5 장. 포인터와 연결된 구조

Jinseog Kim

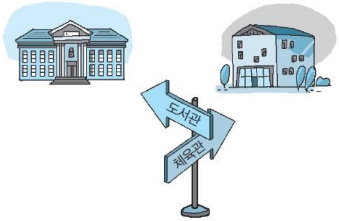
컴퓨터 공학과
동국대학교 WISE 캠퍼스

학습 내용

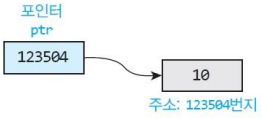
1. Pointer 의 개념과 사용법
2. 동적 메모리 할당 및 해제
3. (단순) 연결구조의 이해
4. 배열구조와 연결구조 비교
5. 연결구조의 응용 및 구현 (스택, 큐, 덱)

포인터와 동적 메모리 할당

- 포인터 (pointer): 메모리의 주소를 저장하는 변수

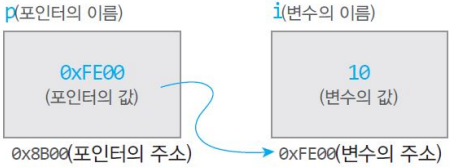



```
int n = 10;
int* ptr = &n;
```



- 포인터 선언 예

```
int i = 10;
int* p;
p = &i;
```



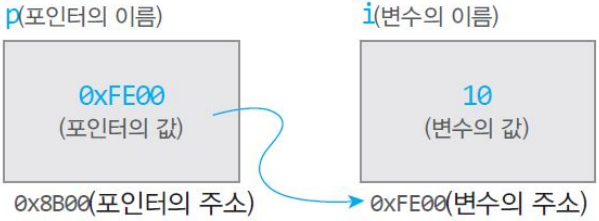
포인터 사용 예시

- 역참조 연산자 * 사용
- *p 는 p 가 가리키는 주소의 내용
- &i 는 변수 i 의 메모리 주소

```
1  int *p = 20; // 포인터 p 를 선언하고 그 주소에 20 을 저장
2
3  c = a * b; // 곱셈연산
4  int *p;    // 포인터 변수 선언
5  *p = 20;   // 포인터의 역참조
6
7  c = a & b;  // 비트연산 AND
8  p = &i;      // 변수 i 의 주소 추출
9  int &p = i;  // 변수 i 의 참조자 선언 (C++ 에서만)
```

이중 포인터

```
int i = 10;
int* p;
p = &i;
```



- 다양한 자료형에 대한 포인터

```
1 float * pf = NULL; // float 형 변수의 포인터
2 char * pc; // char 형 변수의 포인터
3 int ** pp; // 포인터 변수를 저장하기 위한 포인터
4 Polynomial * poly; // 구조체의 포인터
```

배열, 구조체와 포인터

- 배열 이름은 상수 포인터

```
int  A[5], *p = NULL; // 배열 A 와 포인터 p 선언
p = A; // p 에 배열 A 의 주소 (= 배열이름) 를 복사
p[3] = 20; // p[3] 은 A[3] 과 동일
```

- 구조체의 포인터

```
typedef struct{
  int degree;
  cloat coef[MAX_DEGREE];
}Polynomial;

Polynomial a;
Polynomial * p=NULL;
p = &a;
a.degree = 5;  // 구조체 a 를 통한 접근 . 이용
p->coef[0] = 1; // 포인터 p 를 통한 접근 -> 이용
```

- **자체 참조 구조체 (self-referential structure)** 자기와 같은 구조체를 가리키는 포인터를 한 개 이상 멤버로 갖는 특별한 구조체

```c
typedef struct ListNode{
  char data[10];
  struct ListNode * link; //자신과 같은 구조체를 가리키는 포인터
};
```

동적 메모리 할당/해제

- 동적 메모리 할당 및 해제
  - 프로그램 실행 중에 필요한 만큼 메모리를 운영체제에서 할당
  - 사용이 끝나면 메모리를 반드시 반납해야 함 (메모리 누수방지를 위해서)
- 동적 메모리 할당 및 해제 함수
  - malloc(): 바이트 (byte) 단위로 할당하고 시작 주소를 반환
  - free(): 할당되었던 메모리 블록을 시스템에 반납

```
1  int * data=NULL;  // 포인터 선언
2  data = (int *)malloc(sizeof(int)*100); // 메모리 할당
3
4  *data = 10; // data[0] = 10 과 동일
5  data[3] = 30;
6
7  free(data); // 동적메모리 해제 (반납)
```

