반대방향의 회전이 필요

```
front ← (front-1+MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE;
rear ← (rear -1+MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE;
```

– 예:



22

원형 덱의 구현

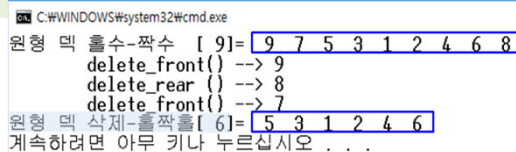
```
void init_deque( ) { init_queue( ); }
void add_rear(Element val) { enqueue( val); }
Element delete_front( ) { return dequeue(); }
Element get_front( ) { return peek(); }
void print_queue(char msg[]) { print_queue(msg); }

void add_front( Element val ) {
    if( is_full( ) )
        error(" 덱 포화 에러");
    data[front] = val;
    front = (front-1+MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE;
}
Element delete_rear( ) {
    Element ret;
    if( is_empty( ) )
        error(" 덱 공백 에러");
    ret = data[rear];
    rear = (rear-1+MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE;
    return ret;
}
Element get_rear( ){
    if( is_empty( ) )
        error(" 덱 공백 에러");
    return data[rear];
}
```

23

원형 덱 테스트 프로그램

```
void main()
{
    int i;
    init_deque( );
    for( i=1 ; i<10 ; i++ ) {
        if( i % 2 ) add_front( i );
        else add_rear( i );
    }
    print_queue("원형 덱 홀수-짝수 ");
    printf("\ndelete_front() --> %d\n", delete_front());
    printf("\ndelete_rear () --> %d\n", delete_rear ());
    printf("\ndelete_front() --> %d\n", delete_front());
    print_queue("원형 덱 삭제-홀짝홀 ");
}
```



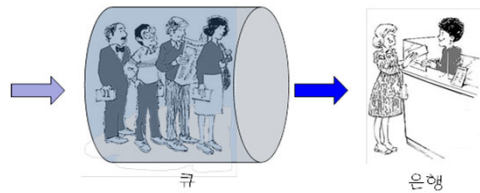
```
C:\WINDOWS\system32\cmd.exe
원형 덱 홀수-짝수 [ 9] = 9 7 5 3 1 2 4 6 8
delete_front() --> 9
delete_rear () --> 8
delete_front() --> 7
원형 덱 삭제-홀짝홀 [ 6] = 5 3 1 2 4 6
계속하려면 아무 키나 누르십시오 . . .
```

24

큐의 응용 : 은행 시뮬레이션



- 은행 시뮬레이션
 - 큐잉이론에 따라 시스템의 특성을 시뮬레이션하여 분석하는 데 이용
 - 큐잉모델은 고객에 대한 서비스를 수행하는 서버와 서비스를 받는 고객들로 이루어진다
 - 고객이 들어와서 서비스를 받고 나가는 과정을 시뮬레이션
 - 고객들이 기다리는 평균시간을 계산



25

큐의 응용 : 은행 시뮬레이션



- 입력:
 - 시뮬레이션 할 최대 시간 (예: 10 [단위시간])
 - 단위시간에 도착하는 고객 수 (예: 0.5 [고객수/단위시간])
 - 한 고객에 대한 최대 서비스 시간 (예: 5 [단위시간/고객])
- 출력:
 - 고객들의 평균 대기시간
- 서비스 인원(은행원): 1명
- 고객 정보:
 - 단위시간에 도착하는 고객 수를 바탕으로 무작위로 발생
 - 서비스 시간: 일정한 범위 내에서 무작위로 결정

26

실행결과 예

```

C:\WINDOWS\system32\cmd.exe
시뮬레이션 할 최대 시간 (예:10) = 10
단위시간에 도착하는 고객 수 (예:0.5) = 0.5
한 고객에 대한 최대 서비스 시간 (예:5) = 5

=====
현재시각=1
고객 1 방문 (서비스 시간:5분)
고객 1 서비스 시작 (대기시간:0분)
현재시각=2
현재시각=3
고객 2 방문 (서비스 시간:4분)
현재시각=4
현재시각=5
현재시각=6
고객 3 방문 (서비스 시간:1분)
고객 2 서비스 시작 (대기시간:3분)
현재시각=7
고객 4 방문 (서비스 시간:5분)
현재시각=8
현재시각=9
고객 5 방문 (서비스 시간:1분)
현재시각=10
고객 6 방문 (서비스 시간:2분)
고객 3 서비스 시작 (대기시간:4분)
=====
서비스 받은 고객수 = 3
전체 대기 시간 = 7분
서비스고객 평균대기시간 = 2.33 분
현재 대기 고객 수 = 3
계속하려면 아무 키나 누르십시오 . . .
  