동적 메모리 할당

- malloc() 으로 동적으로 할당한 메모리를 사용하다가 용량이 부족하면?
  - realloc() 함수를 사용할 수 있음

```
1  int *  data=NULL;
2  data = (int *)malloc(sizeof(int)*100);
3  data[99] = 99;
4
5  data = (int *)realloc(data, sizeof(int)*200); // 100 -> 200 재할당
6  data[101] = 101;
7
8  free(data); // 동적메모리 해제 (반납)
```

동적 할당을 이용한 배열 구조의 스택

- 스택 데이터 정의

```
int MAX_SIZE = 100; // 스택의 크기를 변수로 선언
Element * data = NULL;
int top;
```

- 스택 초기화 함수: 동적 할당

```
void init_stack(){
  data = (Element *)malloc(sizeof(Element)*MAX_SIZE); // 동적 할당
  top = -1;
}
```

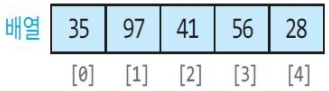
동적 할당을 이용한 배열 구조의 스택

- 삽입 연산 수정

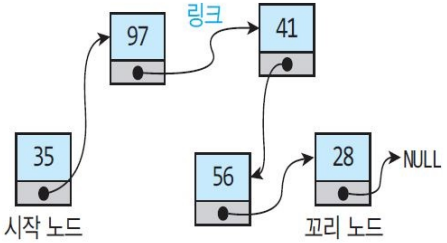
```
1  void push(Element e){
2    if ( is_full() ){ // 포화상태이면 스택 사이즈 증가시켜 할당
3      MAX_SIZE *= 2; // 용량을 2 배로 늘임
4      data = (Element *) realloc(data, sizeof(Element) * MAX_SIZE);
5      printf("realloc(%d)\n", MAX_SIZE);
6    }
7    data[++top] = e;
8  }
9  // 나머지 연산들은 이전과 동일
10 Element pop() ...
11 Element peek() ...
12 int is_empty() ...
13 int is_full() ...
```

연결된 구조 (linked structure) 란?

- **배열구조:** 자료를 연속된 메모리에 저장
- **연결된 구조:** 요소들을 메모리의 빈 공간에 흩어진 형태로 저장
  - 흩어진 요소들을 포인터 변수인 링크 (link)로 연결



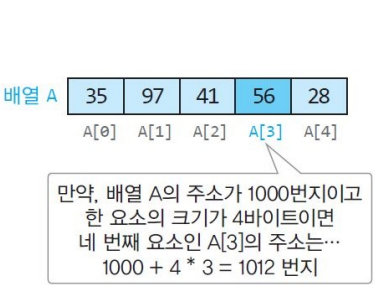
(a) 배열 구조



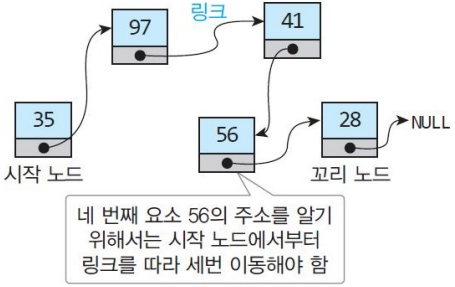
(b) 연결된 구조

배열 구조와 연결된 구조의 비교

- k 번째 요소에 대한 접근
  - 배열구조: 직접 접근 가능, $O(1)$
  - 연결된 구조: 시작노드에서 링크를 따라 k 번 이동, $O(k)$



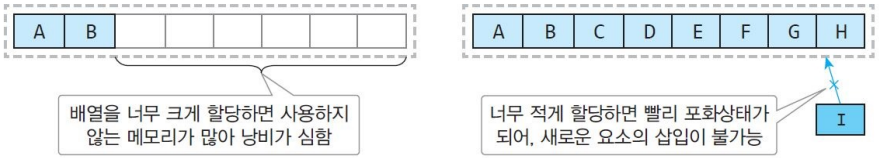
(a) 배열 구조



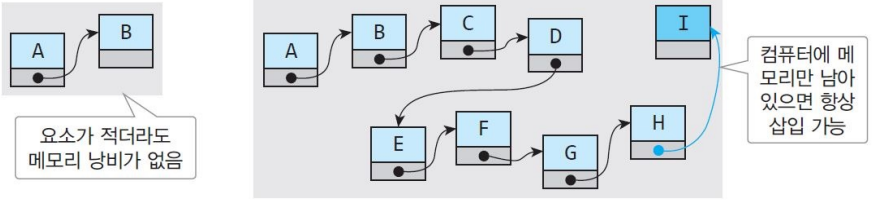
(b) 연결된 구조

용량 변경

- 배열 구조

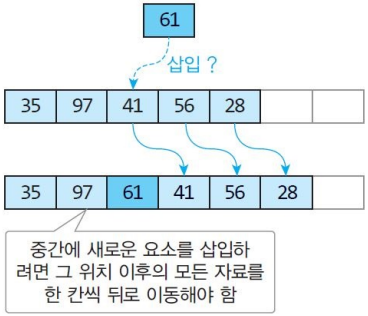


- 연결된 구조

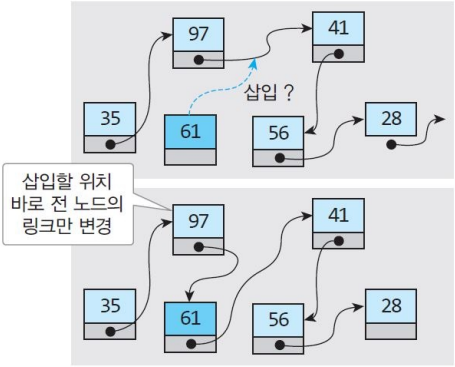


중간에 자료를 삽입하는 연산

- 배열구조에 비해 연결된 구조가 효율적



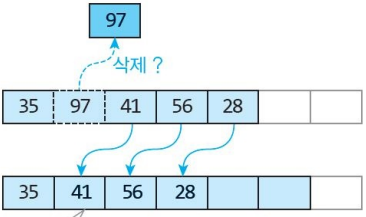
(a) 배열 구조의 삽입 연산



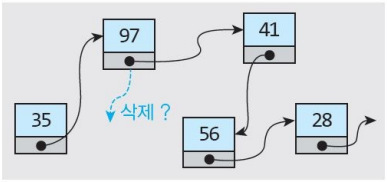
(b) 연결된 구조의 삽입 연산

중간에서 자료를 꺼내는 연산

- 배열구조에 비해 연결된 구조가 효율적



(a) 배열 구조의 삭제 연산



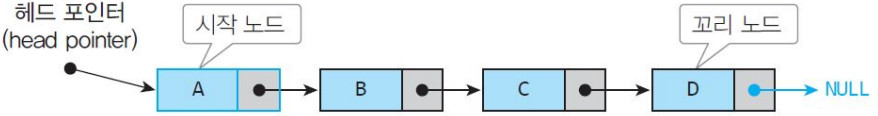
(b) 연결된 구조의 삭제 연산

연결된 구조의 용어

- **노드 (node)**: 데이터 필드 (data field) 와 링크 필드 (link field) 로 구성

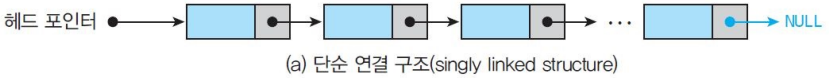
(a) 단순 연결을 위한 노드 (b) 이중 연결을 위한 노드

- **헤드 포인터 (head pointer)**: 시작 노드의 주소를 저장하는 위한 포인터

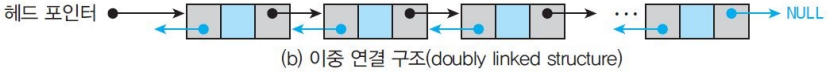


연결된 구조의 종류

- 연결방법에 따라: 단순 연결 (singly linked), 이중 연결 (doubly linked)

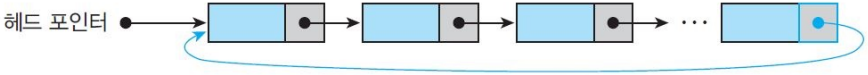


(a) 단순 연결 구조(singly linked structure)