```

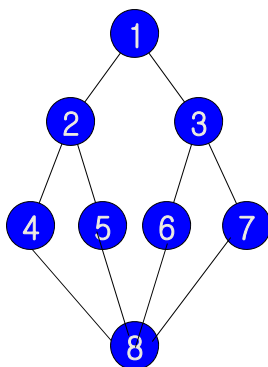
시뮬레이션 파라미터 입력

각 단위 시간별 이벤트 출력

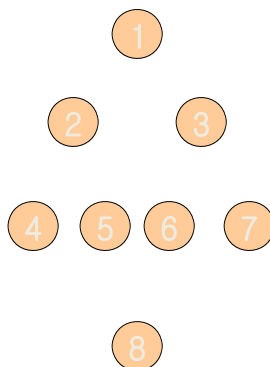
시뮬레이션 결과 출력

27

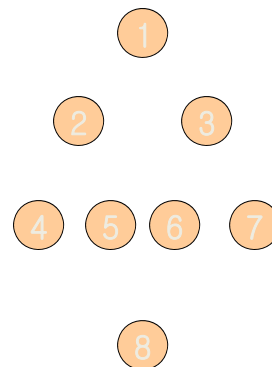
그래프의 탐색(DFS, BFS)



< 그래프의 예 >



D F S 탐색



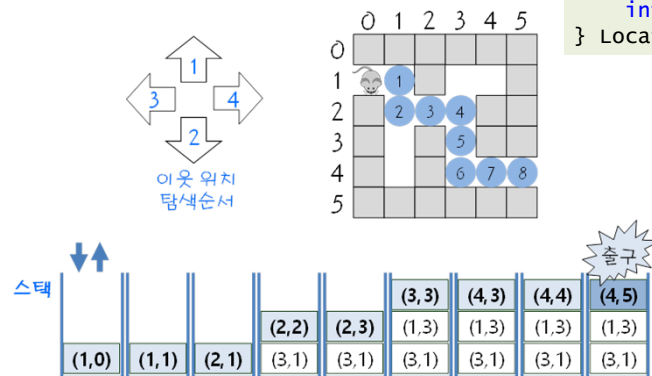
B F S 탐색

28

스택의 응용 : 미로 탐색 DFS

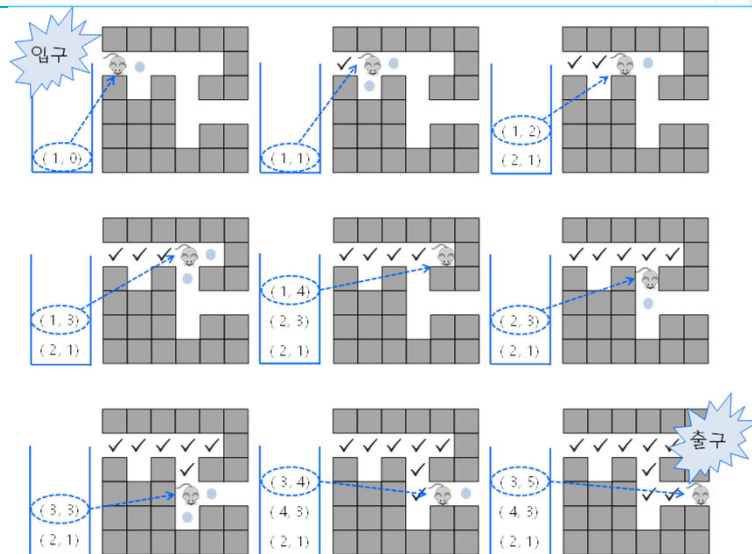
- 깊이 우선 탐색(DFS, Depth First Search)

```
typedef struct {
    int x;
    int y;
} Location2D;
```



29

미로 탐색 DFS



30

```

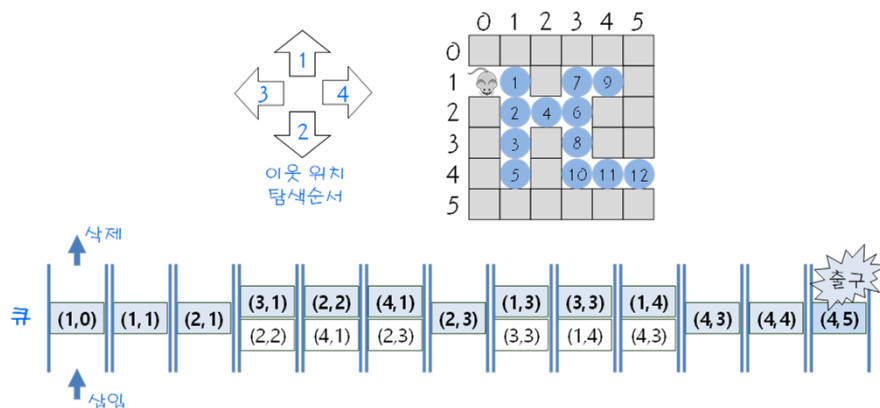
int DFS()
{
    int x, y;
    Location2D here;
    init_deque( );
    add_rear( getLocation2D(0,1) );
    printf("DFS: ");
    while ( is_empty() == 0 ) {
        here = delete_rear( );
        print_elem( here );
        x = here.x;
        y = here.y;
        if( map[y][x] == 'x' ) return 1;
        else {
            map[y][x] = '.';
            if( is_valid( x-1, y ) ) add_rear(getLocation2D(x-1,y));
            if( is_valid( x+1, y ) ) add_rear(getLocation2D(x+1,y));
            if( is_valid( x, y-1 ) ) add_rear(getLocation2D(x,y-1));
            if( is_valid( x, y+1 ) ) add_rear(getLocation2D(x,y+1));
        }
    }
    return 0;
}

```

31

큐의 응용 : 미로 탐색 BFS

- 너비 우선 탐색(BFS, Breadth First Search)



32

4장 정리



- 큐란?
 - FIFO 구조
 - front, rear, enqueue(), dequeue()
- 문제점과 원형큐
 - 선형 큐에서 공간활용
 - % 연산사용
- 덱와 원형덱
- 미로찾기(DFS, BFS) 응용