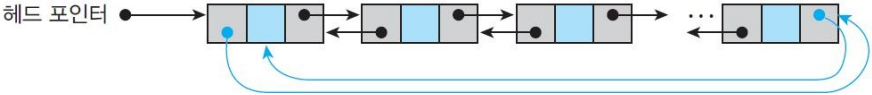


(b) 이중 연결 구조(doubly linked structure)

- 원형 연결 (circular linked): 마지막 노드의 링크가 첫 노드를 가리킴



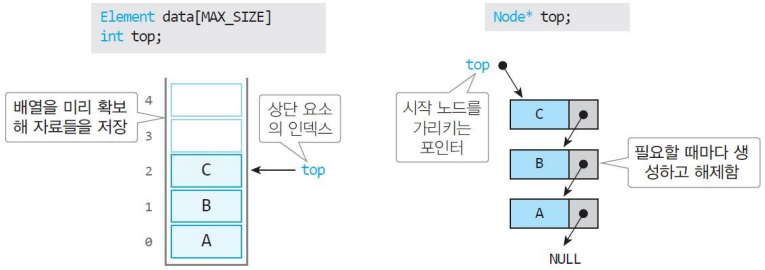
(a) 원형 연결 구조(circular linked structure)



(b) 원형 이중 연결 구조(circular doubly linked structure)

단순 연결 구조 응용: 스택

- 연결된 스택 (linked stack) 의 데이터



- 노드 구조체

```
1  typedef struct Node{
2    Element data;
3    struct Node * link;
4  }Node;
```

단순 연결 구조 응용: 연결스택의 연산

- 스택의 초기화

```
1  init()
2    top <- NULL
```

- 스택의 상태 검사

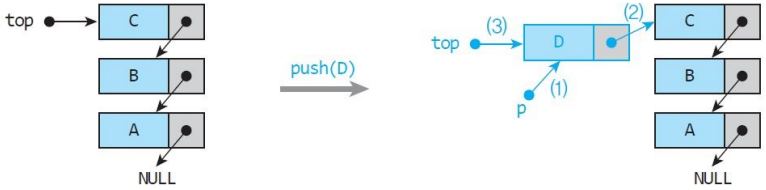
```
1  is_empty()
2    return top == NULL
```

```
1  is_full() // 필요시 노드를 생성할 수 있으므로
2    return FALSE
```

- 상단 요소를 참조하는 peek() 연산

```
1  peek()
2    if is_empty():
3      error "underflow"
4    else:
5      return top.data
```

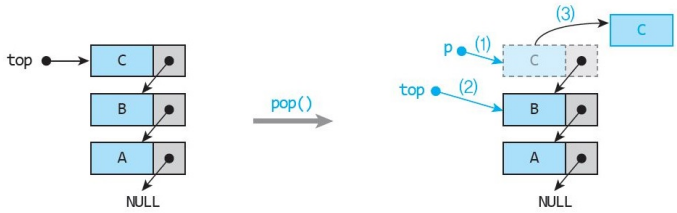
- push(D) 연산: 새로운 요소 D 를 삽입




```
1  push(D):
2    p = alloc_node(D) // (1) 노드 할당 (생성)
3    p.link = top      // (2) 생성노드의 link 는 top 을 가리킴
4    top = p           // (3) top 은 생성노드 (p) 를 가리킴
```

단순 연결 구조 응용: 연결스택의 연산

- 상단 요소를 삭제하는 pop() 연산




```
1  pop():
2    if (is_empty()):
3      error "underflow"
4    else:
5      p = top              // (1) 삭제할 상단 노드 (top) 를 가리킴
6      top <- p.link        // (2) top 은 삭제 노드의 link 를 가리킴
7      return free_node(p) // (3) p 의 data 를 반환하고 메모리 해제
```

연결된 스택의 구현

```
1  // 스택의 데이터 (Element 는 미리 정의되어 있어야 하고, MAX_SIZE 는 필요 없음)
2  typedef struct Node {   // 자기참조 구조체
3      Element data;       // 데이터 필드 (스택 요소)
4      struct Node* link;  // 링크 필드
5  } Node;
6  Node* top = NULL;  // 시작노드를 가리키는 포인터
7
8  // 스택의 연산들
9  int is_empty() { return top == NULL; }
10 int is_full() { return 0; }
11 void init_stack() { top = NULL; }
12
13 // 스택의 모든 노드 삭제 (동적 메모리 해제) : 메모리 누수 (leak) 방지
14 void destroy_stack(){
15     while (is_empty() == 0) pop();
16 }
```

연결된 스택의 구현

- 삽입/삭제 연산

```
1  void push(Element e){ // 삽입 연산
2      Node* p = alloc_node(e);// 그림 5.12 의 (1)
3      p->link = top;          // 그림 5.12 의 (2)
4      top = p;                // 그림 5.12 의 (3)
5  }
6
7  Element pop(){ // 삭제 연산
8      if (is_empty())
9          error("Underflow Error!");
10     Node* p = top;          // 그림 5.13 의 (1)
11     top = p->link;          // 그림 5.13 의 (2)
12     return free_node(p);    // 그림 5.13 의 (3)
13 }
```

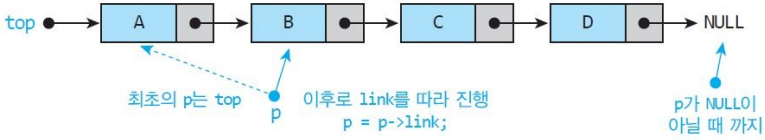
연결된 스택의 활용

- main 함수

```
1  typedef int Element;
2  #include "LinkedStack.h"
3
4  void main(){
5      int A[7] = { 0, 1, 1, 2, 3, 5, 8 };
6
7      init_stack();
8      printf(" 스택 테스트\n 입력 데이터: ");
9      for (int i = 0; i < 7; i++) {
10         printf("%3d", A[i]);
11         push(A[i]);
12     }
13     destroy_stack(); //동적 메모리 해제
14 }
```

Lab: 연결된 스택에서 요소의 수 구하기

- 시작 노드부터 순서대로 모든 노드를 방문해야 함: 시간 복잡도 = O(n)




```
1  // 코드 5.6 연결된 스택의 size() 연산
2  int size()
3  {
4      int count = 0;
5      for (Node* p = top; p != NULL; p = p->link)
6          count++;
7      return count;
8  }
```

Lab: 스택에서 요소 출력

- 입력의 역순으로 출력: 최근 입력 자료 먼저

```c
void print_stack(){
    for (Node* p = top; p != NULL; p = p->link) {
        printf("%3d", p->data);
    }
}
```

- 입력순으로 출력: 먼저 들어온 입력 먼저

```c
void print_recur(Node* p){
    if (p != NULL) {
        print_recur(p->link); // 재귀호출
        printf("%3d", p->data);
    }
}
```

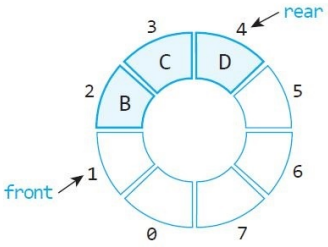
배열과 단순연결구조를 이용한 스택 구현 비교

	고정크기 배열	단순연결구조
장점	1. 구현이 용이 2. 포인터를 사용하지 않아 메모리가 절약	1. 동적 메모리 할당으로 크기 조정 가능 2. 많은 가상머신 (JVM 등) 에 활용 가능
단점	1. 동적크기 조절 불가 2. 사전에 스택의 크기를 정해야 함	1. 포인터를 사용하므로 추가 메모리 필요 2. 임의의 위치를 직접 접근하는 것이 불가능

원형 연결 구조 응용: 큐

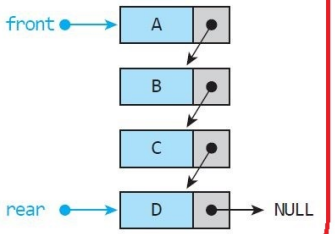
- 배열구조와 단순 연결구조의 큐 (linked queue) 의 구조

```
Element data[MAX_SIZE]
int front;
int rear;
```



(a) 배열 구조의 원형 큐

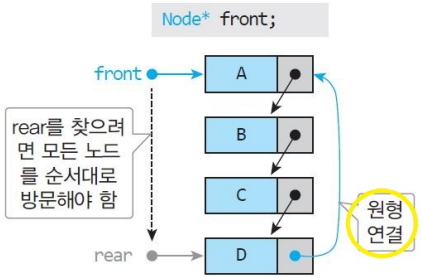
```
Node* front;
Node* rear;
```



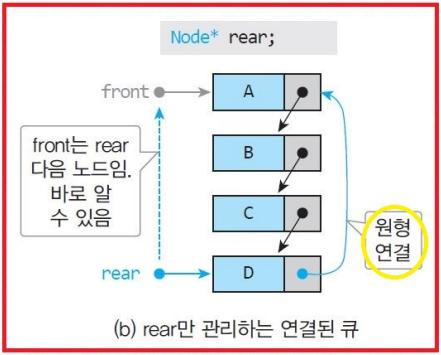
(b) 단순 연결 구조의 큐

원형 연결 구조 응용: 큐

- 연결 큐의 원형 구조
  - rear 의 link 가 시작노드를 가리킴
  - front 또는 rear 둘 중 하나만 관리하면 됨
  - rear 만 관리하는 것이 유리함 (오른쪽 그림)



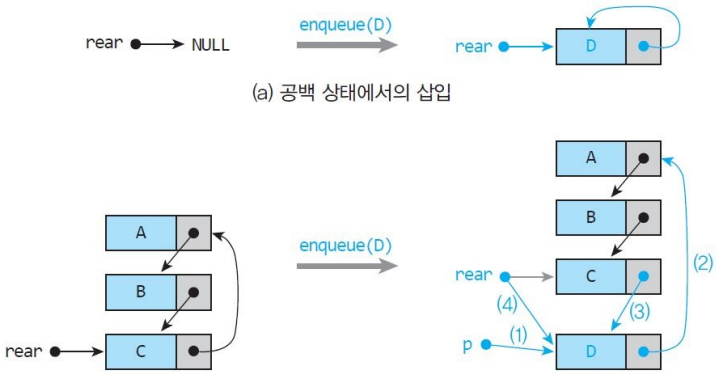
(a) front만 관리하는 연결된 큐



(b) rear만 관리하는 연결된 큐

원형 연결 큐의 연산

- 새로운 요소를 삽입하는 enqueue(e) 연산

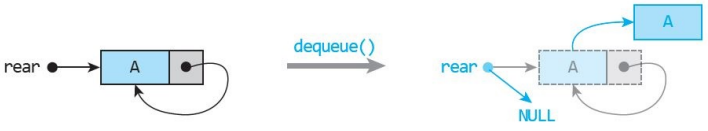


(a) 공백 상태에서의 삽입

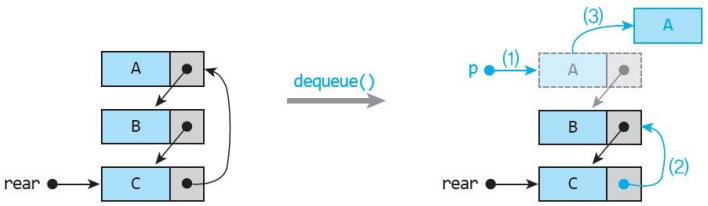
(b) 공백 상태가 아닐 때의 삽입

```
1  enqueue(e)
2    p <- alloc_node(e) //(1)
3    if is_empty() :
4      rear <- p
5      p.link <- p
6    else :
7      p.link <- rear.link //(2)
8      rear.link <- p      //(3)
9      rear <- p           //(4)
```

- 전단 요소를 삭제하는 dequeue() 연산



(a) 노드가 하나인 경우의 삭제



(b) 노드가 둘 이상인 경우의 삭제

```
1  dequeue()
2    if is_empty():
3      error "underflow"
4    else:
5      p <- rear.link     //(1)
6      if rear == p: //노드가 하나
7        rear <- NULL
8      else:  //노드가 둘 이상
9        rear.link <- p.link //(2)
10     return free_node(p)   //(3)
```

연결된 큐의 구현

- 연결 큐에서 데이터 정의
  - 큐의 데이터 (Element 는 미리 정의되어 있어야 함
  - MAX_SIZE 는 필요 없음: 동적 할당

```
1 typedef struct Node {  // 자기참조 구조체
2     Element data;      // 데이터 필드 (스택 요소)
3     struct Node* link; // 링크 필드
4 } Node;
5 Node* rear = NULL;
```

연결된 큐의 구현

- 삽입 연산

```
1  void enqueue(Element e)
2  {
3      Node* p = alloc_node(e); // 그림 5.17 의 (1)
4      if (is_empty()) { // 공백 상태의 삽입
5          rear = p;
6          p->link = p;
7      }
8      else { // 공백이 아닐 때의 삽입
9          p->link = rear->link; // 그림 5.17 의 (2)
10         rear->link = p; // 그림 5.17 의 (3)
11         rear = p; // 그림 5.17 의 (4)
12     }
13 }
```

연결된 큐의 구현

- 삭제 연산

```
1  Element dequeue()
2  {
3      if (is_empty())
4          error("Underflow Error!");
5      Node* p = rear->link; // 그림 5.18 의 (1)
6      if (rear == p) // 노드가 하나인 경우
7          rear = NULL;
8      else // 노드가 둘 이상인 경우
9          rear->link = p->link; // 그림 5.18 의 (2)
10     return free_node(p); // 그림 5.18 의 (3)
11 }
```

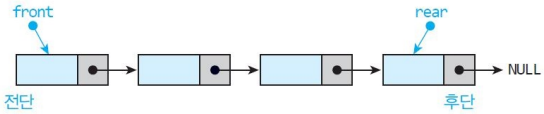
연결된 큐의 구현

- size() 연산

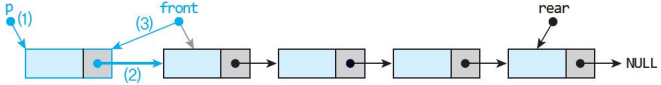
```
1  int size()
2  {
3    int count = 0;
4    if (is_empty()) // 공백인 경우는 0 반환
5        return 0;
6    int count = 1;
7    for (Node* p = rear->link; p != rear; p = p->link)
8        count++;
9    return count;
10 }
```

연결된 덱과 단순 연결 구조의 한계

- 단순 연결 구조를 이용한 덱 (linked deque)



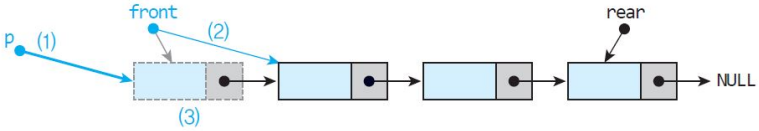
- 전단 삽입을 위한 add_front(e) 연산



1. 요소를 저장할 노드 p 를 생성
2. p 의 link 가 front 를 가리킴
3. front 가 p 를 가리킴

연결된 덱과 단순 연결 구조의 한계

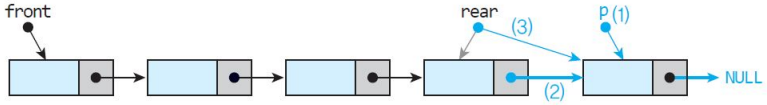
- 전단 삭제를 위한 delete_front() 연산



1. p 는 삭제 노드를 가리킴
2. front 가 다음 노드 (front->link) 를 가리킴
3. p 를 삭제하고 p 의 데이터를 반환

연결된 덱과 단순 연결 구조의 한계

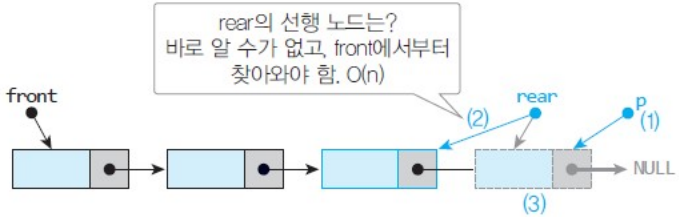
- 후단 삽입을 위한 add_rear(e) 연산



1. 요소를 저장할 노드 p 생성
2. p 의 link 가 rear 를 가리킴
3. rear 가 p 를 가리킴

연결된 덱과 단순 연결 구조의 한계

- 후단 삭제를 위한 delete_rear() 연산: 문제 발생



1. p 는 삭제 노드를 가리킴
2. rear 가 **선행노드??** 를 가리킴

   단순 연결구조에서는 현 노드의 후행노드 정보만 알고 있고 **선행노드를 알 수 없음**
   처음 (front) 부터 탐색하여야 함 $O(n)$
3. p 를 삭제하고 p 의 데이터를 반환

- 단순 연결구조에서 후단 삭제 시, $O(1)$ 구현 불가능
  - 이를 해결하려면 **이중 연결 구조**를 사용해야 함 (다음장에서 다룸)

연습문제

- 5.1 ~ 5.13 (P 192 ~ P 194)